

trustedsec.com /blog/operating-inside-the-interpreted-offensive-python

Unknown Title



TRUSTEDSEC

THE SECURITY BLOG

Introduction

Every once in a while, I get the urge to go back and revisit older techniques that used to be popular but have fallen out of favor with the offensive community. Things like Office Macros, PowerShell, and custom shellcode loaders used to be incredibly effective but are now deemed “burned” by many industry colleagues I chat with. While there is some truth to this, I am still constantly surprising myself and others on my team with so-called “burned” TTPs that prove themselves effective on operations.

Python Malware

In this post, I want to revisit another old technique I believe is a prime candidate to host malware payloads—Python for Windows. But, before we do, let's revisit some existing work in this space.



It seems like hackers have been writing Python malware forever, from Python socket shells to Py2Exe toy implants, but the first Python tradecraft I can remember using is Chris Truncer's [Veil Evasion project](#). Veil was first released in 2013 and still seems like magic—a tool that turns Python into shells.

Shortly after, a golden age of malware began when Microsoft announced PowerShell would come preinstalled on Windows. The original PowerShell empire, PowerSploit, and tons of other tradecraft were written in PowerShell, designed to be used wherever PowerShell was preinstalled. Python malware took a back seat, since the Python interpreter had to be installed on a target before it could be used. After that, Microsoft made some big changes to PowerShell, introducing the Anti-Malware Scanning Interface (AMSI) and various logging capabilities that made developing PowerShell malware just a little bit harder. Eventually, offensive tool developers ditched PowerShell and moved back to unmanaged languages like C and C++.

In late 2018, Microsoft released Python on the Microsoft store (which, at that time, was called the Windows Store). It took me several years to realize how easy it was to install, and I think many others didn't realize this, either. In 2022, Diego Capriotti released [Pyramid](#), an exploitation framework written in Python. In my mind, this marks the first release of modern Python tradecraft. Around the same time, Anthony Rose released an [implementation of a Python 3 engine](#) embedded in .NET, with inspiration taken from Turla's IronNetInjector code. Python code could now be run on Windows without even installing the Python interpreter.

With the easy availability of Python on Windows, as well as some existing Python tradecraft, I believe there is a case to be made for a small but meaningful niche of offensive Python within the malware development space.

Downloading Python From the Microsoft Store

As mentioned before, Python can be easily downloaded from the Microsoft store. On modern Windows systems, simply open the Microsoft store and search for Python. Several versions are available. At the time of writing, Python 3.7 to 3.13 are available. Simply click the "Get" button to download Python. Administrative rights are not required. Alternately, the Microsoft Store has a default protocol handler that you can use if you know the package you want to download. For example, Python 3.13 can be opened using the following URL: `ms-windows-store://pdp/?ProductId=9pnrbtzmb4z`.

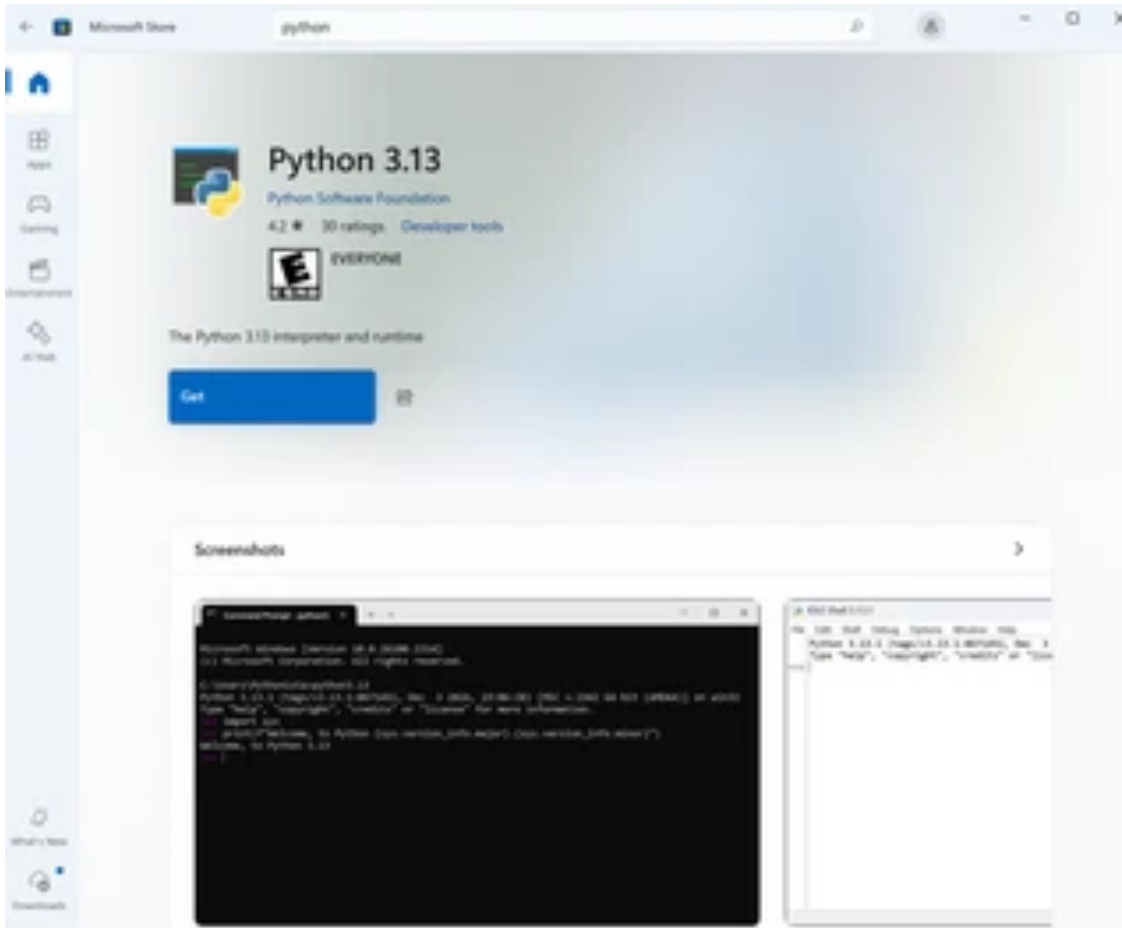


Figure 1 - Python Available in Microsoft Store

After the install, Python can be run from the start menu or a stub executable in the **C:\Users\%AppData\Local\Microsoft\WindowsApps** folder.



Figure 2 - Running Python Downloaded From Microsoft Store

Download and Install Python Offline

There are a few scenarios in which we might not be able to use the Microsoft Store to download Python this way:

- We can only use the command line/No GUI access available
- The Microsoft Store is disabled via Group Policy or other means
- No direct Internet connection on the host

Step 1: Open the desired Microsoft Store application in a browser.



Figure 3 - Opening Python Store Page in Browser

Step 2: Copy the Microsoft Store URL and paste it into <https://store.rg-adguard.net/>. Download the MSIX, APPX, or APPXBUNDLE file, which is the package installer. Note that even though we may not trust this third-party app store, the installer packages are digitally signed, which prevents tampering with the installers. Even so, I recommend installing and testing out the package files in a lab environment before deploying them on the target.

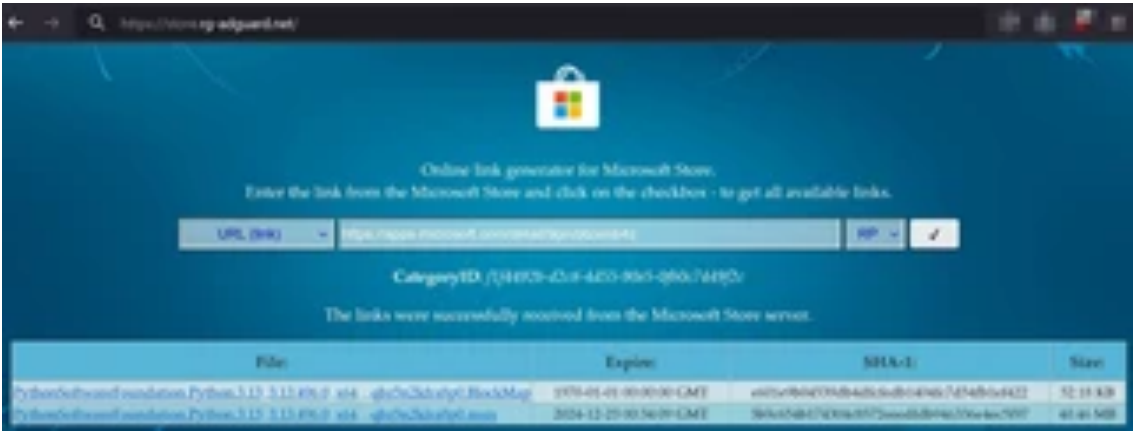


Figure 4 - Getting Python MSIX File From rg-adguard.net

Step 3: Transfer the installer file to the target box. If GUI access is available, simply double-click the file to initiate the installation process. Under the hood, this spawns the program **AppInstaller.exe**.

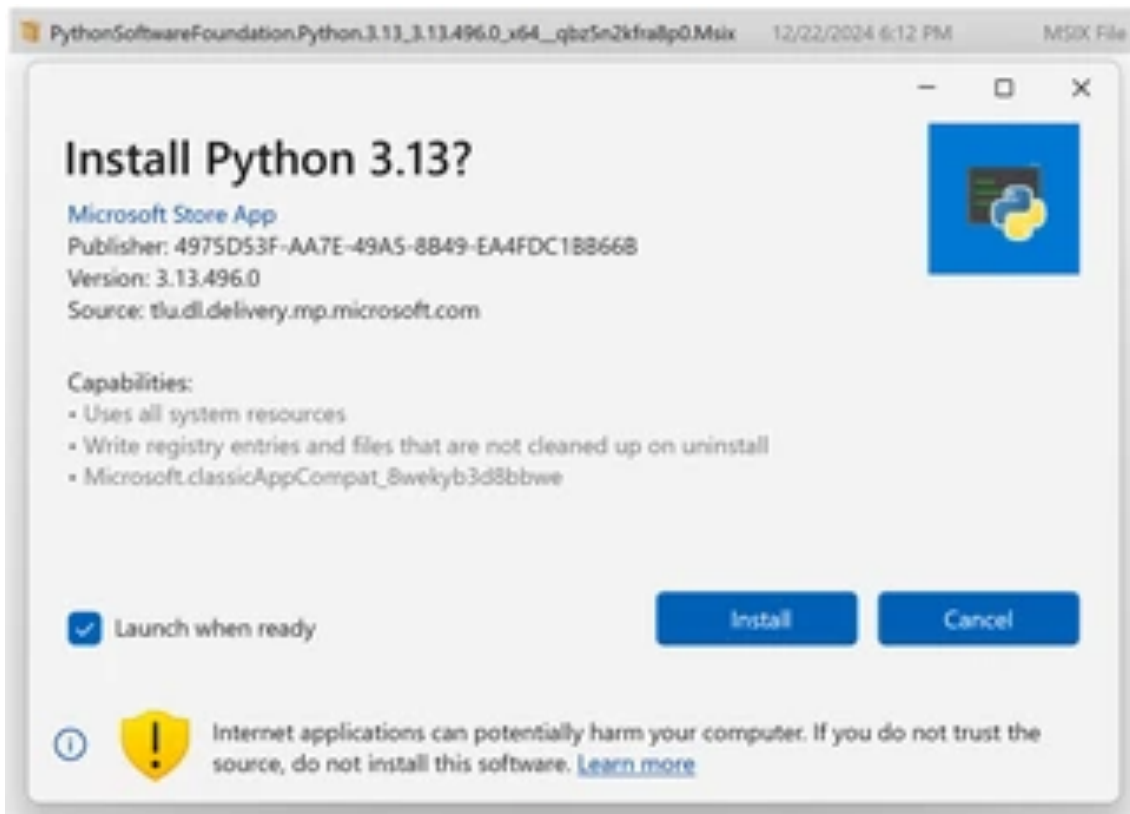


Figure 5 - Double-Click MSIX Installation Prompt

Alternately, the PowerShell **Add-AppxPackage** cmdlet or **dism.exe** can be used to install or uninstall Microsoft Store packages.

```
PS C:\Users\kevinclark\Downloads> Add-AppxPackage .\PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0.MsIX -Verbose
VERBOSE: Performing the operation "Deploy package" on target
"C:\Users\kevinclark\Downloads\PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0.MsIX".
VERBOSE: Operation completed for:
C:\Users\kevinclark\Downloads\PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0.MsIX
```

Figure 6 - Installing MSIX File From PowerShell

```
PS C:\Users\kevinclark\Downloads> Get-AppxPackage *python.3.13*

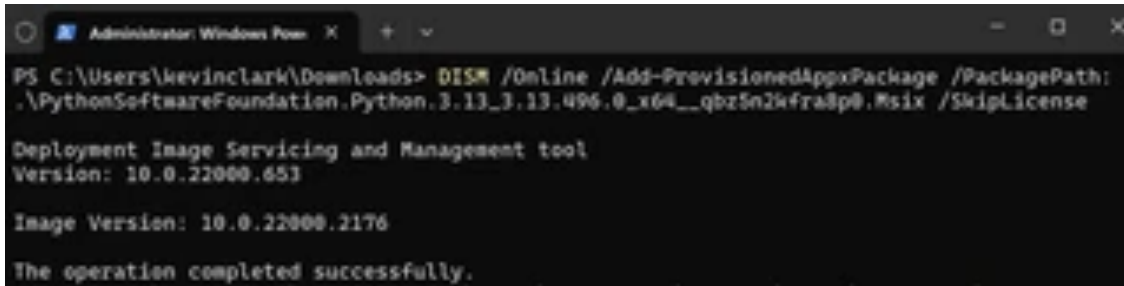
Name                : PythonSoftwareFoundation.Python.3.13
Publisher           : Ck=4975D53F-AA7E-49A5-8B49-EA4FDC1B8668
Architecture        : x64
ResourceId          :
Version             : 3.13.496.0
PackageFullName     : PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0
InstallLocation     : C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0
IsFramework         : False
PackageFamilyName   : PythonSoftwareFoundation.Python.3.13_qbz5n2kfra8p0
PublisherId         : qbz5n2kfra8p0
IsResourcePackage   : False
IsBundle            : False
IsDevelopmentMode   : False
NonRemovable        : False
IsPartiallyStaged   : False
SignatureKind       : Store
Status              : Ok
```

Figure 7 - Showing Python App Installation

```
PS C:\Users\kevinclark\Downloads> Get-AppxPackage *python.3.13* | Remove-AppxPackage -Verbose
VERBOSE: Performing the operation "Remove package" on target
"PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0".
VERBOSE: Operation completed for: PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0
PS C:\Users\kevinclark\Downloads> Get-AppxPackage *python.3.13*
```

Figure 8 - Uninstalling Python via PowerShell

It should be noted that **dism.exe** requires administrative rights to run, even though the package install itself does not require it.



```

Administrator: Windows PowerShell
PS C:\Users\kevinclark\Downloads> DISM /Online /Add-ProvisionedAppxPackage /PackagePath:
.\PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0.Msix /SkipLicense

Deployment Image Servicing and Management tool
Version: 10.0.22000.653

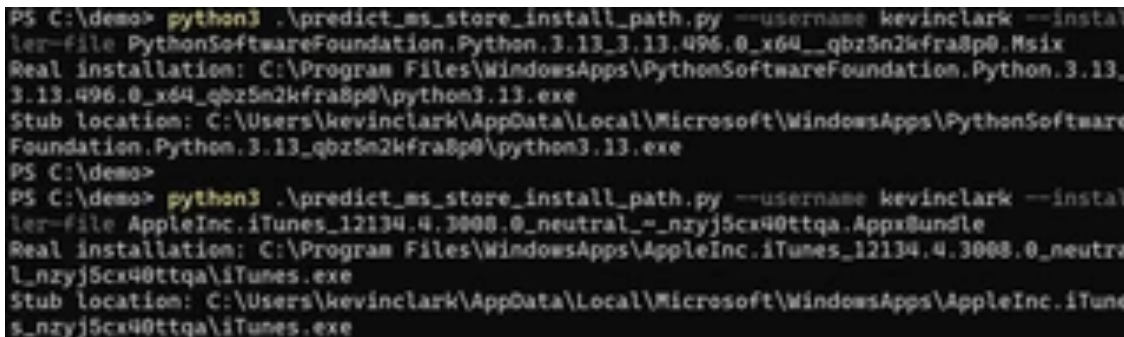
Image Version: 10.0.22000.2176

The operation completed successfully.

```

Figure 9 - Using dism.exe to Install Python From MSIX File

If you're not sure where an MSIX package will be installed, I wrote a [Python script](#) that tries to predict the installation path of the main binary associated with a Windows app package without installing it.



```

PS C:\demo> python3 .\predict_ms_store_install_path.py --username kevinclark --instal
ler-file PythonSoftwareFoundation.Python.3.13_3.13.496.0_x64__qbz5n2kfra8p0.Msix
Real installation: C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.13_
3.13.496.0_x64__qbz5n2kfra8p0\python3.13.exe
Stub location: C:\Users\kevinclark\AppData\Local\Microsoft\WindowsApps\PythonSoftware
Foundation.Python.3.13_qbz5n2kfra8p0\python3.13.exe
PS C:\demo>
PS C:\demo> python3 .\predict_ms_store_install_path.py --username kevinclark --instal
ler-file AppleInc.iTunes.12134.4.3008.0_neutral-__nzyj5cx40ttqa.AppxBundle
Real installation: C:\Program Files\WindowsApps\AppleInc.iTunes.12134.4.3008.0_neutra
l_nzyj5cx40ttqa\iTunes.exe
Stub location: C:\Users\kevinclark\AppData\Local\Microsoft\WindowsApps\AppleInc.iTune
s_nzyj5cx40ttqa\iTunes.exe

```

Figure 10 - Predicted Executable Path for Python and iTunes Applications

Note that the final executable of Python is located at: `C:\Users\kevinclark\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.13_qbz5n2kfra8p0\python3.13.exe`.

In general, this format is: `C:\Users\[username]\AppData\Local\Microsoft\WindowsApps\[company_name].[app_name]_[publisher_id][executable].exe`, where `[publisher_id]` is that unique hash generated from the publisher's certificate, and `[executable]` is the main executable name specified in the `AppxManifest.xml` file inside the package installer file. Although some packages deviate from this structure, most package installations can be predicted with this formula.



Pip Package Manager

The Microsoft store Python package comes with Pip, the Python package manager, installed and can be invoked just by using `Python.exe -m pip`. If network access to the PyPI repos is available, packages can be downloaded and installed normally.

```

PS C:\Users\kevinclark\Downloads> C:\Users\kevinclark\AppData\Local\Microsoft\WindowsApps\
PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\python.exe -m pip install pandas
Defaulting to user installation because normal site-packages is not writeable
Collecting pandas
  Downloading pandas-2.2.3-cp313-cp313-win_amd64.whl.metadata (19 kB)
Collecting numpy>=1.26.0 (from pandas)
  Downloading numpy-2.2.1-cp313-cp313-win_amd64.whl.metadata (60 kB)
Collecting python-dateutil>=2.8.2 (from pandas)
  Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting pytz>=2020.1 (from pandas)
  Downloading pytz-2024.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.7 (from pandas)
  Downloading tzdata-2024.2-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting six>=1.5 (from python-dateutil>=2.8.2->pandas)

```

Figure 11 - Installing a Library via Pip

But many corporate environments block access to the PyPI repos or have web proxies that make downloading packages a challenge. This can make the use of offensive Python a challenge. We must do one (1) of these two (2) things:

- Use Python tools that only rely on the standard library
- Download required libraries offline and transfer them to the host

Although annoying, it is possible to transfer and then install packages on an offline host. I wrote a [simple offline package downloader](#) and a companion installer program that uses only the standard library.

```

usage: offline_package_downloader.py [-h] [-p PACKAGE [PACKAGE ...]] [-r REQUIREMENTS] [-o OUTPUT]
                                     [--os {windows,linux,darwin}] [--arch {32,amd64}]
                                     [--python-version PYTHON_VERSION] [--implementation {cp,pp}]
                                     [-v]

Download Python packages from PyPI into a zip file

options:
  -h, --help            show this help message and exit
  -p PACKAGE [PACKAGE ...], --package PACKAGE [PACKAGE ...]
                        Package name(s) to download (default: None)
  -r REQUIREMENTS, --requirements REQUIREMENTS
                        Path to requirements.txt file (default: None)
  -o OUTPUT, --output OUTPUT
                        Output zip file (default: packages.zip)
  -v, --verbose          Show debug information (default: False)

platform targeting:
  --os {windows,linux,darwin}
                        Target operating system (default: None)
  --arch {32,amd64}      Target architecture (default: None)
  --python-version PYTHON_VERSION
                        Target Python version (e.g., 3.8 or 38) (default: None)
  --implementation {cp,pp}
                        Python implementation (cp=CPython, pp=PyPy) (default: None)

```

Figure 12 - Offline Package Downloader Help Menu

```

PS C:\demo> cat .\requirements.txt
pypykatz
numpy
numba
pandas
lsassy
PS C:\demo> python3 offline_package_downloader.py -r requirements.txt
Output zip file will be: C:\demo\packages.zip
Target platform: Windows, 64-bit, Python 3.12, CPython

Processing pypykatz
Downloaded pypykatz-0.6.10-py3-none-any.whl
Dependencies: unidump<=0.1.0,>=0.0.10, minidump<=0.1.0,>=0.0.21, minikerberos<=0.5.0,>=0.4.1, alsewin
reg<=0.1.0,>=0.0.11, msldap<=0.6.0,>=0.5.7, winacl<=0.2.0,>=0.1.9, alsewb<=0.5.0,>=0.4.8, aessed<=0.2.
0,>=0.1.4, tqdm

Processing unidump<=0.1.0,>=0.0.10
Downloaded unidump-0.0.10-py3-none-any.whl
Dependencies: pycryptodome

Processing pycryptodome
Downloaded pycryptodome-3.21.0.tar.gz

Processing minidump<=0.1.0,>=0.0.21
Downloaded minidump-0.0.24-py3-none-any.whl

```

Figure 13 - Downloading Pip Wheels With Dependencies

After packages have been downloaded, the script zips them up into a single file that can either be uploaded to the offline host directly or hosted on a web server. The installation script is then used to download the packages' ZIP file and install each Pip wheel.

```
PS C:\demo> python3 .\install_packages.py
usage: install_packages.py [-h] source
install_packages.py: error: the following arguments are required: source
PS C:\demo> python3 .\install_packages.py -h
usage: install_packages.py [-h] source

Install Python packages from a zip file (URL or local path)

positional arguments:
  source          URL or path to packages.zip
```

Figure 14 - Offline Package Installer Help Menu

```
PS C:\demo> python3 .\install_packages.py packages.zip
Extracting packages.zip

Found 36 package(s)

Installing aeseDB-0.1.6-py3-none-any.whl
Defaulting to user installation because normal site-packages is not writeable
Processing c:\users\kevinc-l\appdata\local\temp\tmpdyw1s3e\aesedb-0.1.6-py3-none-any.whl
Requirement already satisfied: aiohttp>=0.0.7 in c:\users\kevinc-l\appdata\local\packages\pythonsofwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-packages\python312\site-packages (from aesedb==0.1.6) (0.0.12)
Requirement already satisfied: colorama in c:\users\kevinc-l\appdata\local\packages\pythonsofwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-packages\python312\site-packages (from aesedb==0.1.6) (0.4.6)
Requirement already satisfied: tqdm in c:\users\kevinc-l\appdata\local\packages\pythonsofwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-packages\python312\site-packages (from aesedb==0.1.6) (4.67.1)
Requirement already satisfied: unicycrypto>=0.0.9 in c:\users\kevinc-l\appdata\local\packages\pythonsofwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-packages\python312\site-packages (from aesedb==0.1.6) (0.0.10)
Requirement already satisfied: prompt-toolkit>=3.0.2 in c:\users\kevinc-l\appdata\local\packages\pythonsofwarefoundation.python.3.12_qbz5n2kfra8p0\localcache\local-packages\python312\site-packages (from aiohttp>=0.0.7->aeseDB==0.1.6) (3.0.48)
Requirement already satisfied: winact>=0.1.7 in c:\users\kevinc-l\appdata\local\packages\pythonsofwe
```

Figure 15 - Performing Install From Offline Packages' ZIP File

Python Standard Library

We can see it's possible to download and install libraries onto hosts without a direct connection to the PyPI repos. Otherwise, it is still a good practice to try to limit offensive tooling to standard library packages if possible. A good example of this is using the less user-friendly *urllib* instead of the *requests* library.

Many other useful capabilities are built into the Python standard library. Here is a list of a few things offensive developers might want to do and the library they are implemented in:

Capability	Standard Library Module
Web requests	urllib
TCP connections	socket
Sha256 hashing	hashlib
Compression	gzip, bz2, zlib, zipfile
Load DLLs and call unmanaged functions ctypes	
Spawn processes & run commands	subprocess
Process structured data	json, xml
Access & modify the Registry	winreg
Encode and decode binary data	base64
IP address manipulation	ipaddress

But Python doesn't provide everything we might need in the standard library. Some important things require using a library or need to be implemented by hand:

Capability	Third-Party Library Module
AES encryption	pycryptodome
Web socket connections	aiohttp
Load the CLR & interact with managed code	pythonnet
Packet captures	libpcap
LDAP interaction	ldap3
Windows protocols	impacket

A full list of packages that come default with a Windows Python installation [can be found here](#).

Ctypes for Unmanaged Execution



One of my favorite Python modules is **ctypes**. Along with the **struct** library, **ctypes** lets a Python programmer write code in the style of C, including loading unmanaged libraries and exports, manual memory management, and performing real pointer operations. If you are familiar with PInvoke for C#, **ctypes** is essentially the same thing for Python.

So, how do we use **ctypes** to call a Win32 API? There are five (5) steps:

- Load a DLL providing the function we want
- Get a handle to the function we want to call
- Define the function prototype, including parameters and return type
- Marshal Python parameters into C type parameters
- Call the function
- (Optional) Marshal the return value back to a Python type for use with Python code

For Win32 DLLs, you should use `ctypes.windll.dllname` or `ctypes.WinDLL("dllname")` to load a DLL. To load a DLL from a specific file, use `ctypes.CDLL(r"C:\path\to\dll.dll")`.

```
kernel32 = ctypes.windll.kernel32          # Load kernel32.dll
kernel32_2 = ctypes.WinDLL("kernel32")     # Load kernel32.dll again
custom_dll = ctypes.CDLL(r"C:\test\custom.dll") # Load custom.dll
```

After loading a DLL, you'll need to find an export you want to call. In this example, I will be using the **CreateFileW** API from **kernel32.dll**. DLL exports can be found by name or by export ordinal number. Most programmers prefer to import by name, since it is easier and more reliable across Windows versions.

```
CreateFileW = kernel32.CreateFileW # Get exported function handle
```

```
CreateFileW_ord = kernel32[212]          # Get function by ordinal. Not recommended since these values can change between Windows versions
```

After we have a function pointer, we need to define the function prototype, which includes the function parameters and the return type. The **wintypes** submodule is nice enough to have all of the common Windows primitive types defined for us, so be sure to import that. Set the **argtypes** list that defines the parameter types, and then set the **restype** value to be whatever the function should return. In the case of **CreateFileW**, the parameters and return types are [clearly defined on MSDN](#).

```
# MSDN documentation for CreateFileW:
#
# HANDLE CreateFileW(
#     [in] LPCWSTR          lpFileName,
#     [in] DWORD            dwDesiredAccess,
#     [in] DWORD            dwShareMode,
#     [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
#     [in] DWORD            dwCreationDisposition,
#     [in] DWORD            dwFlagsAndAttributes,
#     [in, optional] HANDLE hTemplateFile
# );
#

from ctypes import wintypes

# ctypes function definition for CreateFileW:
#
CreateFileW.argtypes = [
    wintypes.LPCWSTR,    # filename
    wintypes.DWORD,      # desired access
    wintypes.DWORD,      # share mode
    wintypes.LPVOID,     # security attributes
    wintypes.DWORD,      # creation disposition
    wintypes.DWORD,      # flags and attributes
    wintypes.HANDLE      # template file
]
CreateFileW.restype = wintypes.HANDLE
```

Finally, we need to make some data to give to the **CreateFileW** function, marshal the data as **wintypes**, and then execute the function. Note that **ctypes** performs some conversion of data between Python types and C types automatically for us.

```
# Define the test data
filename = "test.txt" # Test file name
GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
CREATE_ALWAYS = 2
FILE_ATTRIBUTE_NORMAL = 0x80

# Execute the function with test data
handle = CreateFileW(
    filename, # Auto converted from Python str to wintypes.LPCWSTR
    GENERIC_READ | GENERIC_WRITE,
    0,
    None, # Translates to NULL when passed to CreateFileW
    CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL,
    None
)
```

That's it! A file named **test.txt** was created in the current directory via Python's **ctypes** module.

Ctypes Examples

Here is the full code for a Python program that displays the current process ID and name in a MessageBox window.

```
import ctypes
from ctypes import c_char_p, c_uint32, c_void_p, create_string_buffer
import os.path

# Windows API constants
MAX_PATH = 260
MB_OK = 0x00000000
MB_ICONINFORMATION = 0x00000040

# Load DLLs
kernel32 = ctypes.windll.kernel32
user32 = ctypes.windll.user32

# Define function prototypes
GetCurrentProcessId = kernel32.GetCurrentProcessId
```

```

GetCurrentProcessId.restype = c_uint32
GetCurrentProcessId.argtypes = []

GetModuleFileNameA = kernel32.GetModuleFileNameA
GetModuleFileNameA.restype = c_uint32
GetModuleFileNameA.argtypes = [c_void_p, ctypes.c_char_p, c_uint32]

MessageBoxA = user32.MessageBoxA
MessageBoxA.restype = ctypes.c_int32
MessageBoxA.argtypes = [c_void_p, c_char_p, c_char_p, c_uint32]

# Get process ID using Windows API
process_id = GetCurrentProcessId()

# Get process name using Windows API
buffer = create_string_buffer(MAX_PATH)
path_length = GetModuleFileNameA(None, buffer, MAX_PATH)
if path_length == 0:
    raise Exception("Failed to get module filename")

# Convert buffer to string and get just the filename
full_path = buffer.value.decode('ascii')
process_name = os.path.basename(full_path)

message = f"Process Name: {process_name}\nProcess ID: {process_id}"

MessageBoxA(
    None,
    message.encode('ascii'),
    b"Process Information",
    MB_OK | MB_ICONINFORMATION
)

```

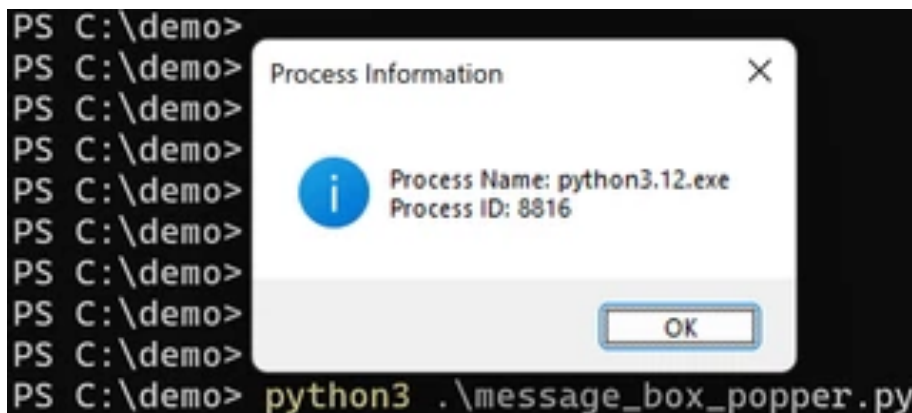


Figure 16 - Popping a MessageBox in Python

Finally, I wrote a [simple reflective DLL loader](#) as a more practical example of Python malware. While this loader is not sufficient for real operations, it clearly demonstrates an essential component of attacker tradecraft implemented in Python.

```

PS C:\demo> python3 .\load_dll.py
Usage: python3 load_dll.py </path/to/dll.dll> [export]

Description:
  Loads the specified DLL into memory and optionally calls an exported function.
  If no export is specified, the DLL is loaded, and only DllMain is executed.

Examples:
  python3 load_dll.py example.dll           # Load DLL and execute DllMain
  python3 load_dll.py example.dll Run       # Load DLL and call 'Run'

```

Figure 17 - Reflective DLL Loader Help Menu

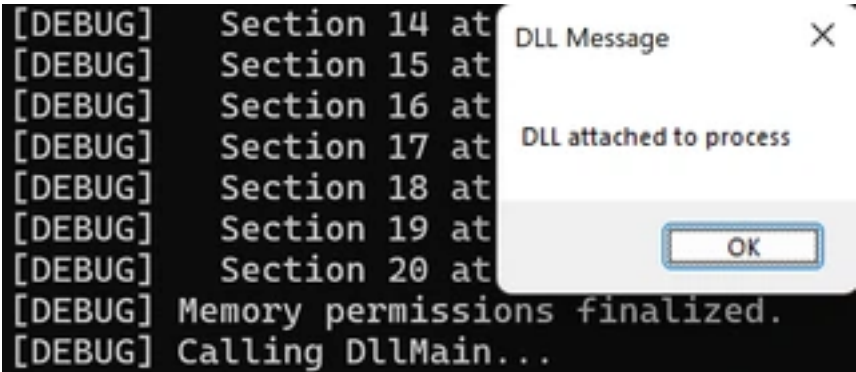


Figure 18 - Calling DllMain on Reflectively Loaded DLL

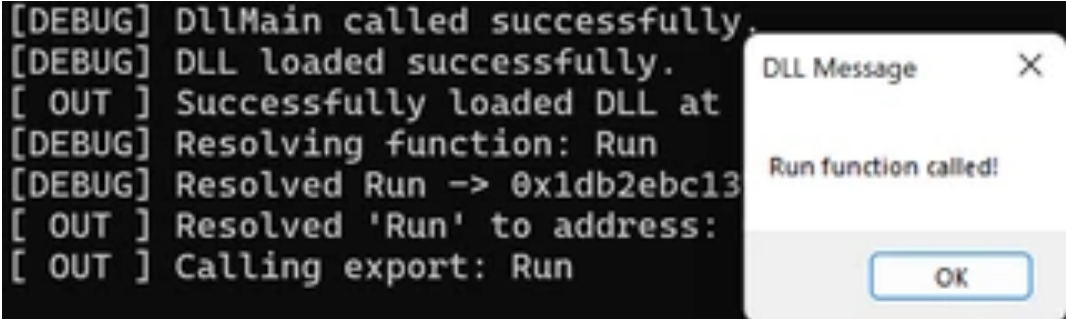


Figure 19 - Finding and Executing Exported Run Function

IoCs Inside of Python

Python is a legitimate application used in enterprise networks around the world. It has a good reputation, and the *python.exe* binary is signed by Microsoft and the Python Software Foundation. It's also quite common to see Python already installed on workstations and servers, making it unnecessary to install it at all.

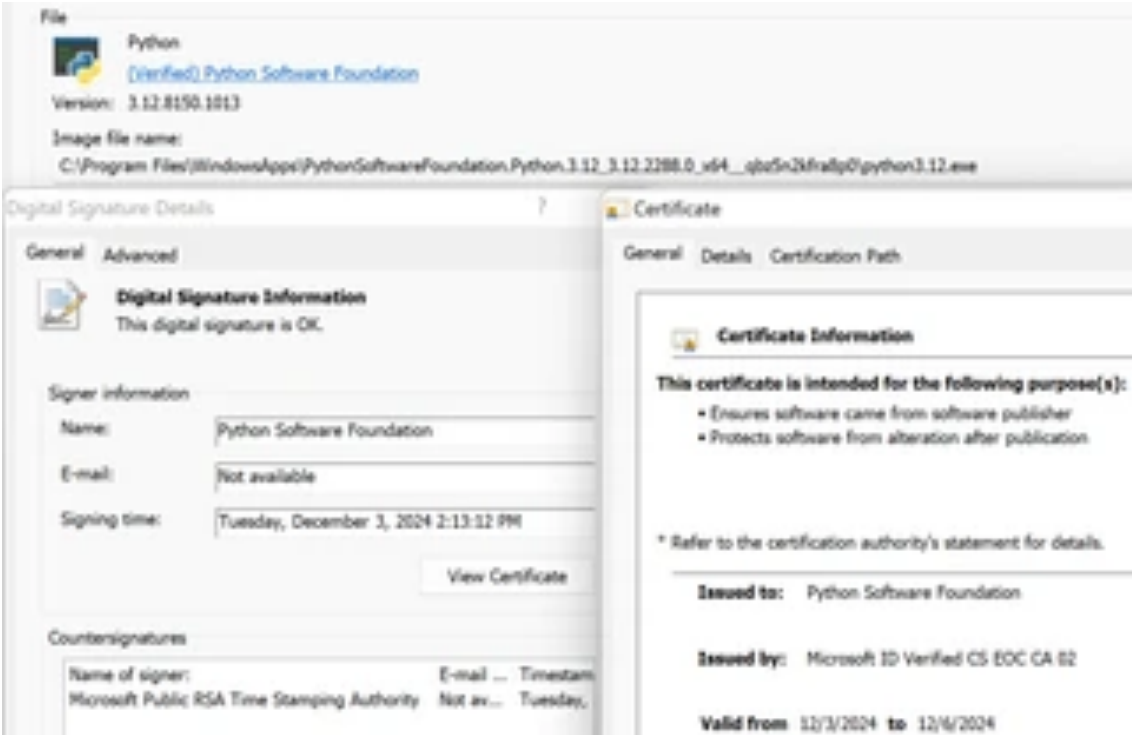


Figure 20 - Valid Code Signing Cert for python.exe

Like many interpreted languages, Python has weird memory indicators that come with dynamically generated code. The default Windows CPython installation creates unbacked executable memory sections and Read-Write-Execute memory even before any Python code is executed.

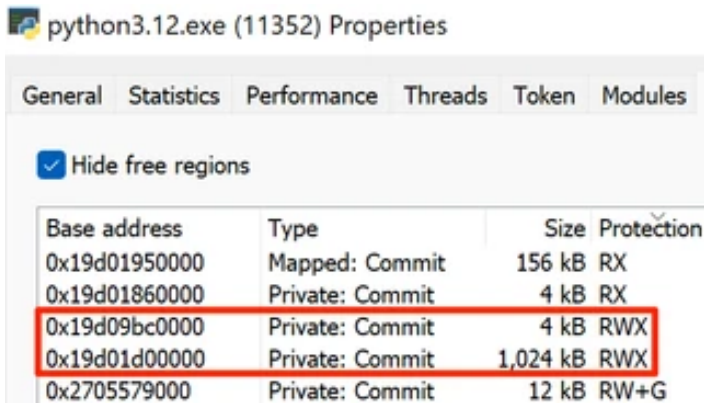


Figure 21 - Read-Write-Execute Memory Found in python.exe

Yet, best of all, Python is a general-purpose language, meaning its uses are practically endless, ranging from data science to system administration, to videogame programming, to DevOps orchestration work and so much more. What, then, is normal for a *python.exe* process to be doing? Should *python.exe* be connecting to websites on the Internet? What about socket connections to other systems? How about allocating or modifying sections of memory? Should *python.exe* be consuming a large amount of memory or CPU?

These questions can concretely be answered with a strong “maybe” or “it depends on what Python is doing.” Although not impossible, all of these factors combined make it more difficult for endpoint products to baseline normal behavior and make it easier for attackers and operators alike to live inside of *python.exe*.

Conclusion

Python as a platform for malware deployment is often an undervalued target. It is easy to install, provides high-quality built-in libraries, and is an excellent target process for operations. It won't replace current unmanaged operational techniques but provides a convenient and practical alternative when other techniques are not feasible.