An In-memory Embedding of CPython for Offensive Use

Ateeq Sharfuddin, Brian Chapman, Chris Balles
SCYTHE
{ateeq, brian, chris}@scythe.io

arXiv:2103.15202v2 [cs.CR] 4 May 2021

***Abstract*—We offer an embedding of CPython that runs entirely in memory without "touching" the disk. This in-memory embedding can load Python scripts directly from memory instead these scripts having to be loaded from files on disk. Malware that resides only in memory is harder to detect or mitigate against. We intend for our work to be used by security researchers to rapidly develop and deploy offensive techniques that is difficult for security products to analyze given these instructions are in bytecode and only translated to machine-code by the interpreter immediately prior to execution. Our work helps security researchers and enterprise Red Teams who play offense. Red Teams want to rapidly prototype malware for their periodic campaigns and do not want their malware to be detected by the Incident Response (IR) teams prior to accomplishing objectives. Red Teams also have difficulty running malware in production from files on disk as modern enterprise security products emulate, inspect, or quarantine such executables given these files have no reputation. Our work also helps enterprise Hunt and IR teams by making them aware of the viability of this type of attack. Our approach has been in use in production for over a year and meets our customers' needs to quickly emulate threat-actors' tasks, techniques, and procedures (TTPs).**

***Keywords*-Computer security, Red Team, in-memory, Python**

I. Introduction

Different classes of threat-actors have different motivations, and therefore, produce and utilize different types of malware to complete their goals. Nation-states are the most sophisticated threat-actors and go through great lengths to ensure the end-user is not alerted to their intrusions. As a consequence, whereas cyber-criminals are not so concerned about leaving artifacts of compromise behind, malware produced by nation-states aim to leave no artifact behind and have built-in security hindering forensic analysis. One method threat-actors use is to keep much of the running malware only in volatile memory, for example, a system's RAM. This way, when the system shuts down or is restarted, there is no disk artifact of the intrusion as state is lost in volatile memory when power is removed from the system.

Python is a popular interpreted high-level programming language [24] with an emphasis on readability and an abundance of reusable packages produced by the community. CPython is the default and most widely used implementation of the Python language, written in C and Python. With the `ctypes` package, CPython is as expressive as C, allowing non-interpreted native instruction execution. One common scheme unsophisticated threat-actors use is to write their malware in Python and package the malware and all its dependencies into an executable using packagers like Py2Exe [6] or PyInstaller [4]. These malware are easy to construct and are executed when a coerced end-user double-clicks and runs the packaged executable file from disk. Security products have collected and analyzed numerous sample executable files of this scheme. As a result, security products have produced signatures for this scheme and are able to successfully block all malware using this scheme.

In this paper, we contribute the first embedding of CPython interpreter that runs entirely in memory without "touching" the disk. We also contribute a scheme to support loading and running Python modules including native C extensions, also residing only in memory, onto this interpreter. We are able to achieve these results without needing any file or dependency to reside on disk, unlike the Py2Exe or PyInstaller packaging schemes. We have not come across any other in-memory embeddings of Python. Constructing this CPython embedding was an engineering effort: We claim no new algorithms or techniques. Our paper details the necessary steps to successfully load and utilize an in-memory embedding of CPython.

Our goal is to assist security researchers and enterprise Red Teams. Enterprise Read Teams are responsible for emulating numerous classes of threat-actors. Red Teams prefer to repurpose existing malware or open-sourced security experiments. Red Teams have deadlines to meet and run campaigns with periodicity: As such, they want to emulate different threat-actors quickly, and this is more time-consuming to accomplish in a binary compiled from C or assembly than executing statements in a Python interpreter. These malware also need to evade Hunt and Incident Response teams until the objectives for the campaign are at least partially met. Red Teams also have difficulty running malware from files on disk as modern enterprise security products emulate, inspect, or quarantine such executables given that these files have no reputation. A month-long campaign effort can be rendered worthless on the first day of use if a Red Team's artifact is immediately caught upon execution by the Hunt or IR teams. We offer this information in the interest of raising awareness of the viability of this type of attack. As a simplification, we do not cover any additional security or obfuscation mechanisms that a threat-actor would use, for example, changing bytecode mappings or preventing memory dumps, nor do we cover how to land onto the virtual address space of a process: These well-known topics have been covered by other researchers [16], [23]. Both Windows and POSIX implementations are in use in

https://t.me/learningnets

production. As a simplification, we only extensively cover the steps for Windows. However, similar steps apply to POSIX, and we describe differences in Section V-E.

Section II describes prior research relevant to this paper. Section III details the in-memory CPython interpreter. Section IV details loading Python scripts from memory into this in-memory interpreter. Section V details in-memory dynamic loading of native Python C Extensions into this in-memory interpreter. Section VI covers emulation of a few security techniques with this interpreter. Section VII covers future work.

II. Background

We cover prior work relevant to this paper in this section. Specifically, we cover related Python embeddings, dynamic loading techniques, and in-memory loading of Python modules. We utilize the in-memory dynamic loading technique derived from a prior work [22] in order to load native C Python Extensions into memory without "touching" the disk: This does not need to be reinvented.

A. Contemporary Python Embeddings

Numerous malware have been publicized that use the Py2Exe or PyInstaller methods [7]. Signatures have been produced for these malware, and often security products simply quarantine any Py2Exe or PyInstaller-generated executable without inspecting if the encapsulated payload is malicious or benign. One malware was publicized in [23] where the malware-authors recompiled the CPython interpreter and remapped the opcodes to make analysis difficult: In this case, this recompiled interpreter was still an executable file on disk. In [8], the authors take a peek inside Dropbox's executable file and use a reflective loader [3] to inject their shared library into the Dropbox process's address space: The Dropbox executable file was also a Python interpreter. These listed cases differ from our approach: We show how to download the interpreter into a location in memory and then run the interpreter from there. Our approach does not "touch" the disk, meaning no intermediate step exists where the interpreter or any dependency is saved as a file to disk in order for the modified CPython interpreter to function.

B. Dynamic Loading

The documented way to load a shared library at runtime (also known as dynamic loading [13]) is to use functions provided by operating system APIs such as `LoadLibrary` on Windows or `dlopen` on Linux. These functions require the shared library to be a file on disk. However, it is entirely possible to load these directly from a location in memory, and well-known approaches are described in [11], [22]. Implementations are available online [1], [3]. We employ one of these approaches to load the Python C extensions, which are native shared libraries, directly from a process's address space. The interpreter is compiled only once, and you can load modules on-demand into the interpreter at run-time with this approach. An alternative approach would have been to "freeze" [9] these Python C extensions and compile them into the interpreter. However, this is not a flexible approach: You must recompile the interpreter each time for a different set of "frozen" modules.

C. Python module Loaders

The official stock CPython interpreter loads modules and packages from files on disk using its *PathFinder* importer object. The interpreter also contains a frozen *zipimport* [20] module that is limited to importing a single package or module from a ZIP file on disk. This *zipimport* cannot import Python C extensions, either. We constructed our `cba_zipimport` derived from *zipimport* that can load multiple packages, modules, and Python C extensions from an in-memory ZIP archive.

III. In-memory Embedding of Python

We cover the concept of embedding the Python interpreter in another program in this section. The Python interpreter is available as a shared library. We cover loading a shared library from memory, and although this is prior art, we feel it is nonetheless important to summarize. Note that using an in-memory dynamic loading method [22] to load this Python interpreter shared library into memory is not enough to initialize and start invoking Python scripts. We list the additional necessary steps that lead to a successful initialization of the interpreter. We concluded that these were the only steps necessary in an iterative fashion by running the CPython interpreter through a debugger observing failures, reading the CPython source code, and reading official CPython documentation, in that order.

A. Embedding Python

The Python interpreter was designed to be embedded into applications. Embedding allows an application developer to implement some application functionality in the Python programming language rather than in C or C++ [17]. An application initializes the embedded Python interpreter with a call to `Py_Initialize`. The interpreter can then be called from any part of the application. Simple and pure embeddings are covered in detail in [17]. The Python interpreter itself is implemented in a shared library. For example, the Python 3.8 core shared library on Windows is python38.dll, and the main Python application (python.exe) loads this shared library and provides a read-eval-print loop (REPL) interface to the user. To embed the Python interpreter into an application, simply link this Python core shared library during compile-time (using static linking or dynamic linking) or load it from the application at run-time (with dynamic loading).

B. Loading Shared Libraries from Disk

There are three mechanisms a program can choose to use other software [26]: static linking, dynamic linking, and dynamic loading. Dynamic loading allows a program to, at run time, load a shared library into memory, retrieve addresses of functions and variables, execute these functions or access these variables, and unload the library from memory when

https://t.me/learningnets

the application no longer needs the library. Shared libraries such as Dynamic-Link Libraries (.dll files) on Windows or Shared Object Libraries (.so files) on Linux can be dynamically loaded at runtime. As a simplification, Windows API provides a C function `LoadLibrary(FilePath)` to load a shared library into a process's address space. The C function `GetProcAddress(ModuleHandle, FunctionName)` can then be used to retrieve the address of an exported function (or variable) in the shared library. If the function is found, the application code can now call this function. Finally, the application can call `FreeLibrary(ModuleHandle)` to unload the shared library from the process's address space when no longer in use [15]. In the POSIX standard API, the equivalent C functions are `dlopen`, `dlsym`, and `dlclose` [26], respectively.

C. Loading Shared Libraries from Memory

Although official documentation exposes APIs where the shared libraries must be files on disk, research has been published [22] that demonstrate that shared libraries can in fact be dynamically loaded directly from memory and reference implementations exist [1]. These research provide details necessary to reimplement the operating system's loader and offer three replacement C functions: `LoadLibraryFromMemory(ModuleBytes)`, `GetMemoryProcAddress(Handle, FunctionName)`, and `FreeInMemoryLibrary(Handle)`. With such a reimplementation of the operating system's loader we can load the Python core shared library directly into a process's address space.

D. Preparation

Prior to loading the shared library into a process's address space, the process itself must be compromised: We write and execute instructions to download the in-memory loader code and the shared library, and proceed to loading this shared library. We do not detail how to compromise a process in this paper: An extensive amount of prior art exists in this area [10], [12] and compromising a process does not have a one-solution-fits-all answer nor is this the topic of this paper. This paper assumes you already control the instruction pointer in a process, wrote to memory to download the loader code and the shared library from the network, and have successfully loaded this shared library with a call to `LoadLibraryFromMemory`. We describe modifications needed to specifically run the Python interpreter after loading its shared library.

IV. Initializing the Interpreter

After successfully loading the Python core shared library into memory, we now need to initialize the embedded interpreter in it. Instead of using the traditional `Py_Initialize` call to initialize the embedded interpreter we use `Py_InitializeFromConfig`. The configuration structure is initialized with a call to `PyConfig_InitIsolatedConfig`. In isolated mode the embedded Python interpreter ignores environment variables, global configuration variables, command line arguments, and the user site directory. This is of particular importance to us: We want to run a Python interpreter that does not check for an existing Python installation on the system. Specifically, not all systems will have Python pre-installed, the version of Python pre-installed on the system may not match the version of the embedded interpreter, or loading packages from the pre-installed Python will produce auditable events (for example, Sysmon Event 11 FileCreate [21]) that may raise alarms. As an example, if hypothetically, the system calculator (calc.exe) starts an embedded Python, and it reads a collection of py files in rapid succession, it is suspicious.

A. Required Modules

Starting the embedded Python interpreter in-memory in isolated mode will fail, as the interpreter needs to load modules and packages such as *encodings*, *codecs*, *abc*, etc. and cannot find these in the path: We must make changes to the embedded Python interpreter so that it can load modules and packages it needs directly from memory instead. We put all the requisite packages and modules in a ZIP archive and transmit this ZIP archive along with the interpreter to the process (as explained in Section IV-E). The embedded interpreter loads the necessary packages and modules from this in-memory ZIP archive. With this approach, we do not have to transmit each package or module individually. The CPython provided builtin *zipimport* [20] module can import Python modules and packages from ZIP files. However, *zipimport* does not support importing from ZIP archives in memory. Nor does *zipimport* support importing multiple modules and packages residing in the same ZIP archive or importing Python C extensions in a ZIP archive. We wrote a derivative of Python's builtin *zipimport* module, to remove these limitations: The `cba_zipimport` module loads packages and modules from ZIP archives in memory and can also load Python C extensions.

B. cba_zipimport MetaPathFinder

PEP 302 [25] describes the concept of *metapath* and its implementation provides a list of importer objects (in *sys.meta_path*) that are traversed to import packages and modules. The last object in *sys.meta_path* is the *PathFinder* object, which loads modules and packages from files on disk. The entire import system is exposed via *sys.meta_path* with no implicit machinery [18]. We implemented the importer `cba_zipimport`, and inserted it into *sys.meta_path*. This way, when the interpreter traverses *sys.meta_path* to load a module or package, it will inquire `cba_zipimport` to load this module. The order in which `cba_zipimport` will be called depends on its position in the *sys.meta_path* list.

The `cba_zipimport` module provides an implementation of *importlib.abc.MetaPathFinder* as *finder* and an implementation of *importlib.abc.Loader* as *loader*. The interpreter will use this *finder* to locate the *loader* for a module. The interpreter uses the *loader* to then load the module. The `cba_zipimport` module also provides functionality, which when invoked inserts `cba_zipimport` as an importer object into *sys.meta_path*. The underlying implementation of parsing

https://t.me/learningnets

ZIP archives in `cba_zipimport` remains exactly the same as Python's *zipimport*.

C. Adding to Builtin Frozen modules

We "freeze" [9] `cba_zipimport` and add `cba_zipimport` to the table referenced by `PyImport_FrozenModules`. This way, the Python interpreter already will have the "frozen" `cba_zipimport` and will not try to search for it using *PathFinder* on the system's disk.

D. PyLifecycle Updates

We slightly update pylifecycle.c: Prior to `init_importlib_external` calling `_PyImportZip_Init`, we add a call to `_PyCBAImport_Init` function. `_PyCBAImport_Init` imports our `cba_zipimport` module using the interpreter's C function `PyImport_ImportModule`. If import is successful, `_CBAZipImport_Init` calls the module's function `install_cba_metafinder`. This function instantiates a global instance of *loader* and *finder*, and inserts *finder* in *sys.meta_path* at offset 2. That is, this *finder* is the third in the list of finders to be called should the interpreter need to load a module: The two importer objects preceding this *finder* are Python's `BuiltinImporter` and `FrozenImporter`. As an example, if the interpreter were looking for *codecs.py*, it would first attempt to find its loader with *BuiltinImporter*. If *BuiltinImporter* cannot provide one, it would attempt to find one with *FrozenImporter*. If *FrozenImporter* cannot provide one, it would attempt to find one with *finder*. If *finder* could not provide one, the interpreter would attempt to find *codecs.py* with *PathFinder*.

E. Bundling Python Libraries

Since a stock installation of Python contains a prepackaged collection of modules, we want the in-memory Python interpreter to also offer this same collection of modules. We create a ZIP file, cba_python38_lib.zip. We put everything under Python's Lib directory into cba_python38_lib.zip: These are standard Python modules, which have .py or .pyc extensions. We constructed an xxd.py, which behaves similar to the xxd application, and used this xxd.py to output a C source file with the contents of the aforementioned ZIP file stored in a C array. We compile this C source file into the Python core shared library. When `_CBAZipImport_Init` calls the `cba_zipimport` module's function `install_cba_metafinder`, this C array is provided as an input parameter.

F. Result

At this point, the in-memory Python interpreter has everything it requires to be loaded directly from memory. To summarize, the in-memory loader loads the Python core shared library hosting the Python Interpreter. We initialize the interpreter calling it with an isolated configuration. The interpreter initializes and loads modules, and initializes builtin modules and frozen modules. The `cba_zipimport` module, which is frozen, is loaded and it's *finder* is added to Python's *sys.meta_path* at offset 2. At this point, the interpreter requests to load modules, and this *finder* resolves these requests and successfully loads them from the in-memory ZIP archive that is bundled into the interpreter. All the necessary modules are found, and the in-memory Python interpreter is now ready to execute Python statements.

V. Python C Extensions

In specific scenarios, calling native code from Python is necessary. Operating systems come with a large number of APIs that aren't present in the Python interpreter, an example scenario being APIs to access hardware or operating system management functions. Similarly, there may be a reason where the Python interpreter needs to communicate with components written in another language that produces a C-style native application binary interface (ABI), an example scenario being some Java code exposed via Java Native Interface (JNI). Python C Extensions offer a way to extend Python itself: Implement native code in the extension and call it from the Python interpreter. For example, the *ssl* Python package imports *_ssl*, which is a C extension linked with the native OpenSSL shared library that calls operating system APIs.

For our approach, when a Python C Extension depends on a shared library, we statically link the shared library into the extension. This rule excludes shared libraries that are guaranteed to exist on the system, for example, kernel32.dll on Windows. Using the previous example, the C extension _ssl.pyd for Python 3.8 expects the shared library libcrypto-1_1.dll to be in the path. In our approach, the in-memory interpreter cannot expect to resolve libcrypt-1_1.dll from disk. Therefore, it is appropriate to statically link OpenSSL into _ssl.pyd.

All the Python C Extensions that come bundled with the standard CPython interpreter are recompiled such that the non-system shared library dependencies are statically-linked. These C Extensions are put in a ZIP file cba_python38_pyd_lib.zip. We use xxd.py to output a C array with the contents of the aforementioned ZIP file. We then compile this C source file into the Python core shared library. `_PyCBAImport_Init` calls the `cba_zipimport` module's function `install_cba_metafinder` with this C array as input parameter, similar to what we did for the Python modules that were not C extensions. With this step the in-memory interpreter has the same collection of modules and packages as a stock installation of the official Python interpreter.

A. Loading C extensions in ZIP archives

1) The _zip_searchorder structure: We update the `_zip_searchorder` structure in `cba_zipimport` to track if a Python module is native. The `cba_zipimport` is derived from Python's *zipimport*: We describe the alterations we made in our derivation. The elements in each tuple are: suffix, is bytecode, is package, is native. In the structure as shown below, files with a .pyd extension are neither bytecode, nor a package, but are native.

https://t.me/learningnets

Listing 1
_ZIP_SEARCHORDER STRUCTURE

```
# (extension, isbytecode, ispackage, isnative)
_zip_searchorder = (
  (path_sep + '__init__.pyc', True, True, False),
  (path_sep + '__init__.py', False, True, False),
  ('.pyc', True, False, False),
  ('.py', False, False, False),
  ('.pyd', False, False, True),)
```

The functions `_get_module_info` and `_get_module_code` use this *_zip_searchorder* structure to determine if the lookup resolves to a package, a module, or neither. The function `load_module` calls `_get_module_code` and has been updated to call the native loader if `_get_module_code` indicates the filename in the ZIP archive being loaded is native.

Listing 2
LOAD_MODULE SNIPPET

```
...
code, ispackage, modpath, isnative =
      _get_module_code(self, fullname)
...
try:
   ...
   # existing path
   if not isnative:
       exec(code, mod.__dict__)
   else:
       # code is a native C extension
       mod = _native_code(fullname, code)
...
return mod
```

The *_native_code* function instantiates a *ModuleSpec* for the C extension, and proxies the remaining work of loading the C extension to be performed by the builtin importer. The builtin import module was extended with an additional method `create_dynamic_inmemory`, which accepts a ModuleSpec instance for the extension to be loaded and a byte array containing the uncompressed bytes storing the extension. Functionally, the only difference between the new `create_dynamic_inmemory` and the `create_dynamic` in the CPython source is that `create_dynamic_inmemory` will ultimately call `LoadLibraryFromMemory` from the custom loader instead.

2) Python core shared library reference: The Python C extensions depend on the Python core shared library (see Figure 1 and Figure 2), which in our case is already loaded in the process's address-space and may not exist in the path. Therefore, the `LoadLibraryFromMemory` implementation in the interpreter that fulfills `create_dynamic_inmemory` returns the reference of this already loaded shared library when a C extension requests to load this Python core shared library from disk. We also store this reference as *sys.dllhandle*, so that other modules can use it if they need (See Section V-B).

B. Special case: ctypes

Python's ctypes is a Foreign Function Interface (FFI) library designed to support calling both functions in native

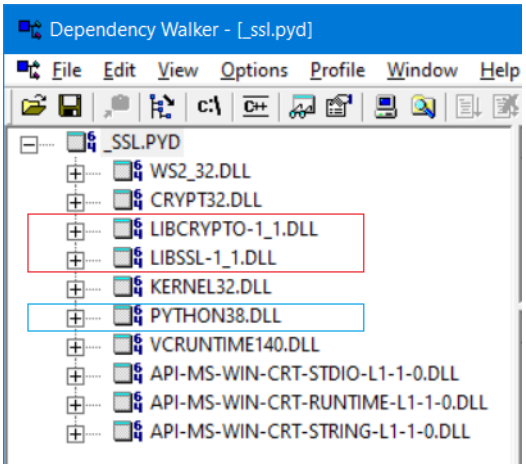

Fig. 1. C Extensions may depend on Python core shared library (python38.dll) and other non-system shared libraries. In this example, _ssl.pyd also depends on OpenSSL's libcrypto and libssl.

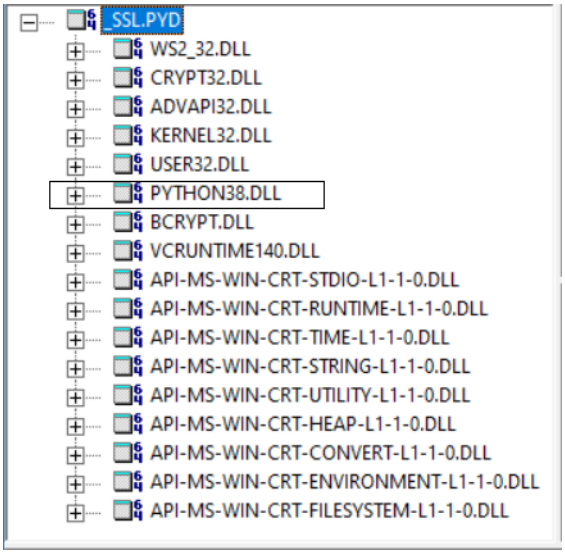

Fig. 2. The _ssl.pyd C Extension statically linked with OpenSSL: The libcrypto and libssl dependencies have been removed. Additional system shared libraries that libcrypto and libssl depended on appear under _ssl.pyd now.

shared libraries and also the underlying C functions in the Python interpreter's core shared library via *ctypes.pythonapi*. An example is presented below, where we invoke the Windows system shared library user32.dll's `MessageBoxA` to display a modal dialog box.

The _ctypes package depends on libffi [5] as a shared library on disk. To load _ctypes into the in-memory interpreter, we statically linked libffi into _ctypes. This removes the requirement for a libffi shared library to reside on disk. Because ctypes allows invoking C functions in the Python core shared library itself using *ctypes.pythonapi*, we must compare the base address reference of the interpreter against the value we stored in *sys.dllhandle*, and if equal, we use the in-memory

https://t.me/learningnets

loader to resolve the function lookups: This is because a call to `GetModuleHandle` will not return the base address of the in-memory CPython interpreter's since it was not loaded from disk by the system's `LoadLibrary` functionality.

Listing 3
INVOKING MESSAGEBOXA FROM CTYPES

```
import ctypes
MB_OK = 0
ctypes.windll.user32.MessageBoxA(None,
    b"Hello", b"World!", MB_OK)
```

C. Special case: Threading

To ensure Python code runs from threads created in C, first call `PyGILState_Ensure` to acquire the global interpreter lock (GIL) and store the thread state. Then run Python code. Once done, call `PyGILState_Release` to reset the thread state and release the GIL [19].

D. Other minor details

1) OpenSSL: The Windows OpenSSL implementation calls `GetModuleHandleEx`, which will not return the expected value since the library was not loaded by the Windows' Loader but our customer loader. Therefore, OpenSSL should be configured with `-DOPENSSL_USE_NODELETE`, which does not emit the `GetModuleHandleEx` code. Any C Extension calling `GetModuleHandleEx`, will need to be conditionally compiled to properly account for the case where the extension was not loaded by the Windows' Loader.

2) C Runtime Library: Similar to the official CPython interpreter, this in-memory CPython interpreter expects Microsoft C Runtime Library v14 (vcruntime140.dll) is installed on the system [14]: A fully-patched version of Windows 10 will already have this runtime DLL.

E. POSIX implementation

The POSIX implementation follows the same process; the only distinction is that the custom loader implements `dlopen_from_memory`. The usual route of in-memory code loading and execution involves allocation of memory pages marked executable via `mmap` and `mprotect`, however a shortcut is available in most cases. Because POSIX systems present the opportunity to represent many resources as a file, bytes from the shared object can be loaded into a pipe or memory-backed file descriptor and the loading and relocation process can be handled by `dlopen` from the C standard library (libc). Once loaded into an executable memory page, the `dlsym` provided by the operating system libc yields pointers to the desired functions from the shared object. A modified dynload_shlib.c routes all imports of native code through this mechanism for a complete in-memory experience.

VI. DEMONSTRATIONS

The in-memory embedding described in this paper is equivalent in functionality, in every way, to a stock CPython interpreter available for download from Python's official website. Of course, additional functionality can be loaded onto this in-memory embedding using modules or packages stored in ZIP archives, including native code implemented as Python C Extensions.

One approach threat-actors take is to use an exploit to deploy shellcode to perform actions in the context of an exploited process. Instead of using a 0-day or an *N*-day exploit, we simulated one: We developed an application that instantiates notepad.exe as a child process, writes shellcode into this instantiated notepad.exe's memory, and creates a remote thread to run this shellcode. That is, in this simulation, we are assuming notepad.exe was somehow exploited and a threat-actor successfully wrote shellcode to a location on memory and started a thread to run this shellcode. This shellcode downloads the in-memory Python interpreter and a ZIP archive into memory. The shellcode then loads and initializes the interpreter, adds a ZIP archive in memory with the target module, and calls the module in this archive.

The following demonstrations implement the topics we discuss in this paper, specifically, executing interpreted Python code directly from memory, calling native operating system functions, and loading Python C extensions in-memory and invoking functions in them. The demonstrations present a few well-known security research techniques implemented in Python. The artifacts will be made available as per Section VIII. We choose not to offer any new malware or reimplement existing malware into Python to run on top of this embedding: We are extremely wary about repurcussions of publishing malware along with this paper that security products may or may not defend against.

A. A simple demonstration covering all topics

In this demonstration, the module invokes the code in Listing 3 and produces a "Hello, world!" Message Box by calling an operating system C function. This demonstrates all the concepts discussed. Specifically, in this demonstration, we showed the CPython embedding executing interpreted Python code, loading a Python C Extension (ctypes), and then invoking an operating system C function `MessageBoxA` exported by the Windows shared library user32.dll.

B. Enumerate all processes on system

In this demonstration, we enumerate all processes by making a call to the partially documented Windows operating system function `ntdll!NtQuerySystemInformation`. This is a common technique malware use for reconnaissance: Enumerate processes and determine if any may indicate the process is emulated either in a virtual machine or if the system has a security product such as an antivirus is installed. This example also demonstrated all the concepts discussed: We loaded Python C Extension ctypes, and then invoked the operating system C function exported by Windows shared library ntdll.dll.

C. Download a file from the Internet

In this demonstration, using the urllib.request package, we download a file from a website. This also demonstrates all

https://t.me/learningnets

the concepts discussed. The urllib.request packages utilizes the socket and ssl packages. The socket and ssl packages import _socket.pyd and _ssl.pyd Python C Extensions, respectively. The extension _ssl.pyd statically links with OpenSSL. Sophisticated malware usually deploys in stages: The first stage performs reconnaissance, e.g., process enumeration. If the malware determines it is in the desired environment, it then reaches out to the Internet and downloads additional stages. In this demonstration we used urllib.request package to download a file from the Internet.

D. A capability to use the system BCrypt library

In this demonstration, we call cryptographic functions in the Windows operating system's BCrypt library from Python. This is something a late stage Malware may use, for example, to decrypt users' website credentials stored in Chrome's sqlite databases.

We offered a few samples demonstrating techniques security researchers employ. Given that this an in-memory embedding of a stock CPython, any code that would work in the stock CPython will also work in this embedding.

VII. Future Work

Some third-party Python packages not shipped with Python itself, such as PyCryptodome, require additional work to load into the in-memory interpreter. Specifically, files such as _raw_cbc.pyd, _raw_cfb.pyd, etc., are not proper Python C Extensions: These cannot initialize as C extensions as defined per `FAKE_INIT` [2]. PyCryptodome uses the *cffi* package to load these, and *cffi* expects the Python C extensions to reside on disk. The *cffi* package does not come standard with CPython. We may fork this *cffi* package, such that the requirement that these pseudo extension files need to be on disk is removed. The number of packages dependent on *cffi* can be identified in https://libraries.io/pypi/cffi/dependents. Essentially, any third-party library that offers a C Foreign Function Interface (CFFI) and wants to support in-memory dynamic loading will need to implement a functionality similar to [1]. We demonstrated our approach for CPython 3.8. This approach has also been validated to work in CPython 3.7 and CPython 3.9. We intend to support future versions of CPython, as the changes to apply are the same. We believe our in-memory CPython embedding has utility beyond offensive computer security research given it's simplicity in containing the entire CPython interpreter and default libraries in a single shared library.

VIII. Conclusion

Red Teams want to prototype malware from different threat-actors for their campaigns, which is difficult to accomplish rapidly when writing in C or assembly. Python is a popular interpreted high-level language, and would allow for such rapid prototyping. In this paper, we describe an in-memory embedding of CPython, which can be used for this type of prototyping, and can accomplish any specific task needing C or assembly via Python C extensions. We mentioned examples of malware written in Python and existing embeddings of the CPython interpreter that required the interpreter's files to reside on disk. We then detailed changes that allowed the CPython interpreter to run entirely in memory without "touching" the disk, and also loading Python packages and modules into it directly from memory. Finally, we covered handling special cases such as the *ctypes* package and threading. We believe our approach to embedding is simple and therefore, has utility beyond offensive computer security.

Availability

The artifacts submitted for evaluation are provided in Appendix A. The complete CPython source code with our modifications is available on GitHub under https://www.github.com/scythe-io/in-memory-cpython.

Appendix
Artifact Readme

This appendix describes the steps necessary to download a harness-exe that performs the demonstrations listed in the Demonstration section. Depending on the specific demonstration, the harness either creates a notepad.exe child process or uses the current console process. The harness-exe process allocates some Write+Execute memory, copies some shellcode and a harness DLL and starts a thread to execute the shellcode. The shellcode loads the harness DLL, and this downloads a CPython DLL constructed as described in the paper, and a demonstration chosen by the user in the steps of harness-exe. The specific steps involving the shellcode and the harness DLL emulate what would happen after an 0-day or N-day exploit without using an actual 0-day or N-day or compromising a system. Because this harness-exe is downloaded from the Internet, it will have a "Mark-of-the-Web," and Windows Defender will quarantine when you run it. Therefore, please follow the instructions in this document to Add an Exclusion for the harness-exe in Windows Defender.

A. Artifact Checklist

Binary: harness-exe.exe
Run-time environment: verified on Windows 10 x64
Hardware: A commodity PC is sufficient
Disk space required approximately: 100GB
Time needed to prepare workflow: 60 minutes
Time needed to complete experiments: 30 minutes
Publicly available: https://doi.org/10.5281/zenodo.4638251 and also under https://github.com/farfella/woot2021/
Code licenses: zlib

B. Description

The harness-exe has been verified to work on Microsoft's Windows 10 x64 Edge VM for VMWare. These instructions assume using Microsoft's Edge VM. Step 6 of the Installation section covers how to retrieve and execute harness-exe.

1) Download *VMWare Workstation Player* (approximately 215MB) and install on your computer (free for non-commercial use): https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html

https://t.me/learningnets

2) Download *MSEdge on Win 10 (x64) Stable 1809* (approximately 6.7GB) from: https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/. Choose VMWare for the VM platform.
3) Extract the downloaded MSEdge.Win10.VMware.zip and open the `MSEdge-Win10-VMware.ovf` inside the extracted folder MSEdge-Win10-VMware.
4) This will start VMWare Workstation Player importing the Edge Virtual Machine (VM).
5) Log into this Edge VM using:
   a) User: `IEUser`
   b) Password: `Passw0rd!`
6) In this VM, create a folder named `woot2021` on the C drive.
7) Start an *Administrative Command Prompt*.
8) From the Start menu, type `cmd`, and when Command Prompt appears, choose *Run as administrator* on the right as shown in the figure below and approve the consent dialog box.
9) Type *powershell* and hit Enter to start a Powershell prompt:
10) Add `C:\woot2021` to the Windows Defender Exclusions, turn off automatic sample submission, and download and install Visual Studio redistributable (approximately 1MB). The installation of the redistributable is a requirement for CPython itself. To accomplish these tasks, copy-and-paste the following commands into the Powershell prompt (Step 5) and hit Enter:
11) Install the redistributable as shown below.
12) Download the `harness-exe` zip also using Powershell and open the folder. Type the following and hit Enter.

```
Invoke-WebRequest
-Uri "https://github.com/farfella/
      woot2021/raw/master/
      harness/harness-exe.zip"
-OutFile "C:\woot2021\harness.zip"
explorer .
```

13) Extract the archive into the C:\woot2021\harness folder. The password to extract is: `w00t-2021-w00t`

C. Experiment workflow

In the same command prompt as Installation steps, change directory to c:\woot2021\harness. Then run .\harness-exe.exe and hit Enter.

```
CD C:\woot2021\harness
.\harness-exe.exe
```

When you run harness-exe.exe, some instructions are presented. After this, the reviewer may choose the default URL in which the in-memory CPython DLL resides (e.g., https://github.com/farfella/woot2021/tree/master/in-memory-embedding-cpython) or download from this GitHub repository and offer the DLL from the reviewer's own hosted server, for example.

Following this step, the reviewer is prompted to choose one of five demonstration options. Options one through four are the examples described in the Demonstrations section in the paper:

1) A demonstration covering all topics (spawns child notepad.exe and shows a dialog box)
2) A demonstration that prints all the process names running on the system
3) A demonstration that downloads a file from the Internet
4) A demonstration calling BCrypt to encrypt and decrypt a string

You will need to run harness-exe.exe for each demonstration.

D. Evaluation and expected results

For option 1, a child notepad.exe will be spawned, in which Write+Execute memory will be allocated, and shellcode and harness DLL will be written. Then a thread will be spawned in this child notepad.exe to execute this shellcode. The shellcode will load the harness DLL. The harness DLL will retrieve the CPython DLL constructed as described in the paper and also the zip file corresponding to demonstration 1 from either the default URLs or reviewer-provided URLs.

For option 2, 3, and 4, in the current process (i.e., process of harness-exe.exe) the write+execute memory will be allocated and the shellcode and harness DLL will be written. This is because these examples use print() to write to standard output, and the current process is a console application that can support this functionality. The CPython DLL is instantiated with verbosity turned on (i.e., equivalent to running python=v). As such, you will see debug messages on the screen to give you more insight.

E. Experiment Customization

The fifth demonstration option allows you to offer a zip file with a module named magic inside (i.e., magic.py or magic.pyc) from their own hosted URL. The harness DLL is constructed to only `import magic`. This allows you to test with some custom Python code instead of the four demonstration options. Note however, that only the standard Python packages are embedded in this CPython DLL. The demonstrations are available under: https://github.com/farfella/woot2021/.

REFERENCES

[1] Joachim Bauch. Loading a DLL from memory, 2010. https://www.joachim-bauch.de/tutorials/loading-a-dll-from-memory/.
[2] Helder Eijs. PyCryptodome. https://github.com/Legrandin/pycryptodome/blob/master/src/raw_cbc.c#L34.
[3] Stephen Fewer. Reflective DLL Injection. https://github.com/stephenfewer/ReflectiveDLLInjection.
[4] Hartmut Goebel. PyInstaller Quickstart - PyInstaller bundles Python applications, 2020. https://www.pyinstaller.org/.
[5] Anthony Green et al. libffi: A Portable Foreign Function Interface Library, 2019. https://sourceware.org/libffi/.
[6] Thomas Heller, Jimmy Retzlaff, and Mark Hammond. py2exe, 2019. https://www.py2exe.org/.
[7] Austin Jackson. Python Malware on the Rise, 2020. https://www.cyborgsecurity.com/python-malware-on-the-rise/.
[8] Dhiru Kholia and Przemyslaw Wegrzyn. Looking inside the (Drop) box. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.
[9] Lion Kimbro et al. Freeze, 2013. https://wiki.python.org/moin/Freeze.

https://t.me/learningnets

[10] Amit Klein and Itzik Kotler. Process Injection Techniques: Gotta Catch Them All. In *Black Hat USA*, 2019.
[11] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1.0 edition, 1999. https://linker.iecc.com/.
[12] Tal Liberman and Eugene Kogan. Lost in Transaction: Process Doppelgänging. In *Black Hat Europe*, 2017.
[13] Hongjiu Lu. Elf: From the programmer's perspective, 1995.
[14] Microsoft Corporation. C++ binary compatibility between Visual Studio 2015, 2017, and 2019. https://docs.microsoft.com/en-us/cpp/porting/binary-compat-2015-2017?view=vs-2019.
[15] Ross Osterlund. What Goes On Inside Windows 2000: Solving the Mysteries of the Loader, 2002. https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/march/windows-2000-loader-what-goes-on-inside-windows-2000-solving-the-mysteries-of-the-loader.
[16] Lauren Pearce. Covert Malware Launching and Data Encoding: Malware Analysis Day 5. Technical report, Los Alamos National Lab (LANL), Los Alamos, NM (United States), 2018. https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-18-25474.
[17] Python Software Foundation. Embedding Python in Another Application, 2020. https://docs.python.org/3/extending/embedding.html.
[18] Python Software Foundation. The import system, 2020. https://docs.python.org/3/reference/import.html.
[19] Python Software Foundation. Initialization, Finalization, and Threads, 2020. https://docs.python.org/3/c-api/init.html#non-python-created-threads.
[20] Python Software Foundation. zipimport - Import modules from Zip archives, 2020. https://docs.python.org/3/library/zipimport.html.
[21] Mark Russinovich and Thomas Garnier. Sysmon v11.11, 2020. https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#event-id-11-filecreate.
[22] Skape and Jarkko Turkulainen. Remote Library Injection, 2004. http://hick.org/code/skape/papers/remote-library-injection.pdf.
[23] Ryan Tracey. Meet PyXie: A Nefarious New Python RAT, 2019. https://blogs.blackberry.com/en/2019/12/meet-pyxie-a-nefarious-new-python-rat.
[24] Liam Tung. Programming language popularity: Python overtakes Java - as Rust reaches top 20, 2020. https://www.zdnet.com/article/programming-language-popularity-python-overtakes-java-as-rust-reaches-top-20/.
[25] Just van Rossum and Paul Moore. PEP 302 – New Import Hooks, 2002. https://www.python.org/dev/peps/pep-0302/.
[26] David A. Wheeler. Program Library HOWTO, 2003. https://tldp.org/HOWTO/Program-Library-HOWTO/.

https://t.me/learningnets