PENTERA | Pentera Labs™ Research Series

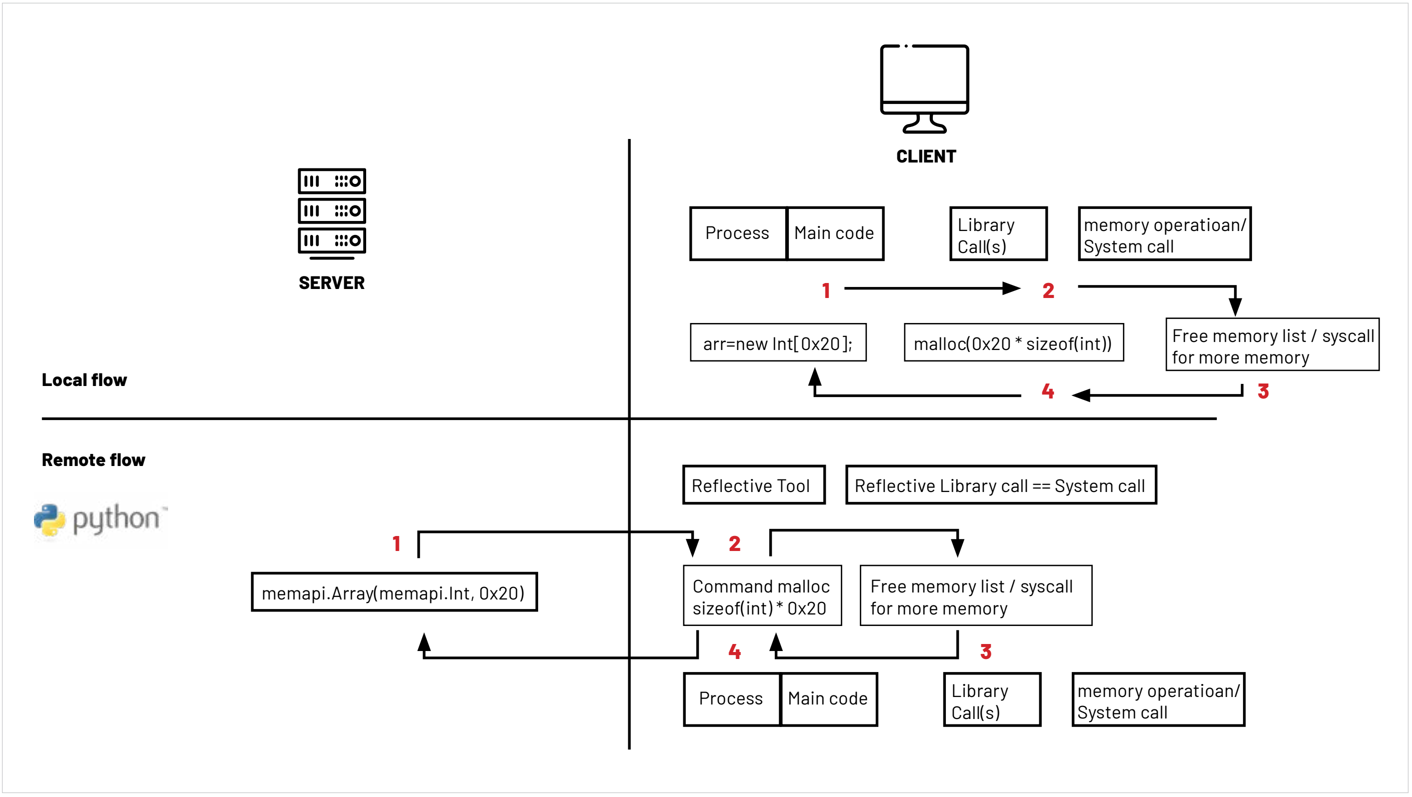
Reflective loading with Remote Memory Interactions: Controlling local operations remotely

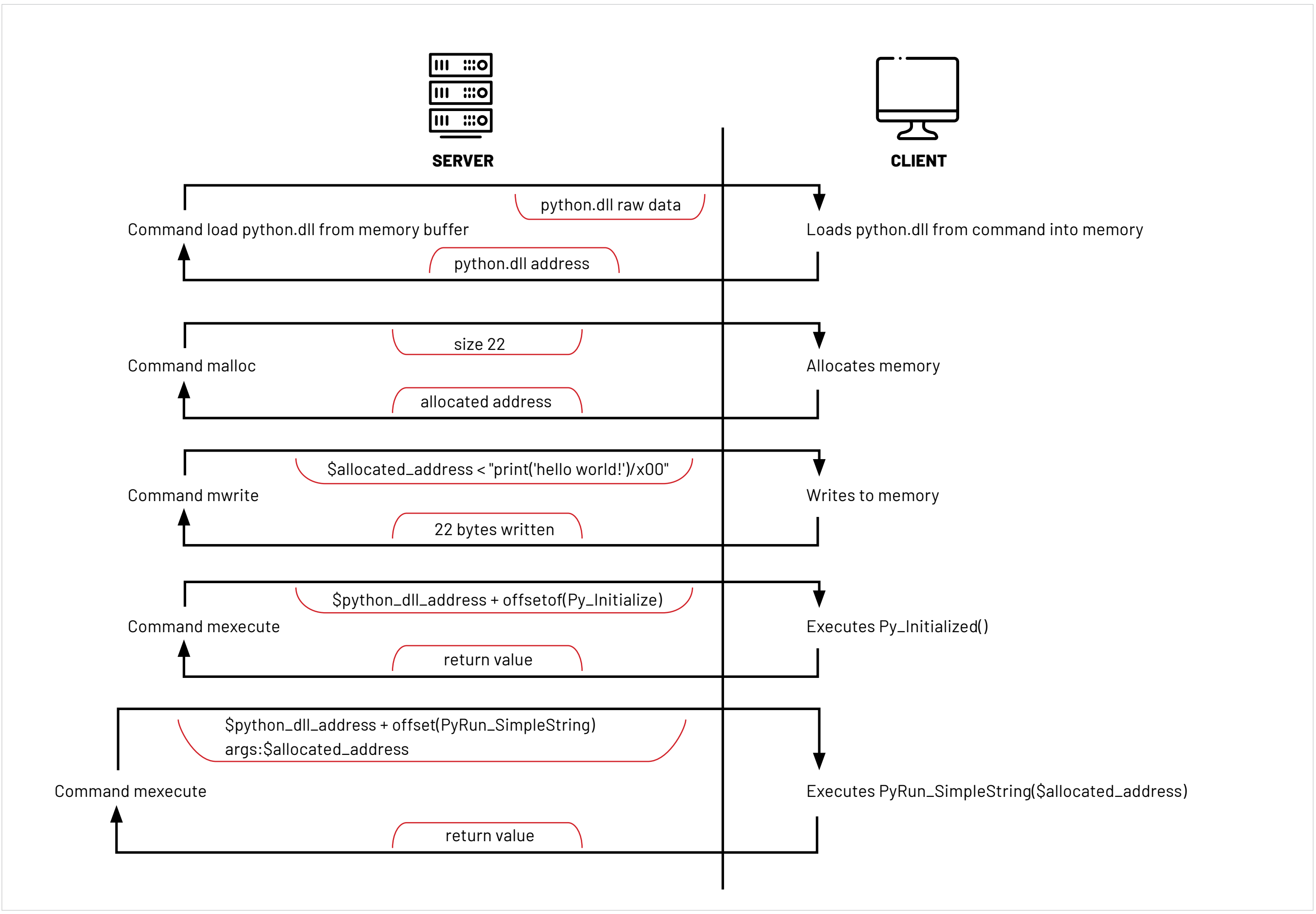
Itamar Brem

Table of contents

Executive summary

In a previous article we talked about implementing a tool for reflective loading
This article is about how to implement the framework that enables interactions with reflectively loaded modules. The framework can be used to reflectively load DLLs into a remote host's memory and interact with them. This will allow us to allocate objects in the remote host's memory and then call functions inside the DLL with those objects. By using a framework to manage interactions with the remote memory we can scale up our attack tools and use the local Reflective DLL loading technique remotely on multiple hosts at once.


CLIENT
SERVER
Process
Main code
Library Call(s)
memory operatioan/ System call
1
2
arr=new Int[0x20];
malloc(0x20 * sizeof(int))
Free memory list / syscall for more memory
4
3
Local flow
Remote flow
python
Reflective Tool
Reflective Library call == System call
1
2
memapi.Array(memapi.Int, 0x20)
Command malloc sizeof(int) * 0x20
Free memory list / syscall for more memory
4
3
Process
Main code
Library Call(s)
memory operatioan/ System call


SERVER
CLIENT
python.dll raw data
Command load python.dll from memory buffer
Loads python.dll from command into memory
python.dll address
size 22
Command malloc
Allocates memory
allocated address
$allocated_address < "print('hello world!')/x00"
Command mwrite
Writes to memory
22 bytes written
$python_dll_address + offsetof(Py_Initialize)
Command mexecute
Executes Py_Initialized()
return value
$python_dll_address + offset(PyRun_SimpleString)
args:$allocated_address
Command mexecute
Executes PyRun_SimpleString($allocated_address)
return value

Example use case: Calling PyRun_SimpleString()

Let's say we want to reflectively load python.dll, call Py_Initialize() and then call PyRun_SimpleString ("print('hello world')"). To do so, we would need a way to allocate the string in the remote host's memory and call these functions with the string as an argument(for PyRun_SimpleString()).

Essentially, we want some sort of interface, which resembles python's ctype library to access memory on the remote host.

The problem can be divided into three parts:

- How to handle memory operations
- How to handle data types
- What actions to perform

In other words, the three parts of the problem are memory, objects and functions.
The MemoryAPI is an API to access data, the BufferAPI is in charge of buffering local objects to remote memory and the ModuleAPIs are ways to represent functions on the server but have the native code execution performed on the remote client.

Memory API

The first thing we need is access to the remote host's memory. To do so, we need to implement some sort of MemoryAPI that contains all the required logic.

The MemoryAPI is merely a class that implements six functions:

```
allocate(size) -> address
free(address) -> optional? int (result_code)
read(address, size=1) -> bytes (data)
write(address, data) -> optional? int (result_code)
execute(address, *args) -> int (result_value)
*args must be ints up to size sizeof(void*) (size of registers)
protect(address, size, set_to, *attributes) -> int (result_code)
(not yet implemented, can be implemented by executing functions with
execute())
      if not attributes
            set_to represents the full collection of MemoryAttributes
(mask of OR'd MemoryAttribute values)
      otherwise
            bool(set_to) is True/False to turn the attributes on/off
            *attributes is list of MemoryAttribute values
```

These six functions can be implemented in a variety of ways. For instance, through a CnC-like tool that relays each command to the host and has the host serve the request. Or, by using "primitives" through some vulnerability. A write primitive could be a ropchain that either writes directly or calls something like socket_recv() with the desired memory address, for example.

We created a simple class that is initialized with a callback for each of the six functions. The class simply stores/holds all the functions for us in its members.

We implemented each of the six functions as a command in the ReflectiveTool. On the server side, we initialized the MemoryAPI with the relevant ReflectiveTool instance's functions.

As long as each of the six functions perform the desired effect and relay the results back to us when we call it, we're content.

Buffer API

We now have access to the remote host's memory. Technically, we have everything we need to allocate objects and execute functions remotely.

The issue is that the caller is now in charge of managing all memory operations. This includes allocating the required size, writing the data itself and then freeing the memory when it's no longer needed.

The idea is to abstract away things that the programmer shouldn't worry about, similar to memory management in modern programming languages like Python, as opposed to older programming languages like C where the programmer was in charge of memory management.

To solve this issue, we implemented BufferAPI classes. These classes are in charge of allocating, writing, reading and freeing anything we want to have on the remote host, AKA buffering data from our server to the remote host.

The BufferAPI class is initialized with a MemoryAPI and has a property called "data". When set, it allocates the required size and writes the data to the newly allocated memory. The address of the memory is stored in the "address" member.

When the pythonic reference to the BufferAPI is lost, the __free__() method will free the memory on the remote host as well. When the instance on our server is destroyed, the instance on the remote host will be destroyed as well.

The important functions in the BufferAPI classes are get_data(), set_data(), fetch() and flush(). [get/set]_data() are in charge of normalizing the member "data" so that we're always handling bytes. fetch()/flush() are in charge of streaming the bytes from/to the remote host.

The BufferAPI class knows how to handle bytes and each subclass of BufferAPI handles a different type. For example, the WideStringAPI and StringAPI built upon (NULLTerminatedAPI built upon) BufferAPI handles strings and QwordAPI handles 64 bit integers.

The execute() function should natively know how to receive integers, but the QwordAPI can be used to "pass-by-reference", for example as an output to GetFileSizeEx. Then we can cast the QwordAPI as an int (example below).

Module API

After loading the DLL on the remote host we want to call its functions. To do so, we need to understand where they are located in the remote host's memory.

To find the location of functions inside a DLL we need:

1. The base address where the DLL was loaded
2. The offset of the function we want from the base address

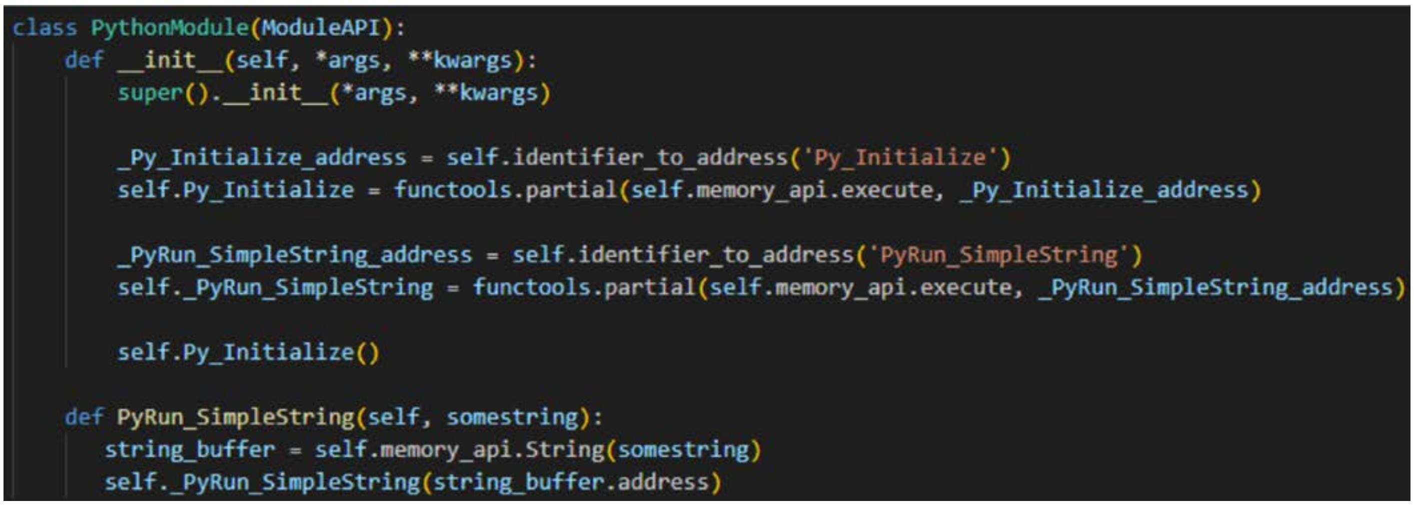
The command command_preload_DLL() in the ReflectiveTool returns the address of where it loaded the DLL, so we can satisfy the first requirement.
Debug symbols usually store function locations as offsets, since the base address may change each time the DLL is loaded. So all we need for the second requirement is the debug or export symbols from the DLL we loaded.

The ModuleAPI is initialized with the base address and a DebugHandler object to look up function locations from debug symbols.

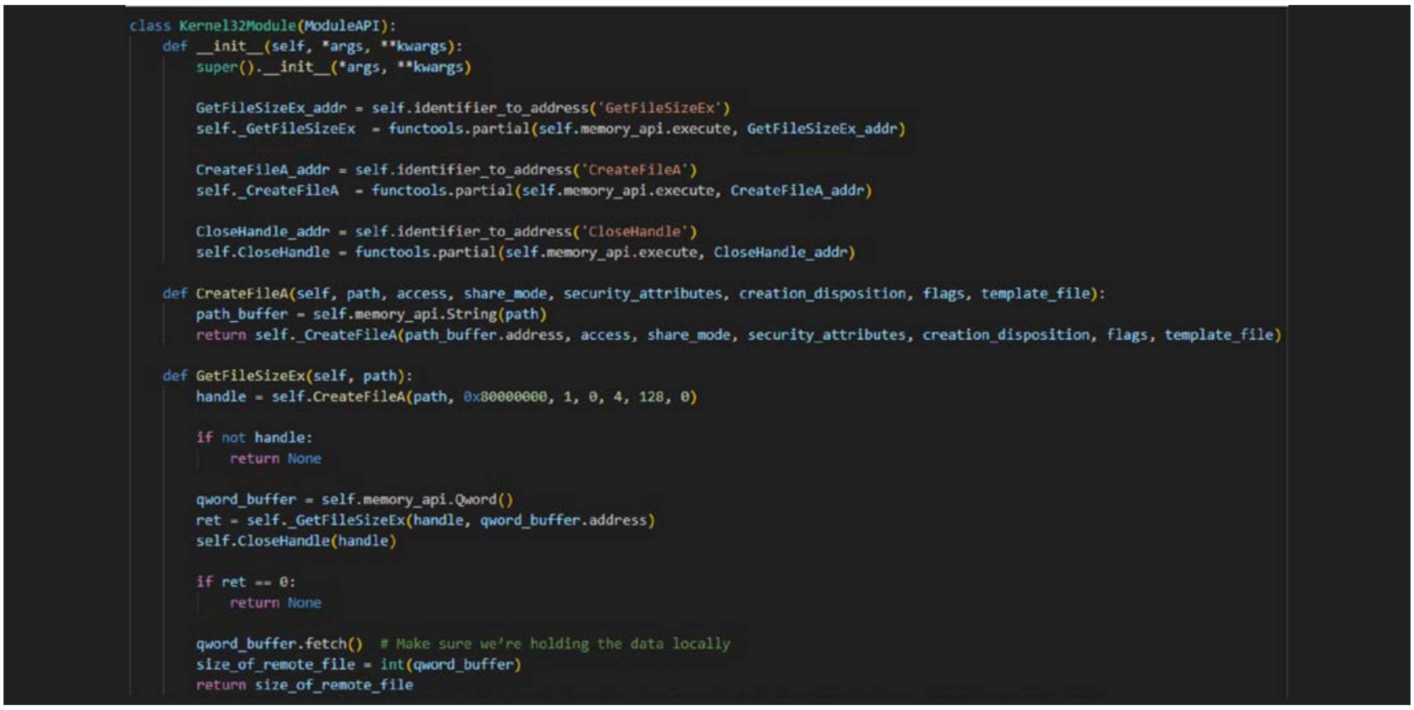
The only interesting function that the base ModuleAPI class implements is identifier_to_offset(). The function receives a string representing the name of the function we want to call and returns its offset.

The ModuleAPI class can be used to implement logic using the MemoryAPI and BufferAPI classes.

For example, the PythonModule class might look like this:



```
class PythonModule(ModuleAPI):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        _Py_Initialize_address = self.identifier_to_address('Py_Initialize')
        self.Py_Initialize = functools.partial(self.memory_api.execute, _Py_Initialize_address)

        _PyRun_SimpleString_address = self.identifier_to_address('PyRun_SimpleString')
        self._PyRun_SimpleString = functools.partial(self.memory_api.execute, _PyRun_SimpleString_address)

        self.Py_Initialize()

    def PyRun_SimpleString(self, somestring):
       string_buffer = self.memory_api.String(somestring)
       self._PyRun_SimpleString(string_buffer.address)
```



```python
class Kernel32Module(ModuleAPI):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        GetFileSizeEx_addr = self.identifier_to_address('GetFileSizeEx')
        self._GetFileSizeEx  = functools.partial(self.memory_api.execute, GetFileSizeEx_addr)

        CreateFileA_addr = self.identifier_to_address('CreateFileA')
        self._CreateFileA  = functools.partial(self.memory_api.execute, CreateFileA_addr)

        CloseHandle_addr = self.identifier_to_address('CloseHandle')
        self.CloseHandle = functools.partial(self.memory_api.execute, CloseHandle_addr)

    def CreateFileA(self, path, access, share_mode, security_attributes, creation_disposition, flags, template_file):
        path_buffer = self.memory_api.String(path)
        return self._CreateFileA(path_buffer.address, access, share_mode, security_attributes, creation_disposition, flags, template_file)

    def GetFileSizeEx(self, path):
        handle = self.CreateFileA(path, 0x80000000, 1, 0, 4, 128, 0)

        if not handle:
            return None

        qword_buffer = self.memory_api.Qword()
        ret = self._GetFileSizeEx(handle, qword_buffer.address)
        self.CloseHandle(handle)

        if ret == 0:
            return None

        qword_buffer.fetch()  # Make sure we're holding the data locally
        size_of_remote_file = int(qword_buffer)
        return size_of_remote_file
```

Transforming arguments

Notice some of the code above looks like it can be generalized since we see almost duplicate things repeat. For instance, x = self.identifier_to_address(x), which is always sent as the first argument to self.memory_api.execute().

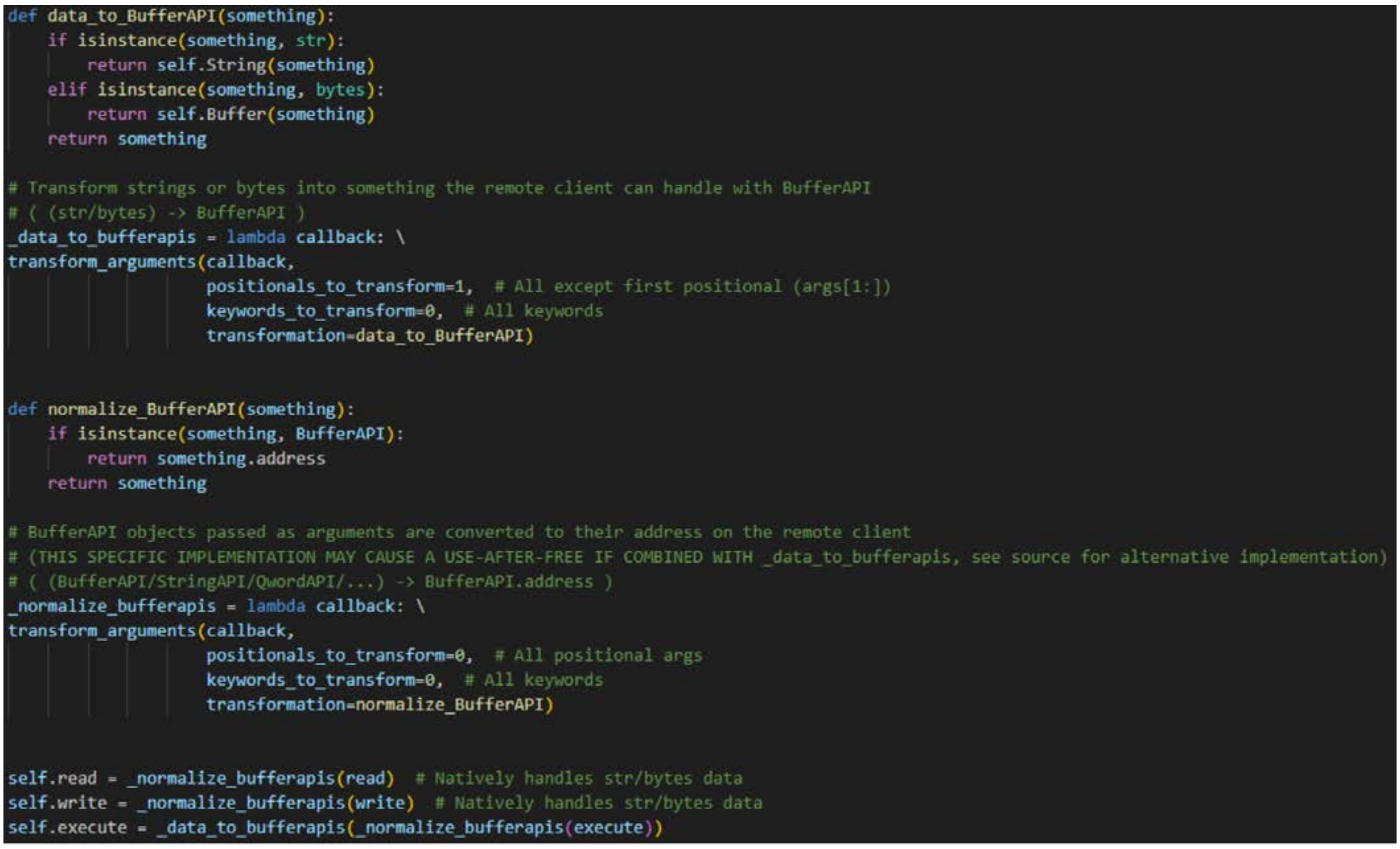
Another generalization we can do is to automatically convert BufferAPI objects to their address in the remote host's memory. The pythonic object on our server has no meaning to the remote host, so we can't use it as an argument to function calls.

We wish to simply call:
`self._PyRun_SimpleString(string_buffer)`
or even better yet:
`self._PyRun_SimpleString(somestring)`

Then, to have the internals of the framework transform somestring into a StringAPI object, which in turn will be transformed to an address.

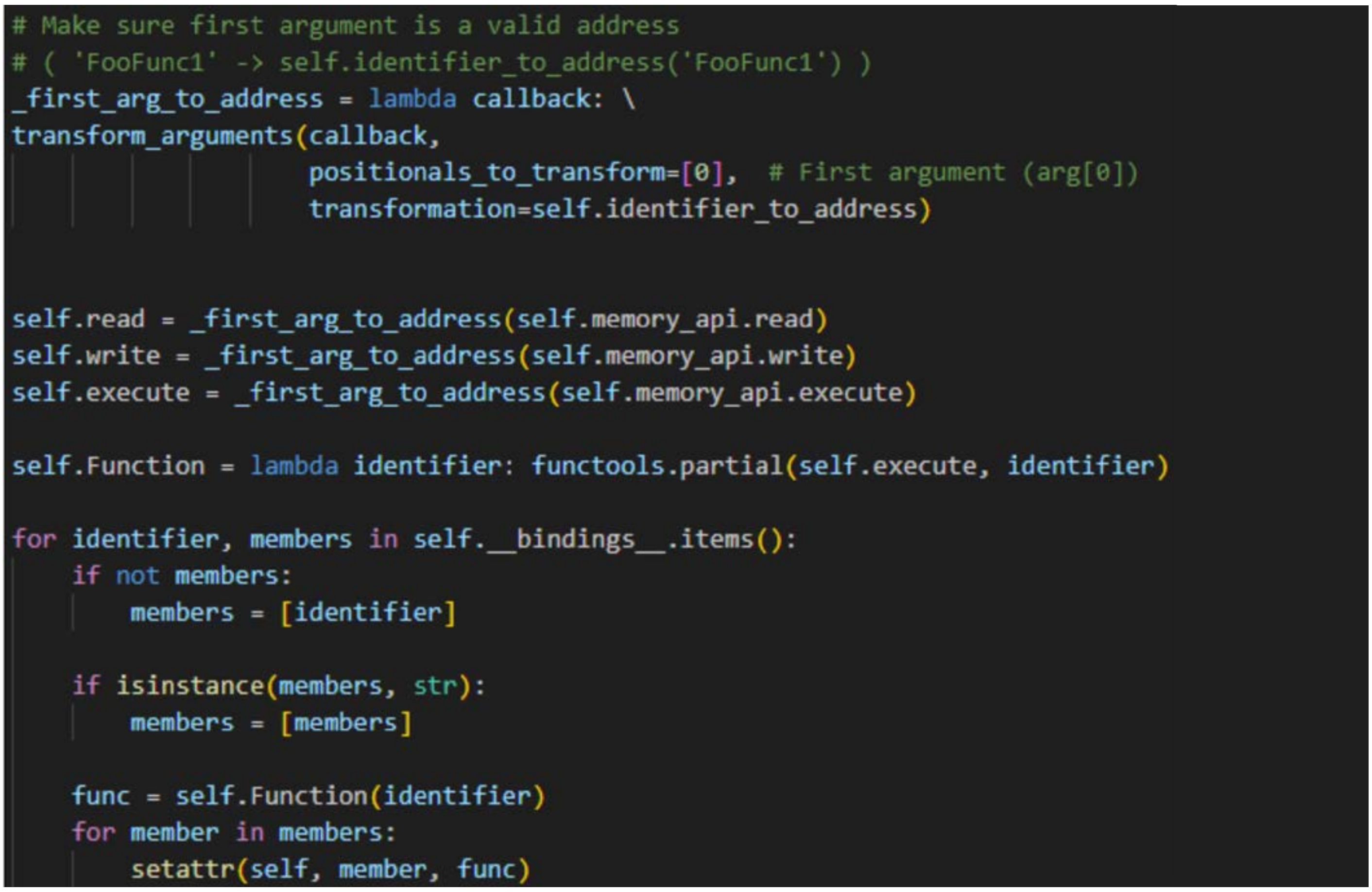
To automate this, we implemented a decorating function called transform_arguments(). It receives a callback and transformation function as well as specification of which arguments to transform. The function transform_arguments() returns the callback wrapped in another function, which simply applies the transformation to the specified arguments and calls the original callback with the transformed arguments.

Here are parts of the __init__() method of the MemoryAPI class.



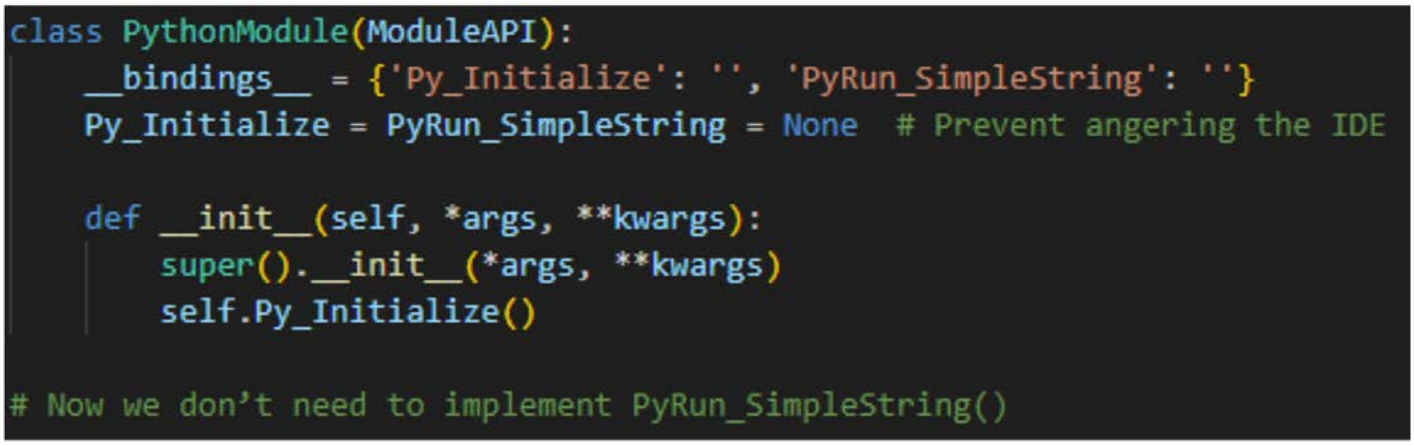
```python
def data_to_BufferAPI(something):
    if isinstance(something, str):
        return self.String(something)
    elif isinstance(something, bytes):
        return self.Buffer(something)
    return something

# Transform strings or bytes into something the remote client can handle with BufferAPI
# ( (str/bytes) -> BufferAPI )
_data_to_bufferapis = lambda callback: \
transform_arguments(callback,
                    positionals_to_transform=1,  # All except first positional (args[1:])
                    keywords_to_transform=0,  # All keywords
                    transformation=data_to_BufferAPI)


def normalize_BufferAPI(something):
    if isinstance(something, BufferAPI):
        return something.address
    return something

# BufferAPI objects passed as arguments are converted to their address on the remote client
# (THIS SPECIFIC IMPLEMENTATION MAY CAUSE A USE-AFTER-FREE IF COMBINED WITH _data_to_bufferapis, see source for alternative implementation)
# ( (BufferAPI/StringAPI/QwordAPI/...) -> BufferAPI.address )
_normalize_bufferapis = lambda callback: \
transform_arguments(callback,
                    positionals_to_transform=0,  # All positional args
                    keywords_to_transform=0,  # All keywords
                    transformation=normalize_BufferAPI)


self.read = _normalize_bufferapis(read)  # Natively handles str/bytes data
self.write = _normalize_bufferapis(write)  # Natively handles str/bytes data
self.execute = _data_to_bufferapis(_normalize_bufferapis(execute))
```

And here are parts of the __init__() method of the ModuleAPI class:



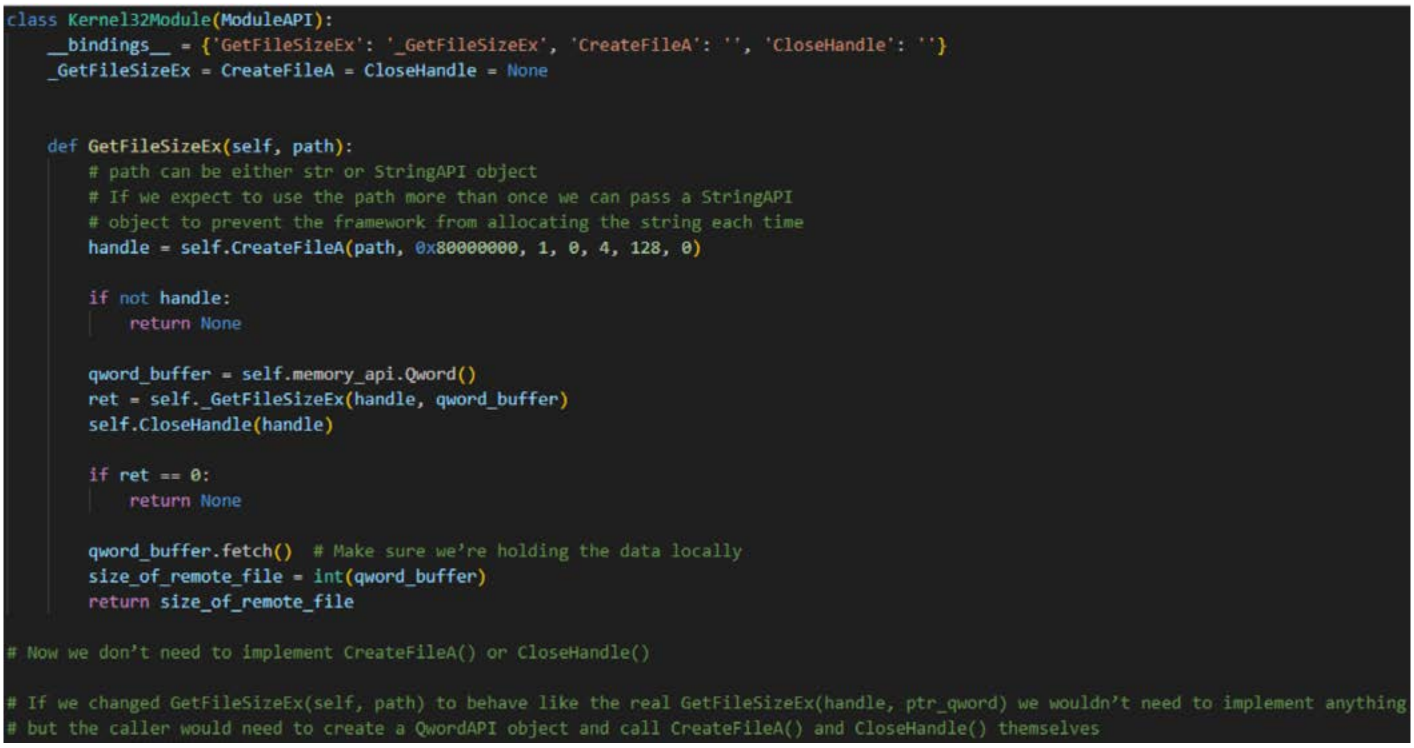
```python
# Make sure first argument is a valid address
# ( 'FooFunc1' -> self.identifier_to_address('FooFunc1') )
_first_arg_to_address = lambda callback: \
transform_arguments(callback,
                    positionals_to_transform=[0],  # First argument (arg[0])
                    transformation=self.identifier_to_address)


self.read = _first_arg_to_address(self.memory_api.read)
self.write = _first_arg_to_address(self.memory_api.write)
self.execute = _first_arg_to_address(self.memory_api.execute)

self.Function = lambda identifier: functools.partial(self.execute, identifier)

for identifier, members in self.__bindings__.items():
    if not members:
        members = [identifier]

    if isinstance(members, str):
        members = [members]

    func = self.Function(identifier)
    for member in members:
        setattr(self, member, func)
```

The PythonModule can now be written like:



```
class PythonModule(ModuleAPI):
    __bindings__ = {'Py_Initialize': '', 'PyRun_SimpleString': ''}
    Py_Initialize = PyRun_SimpleString = None  # Prevent angering the IDE

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.Py_Initialize()

# Now we don't need to implement PyRun_SimpleString()
```

Kernel32Module can be rewritten as:



```
class Kernel32Module(ModuleAPI):
    __bindings__ = {'GetFileSizeEx': '_GetFileSizeEx', 'CreateFileA': '', 'CloseHandle': ''}
    _GetFileSizeEx = CreateFileA = CloseHandle = None


    def GetFileSizeEx(self, path):
        # path can be either str or StringAPI object
        # If we expect to use the path more than once we can pass a StringAPI
        # object to prevent the framework from allocating the string each time
        handle = self.CreateFileA(path, 0x80000000, 1, 0, 4, 128, 0)

        if not handle:
            return None

        qword_buffer = self.memory_api.Qword()
        ret = self._GetFileSizeEx(handle, qword_buffer)
        self.CloseHandle(handle)

        if ret == 0:
            return None

        qword_buffer.fetch()  # Make sure we're holding the data locally
        size_of_remote_file = int(qword_buffer)
        return size_of_remote_file

# Now we don't need to implement CreateFileA() or CloseHandle()

# If we changed GetFileSizeEx(self, path) to behave like the real GetFileSizeEx(handle, ptr_qword) we wouldn't need to implement anything
# but the caller would need to create a QwordAPI object and call CreateFileA() and CloseHandle() themselves
```

The only issue with implementing the code with transformations is that the transformation function gets called each time the function gets called. If looking up symbols takes time it will slow us down.

So we implemented a decorating function (called to create a decorator) with the name `cache_by_arguments()`. It wraps a function with a caching function to prevent duplicate calls with the same arguments, returning cached values.

Conclusion

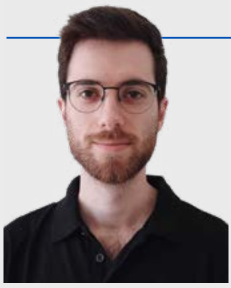
We now have a ctype-like framework to connect us to the memory of a remote host. It only required us to implement six basic functions. Going back to the problem we outlined at the start, we used MemoryAPI to answer "how to handle memory operations", BufferAPI to answer "how to handle data types" and ModuleAPI for "which actions to perform".

Now, we can load Python as a DLL on the remote host and execute Python scripts through the PythonModule.

Going further, we could also generalize the MemoryAPI functions to perform additional actions, such as create/delete/read/write/execute/protect files on a remote filesystem. To do so, you would need a MemoryAPI if you wanted to pass file names (or an object representing arguments) to the remote host. Otherwise you could use magic numbers to trigger different predefined events. With this capability, we could reuse the framework to manage executable files and have our management framework delete files after usage.

In our next and final article of the series, we'll discuss the API scripting language we developed that integrates these ideas to enable us to send complex requests to the client.

About the author



Itamar Brem is a Senior Security Researcher and exploit developer at Pentera.
Prior to joining Pentera, Itamar served in a classified unit in the IDF specializing in malware analysis, memory forensics, reverse engineering and exploit development and automation.

For any questions, feel free to reach out at labs@pentera.io

About Pentera

PENTERA

Pentera is the category leader for Automated Security Validation, allowing every organization to test with ease the integrity of all cybersecurity layers, unfolding true, current security exposures at any moment, at any scale. Thousands of security professionals and service providers around the world use Pentera to guide remediation and close security gaps before they are exploited.

For more info, visit: pentera.io

PENTERA | Pentera Labs™ Research Series

Reflective loading with Remote Memory Interactions: Controlling local operations remotely

www.Pentera.io