Improving the stealthiness of memory injection techniques

30th of May 2023

x33fcon

whoami

Diego Capriotti @naksyn

- Offensive Cyber Security @ Axians Italy
- ITA Army veteran Engineer Officer
- Past roles: RF jamming , Cyber Security
- Side interests: chess, tinkering with Software Defined Radios

Agenda

- Injection techniques
  - Main categories
- Improvement strategy
  - Setting the constraints
  - Testing with Memory Scanners
  - Reducing IoCs
- Module Shifting
  - Improvements
  - Bypassing memory scanners
  - Detection Opportunities

1

INTRODUCTION

Memory Injection Definition

Technique where an attacker inserts or alters code in a running local or remote process's memory space. The code can be executed within the context of the target process.

Memory Injection - Purposes

- Bypassing security measures
- Achieving stealth and persistence
- Extending agent capabilities
- Getting access to target process' sensitive information
- Debugging or RE

Memory Injection - Main categories

- **Code injection**
  - insert and execute malicious code within a target process's memory, typically involving dynamic memory allocation
- **PE injection**
  - injecting Portable Executable (PE) files, such as DLLs and EXEs, into the address space of a running process.
- **Process Manipulation**
  - manipulating or modifying the memory and execution context of running processes, libraries, or creating new processes with malicious payloads.

Code injection – Common techniques

Classic Shellcode Injection

- Allocate memory in the target process
- Write malicious code into the allocated memory
- Create a remote thread or execute via callback functions

Hook Injection

- Intercept API calls made by the target process
- Redirect the intercepted API calls to the malicious code

Thread Local Storage injection

- Modify the target process's PE header (TLS callback function)
- Execute the injected code as a TLS callback

APC injection

- Allocate memory in target process
- Write malicious code into it
- Queue APC
- Resume thread execution

Exception-Dispatching injection

- Allocate memory in the target process
- write malicious code into it
- Modify the target process's exception handler
- Trigger an exception in the target process

Code injection – IoCs

- **Dynamic memory allocation – prevalent IoC**
- **Changes in private memory permissions – prevalent IoC**
- **Technique-specific IoCs:**
  - Modifications to function pointers or function prologues
  - Modifications to target process's PE header
  - Modification of exception handler in target process
  - Queuing of APCs for target process

PE injection – Common techniques

Classic dll injection

- Drop dll on disk
- Allocate memory to target process and write malicious dll
- Load dll using Loadlibrary or similar method

Reflective dll injection

- Reflective loader is part of the malicious dll
- Load and map the malicious DLL into target process without the Windows loader
- Resolve dependencies and perform relocations

MemoryModule

- similar to Reflective dll injection but the loader code is external and not embedded in the dll
- More flexible, it allows the loading of unmodified dlls

Module Stomping

- Load a dll into the target process
- Overwrite sections with shellcode and execute it

Module Overloading

- Load a dll into the target process
- Overwrite loaded dll memory space with malicious PE

PE injection – IoCs

- PE (DLL or EXE) or shellcode is residing in memory – prevalent IoC
- Changes in memory permissions – prevalent IoC
- Mismatch between in-memory and on-disk dll code – prevalent IoC
- Technique-specific IoCs:
  - Loaded PE not backed by image on disk

Process Manipulation – Common techniques

Process Hollowing

- Create process in suspended state
- Replace memory contents with malicious exe
- Resume execution

Process doppelgänging

- Abuse NTFS transactions to load a malicious exe within the context of a legitimate process

Sideloading

- Drop dll on disk
- Abuse windows dll search order or missing dlls to load A malicious dll into a legitimate process

Thread execution hijacking

- Suspend a thread in the target process
- Modify instruction pointer to execute malicious code

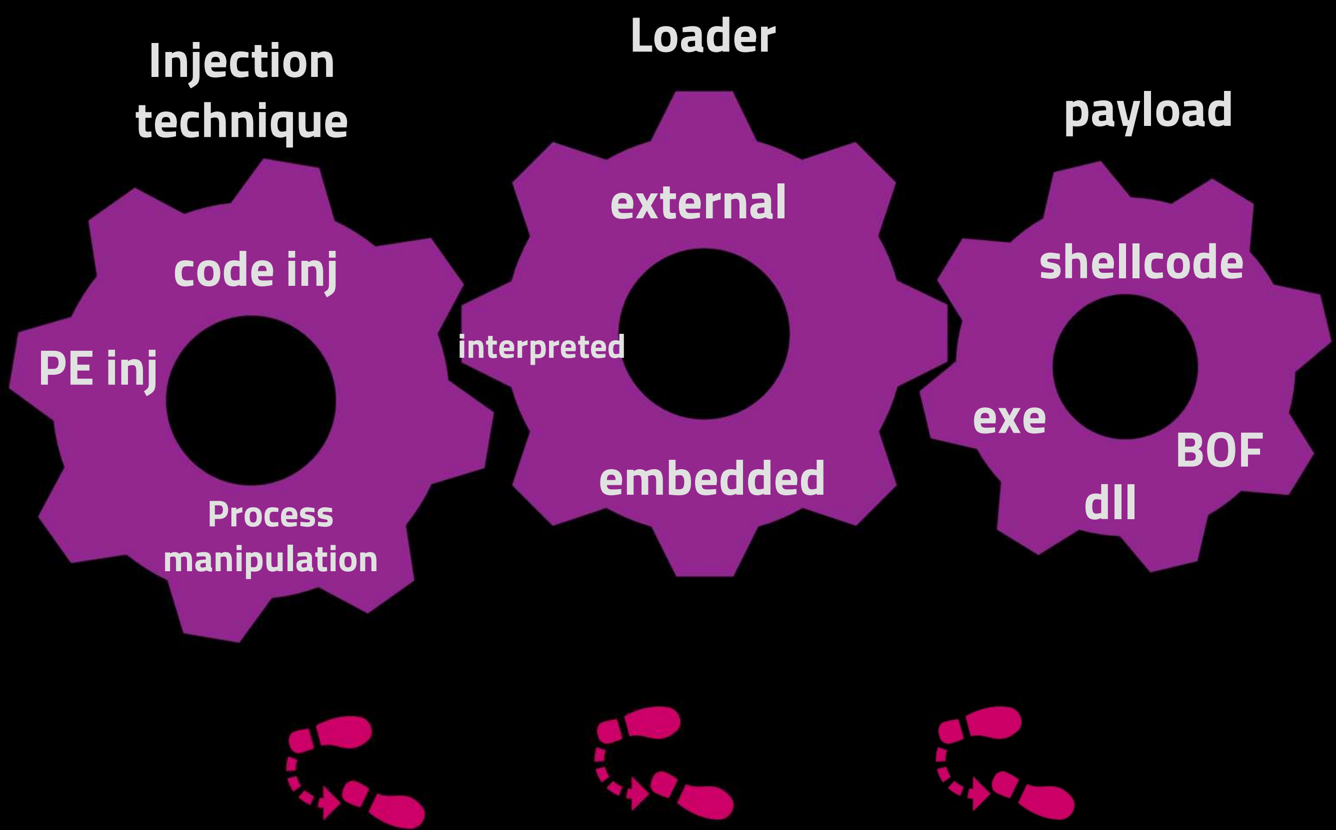
Process Manipulation – IoCs

- Altering the context or normal execution flow of PE – prevalent IoC
- Technique-specific IoCs:
  - Abusing NTFS transactions
  - Malicious dll dropped on disk

2

IMPROVEMENT STRATEGY

Memory Injection - Moving Parts



IoCs

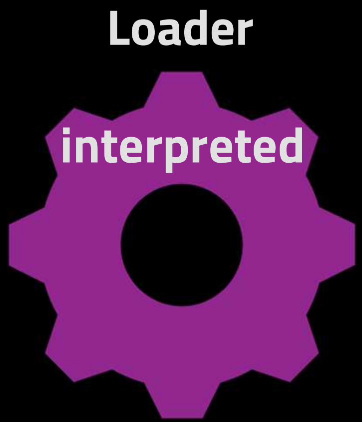
Setting the constraints - Injection

Injection technique

PE injection

- Blend with the environment without context alteration
- Avoid allocation of dynamic memory allocation during injection
- Module Overloading and Module stomping can be challenging to detect, we'll aim to improve them

Setting the constraints



- Python ctypes allows calling Windows APIs
- Python code and modules can be executed dynamically
- By executing dynamic Python ctypes code we can avoid

  using compiled loaders

Setting the constraints - Payload

- PE payloads need to reside in memory
- some features in the payload can enable improvements in the injection technique and reduce IoCs
  - Functional independence from further stages
  - Sleep obfuscation
  - Custom reflective loading
- Shellcode can be better suited for these features
  - Kyle Avery’s Aceldr

https://github.com/kyleavery/AceLdr

Testing with Memory scanners

Aleksandra Doniec (@hasherazade) PESieve:

- runtime usermode memory scanner designed to identify suspicious memory regions based on malware IOCs
- uses a variety of data analysis tricks to refine its detection criteria

Forest Orr's Moneta focuses on:

- presence of dynamic/unknown code
- suspicious characteristics of the mapped PE image regions
- IOCs related to the process itself (e.g. mapped image without entry in the PEB

False Positives

.NET dlls and CLR

```
| .NET DLL Image      | C:\Windows\assembly\NativeImages_v4.0.30319_64\mscorlib\b647019e9c1f6030fc3caa74d4f8f2ac\mscorlib.ni.dll
| RX        | .text    | 0x0000b000 | Modified code
```

45 kB

Normal behaviour
.text section overwritten

Discord ant third-party apps

```
| DLL Image            | C:\Users\diego.cappiotti\AppData\Local\Discord\app-1.0.9013\modules\discord_voice-1\discord_voice\discord_voice.node | Mismatching PEB module
| R          | Header  | 0x00001000 | Modified PE header
| RX         | .text   | 0x00203000 | Modified code
```

2.1 MB

False Positives – Python + .NET

```
| .NET DLL Image      | C:\Users\naksyn\Desktop\python-3.10.9-embed-amd64\pythonnet\runtime\Python.Runtime.dll | Unsigned module | Missing PEB module
| R         | Header    | 0x00001000 | Modified PE header
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Heap | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Heap | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
| Private
| RWX       | 0x00000000 | Abnormal private executable memory
```

Normal behaviour

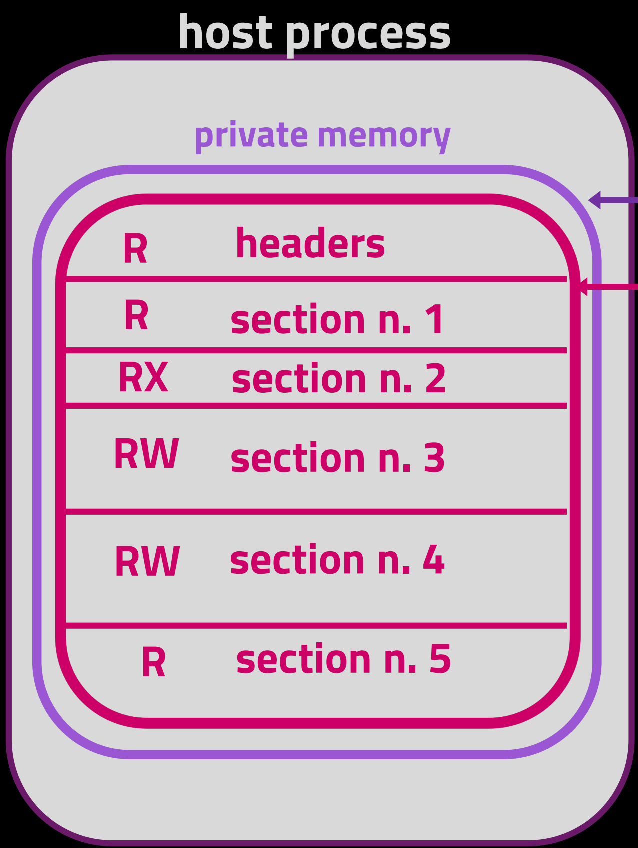
Private Memory with RWX permissions

Starting Point - PythonMemoryModule

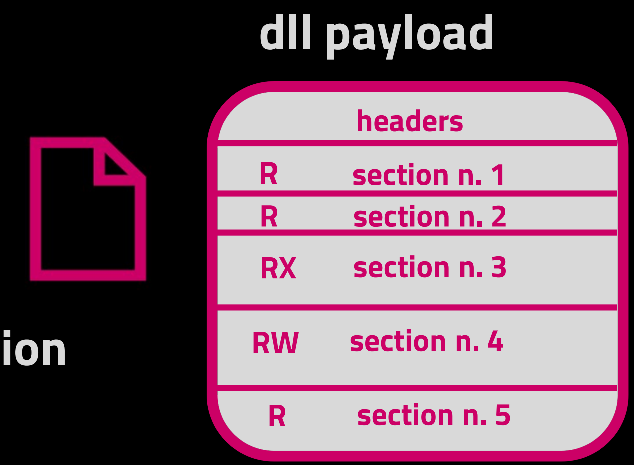
- Python ctypes porting of Joachin Bauch @fancycode MemoryModule technique
    - can be imported and executed dynamically in-memory with Pyramid
- No need to use a compiled loader if a Python interpreter is available (or droppable)
- Can download, map and execute a stageless dll agent like Cobalt Strike or Sliver
- Can also execute BOFs via injected COFFLoader dll

https://github.com/naksyn/PythonMemoryModule
https://tishina.in/execution/python-inmemory-bof

Starting Point - PythonMemoryModule



1. Dynamic Memory allocation
2. Download and copy of malicious dll section by section
3. Base relocation
4. Import resolution
5. Setting permissions
6. Entrypoint execution



credits to Joachim Bauch @fancycode for the technique
https://github.com/fancycode/MemoryModule

This is the eleventh maintenance release of Python 3.10

Python 3.10.10 is the newest major release of the Python programming language, and it contains many new features and optimizations.

Major new features of the 3.10 series, compared to 3.9

Among the new major new features and changes so far:

- PEP 623 -- Deprecate and prepare for the removal of the wstr member in PyUnicodeObject.
- PEP 604 -- Allow writing union types as X | Y

PythonMemoryModule – Pros and cons

- **Pro:**
  - **Avoids the creation of RWX memory**
- **Cons:**
  - **Dynamic Memory allocation**
  - **Executable section resident in memory**

```
python.exe : 8104 : x64 : C:\Users\naksyn\Desktop\python-3.10.11-embed-amd64\python.exe
  0x000000006BAC0000:0x00055000   | Private
    0x000000006BAC1000:0x00002000 | RX         | 0x00000000 | Abnormal private executable memory | Thread within non-image memory region
      Thread 0x000000006BAC16C0 [TID 0x00001554]
  0x000001C575A90000:0x00056000   | Private
    0x000001C575A90000:0x0002f000 | RX         | 0x00000000 | Abnormal private executable memory
```

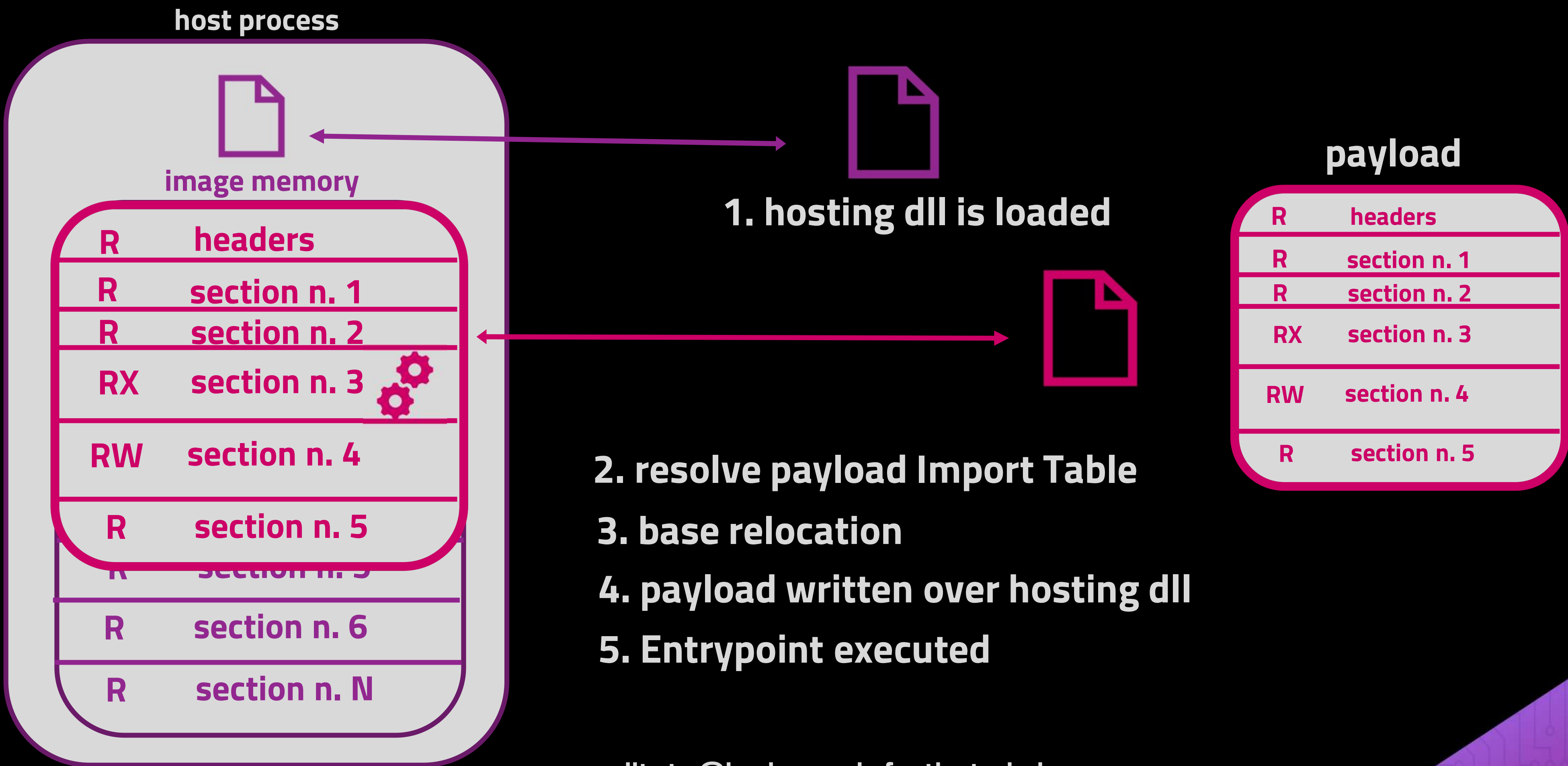
Target for improvement

Next step - Module Overloading

- Injection that can avoid allocation of Dynamic Memory by overwriting a PE over a legitimate loaded dll
- Can map the PE in memory using the same MemoryModule loading operations
- Can be implemented in Python ctypes building off of PythonMemoryModule

Next step - Module Overloading

Main Technique Improvement - Dynamic memory allocation is avoided



credits to @hasherezade for the technique
https://github.com/hasherezade/module_overloading

Module Overloading - IoCs

Moneta output

```
module_overloader64.exe : 8176 : x64 : C:\Users\naksyn\Desktop\module_overloader64.exe
  0x00007FF612B40000:0x00053000  | EXE Image             | C:\Users\naksyn\Desktop\module_overloader64.exe | Unsigned module
  0x00007FFCFEF70000:0x00042000  | DLL Image             | C:\Windows\System32\tapi32.dll
    0x00007FFCFEF70000:0x00001000 | R          | Header  | 0x00001000 | Modified PE header
    0x00007FFCFEF71000:0x00010000 | RX         | .text   | 0x00010000 | Modified code
```

Target for improvement

IoC

PESieve output

```
Total scanned:       55
Skipped:             0
-
Hooked:              0
Replaced:            1
Hdrs Modified:       0
IAT Hooks:           0
Implanted:           0
Unreachable files:   0
Other:               0
-
Total suspicious:    1
---
```

IoC

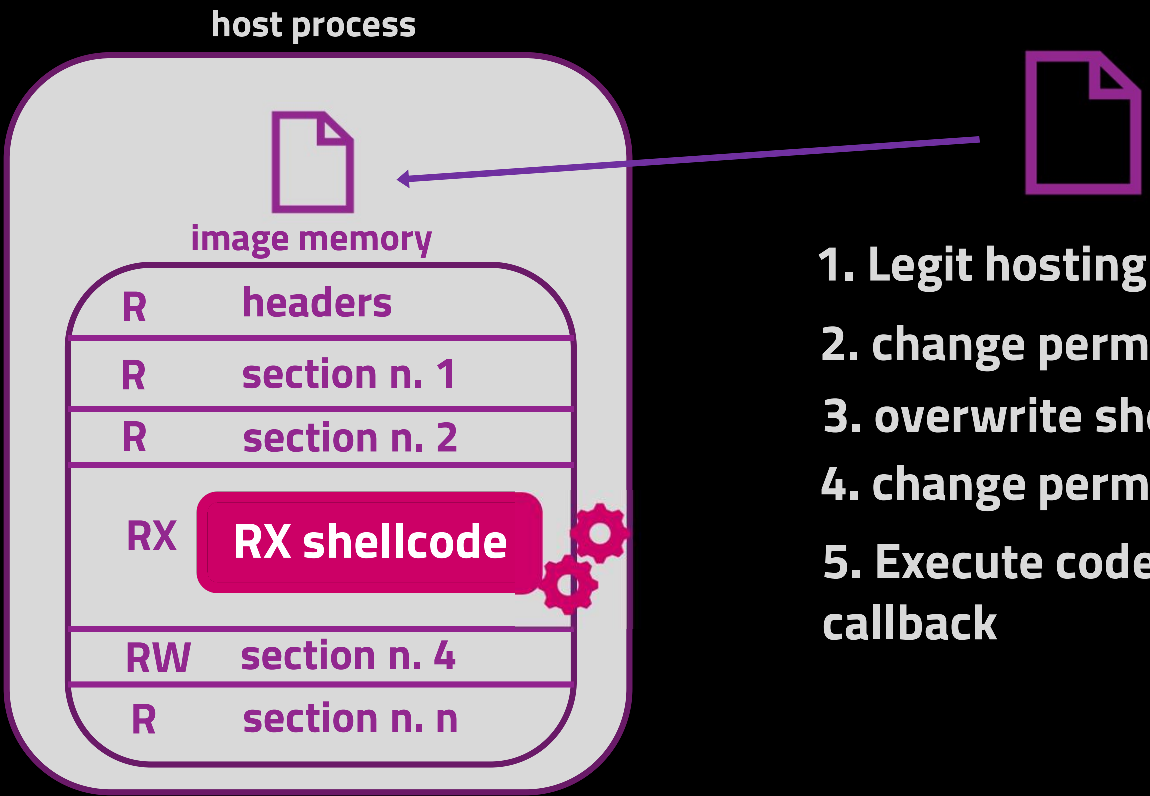
payload overwritten over hosting dll starting from PE header

Next step - Module Stomping

- Very similar to Module Overloading
- Uses shellcode payload instead of a PE
- Much simpler to implement – avoids the PE loading operations
- With the right payload we can get rid of the resident executable section IoC

Next step - Module Stomping

Main Technique Improvement – smaller and more versatile payload (shellcode)



1. Legit hosting dll is loaded
2. change permissions to RW
3. overwrite shellcode on section
4. change permissions to RX
5. Execute code via thread creation or function callback

credits to @0xBoku and @WithSecure for the technique
https://blog.f-secure.com/hiding-malicious-code-with-module-stomping/
https://github.com/boku7/Ninja_UUID_Runner/

Module Stomping IoCs

```
Moneta64.exe --option suppress-banner -m ioc -p 20196

 x64 : C:\Users\naksyn\Desktop\VScode-interpreter-python-3.11.3-embed-amd64\python.exe
:0x00aff000    | DLL Image            | C:\Windows\System32\wmp.dll
00:0x0004b000 | RX         | .rsrc     | 0x0004b000 | Inconsistent +x between disk and memory | Modified code
```

IoC

Target for improvement

```
Total scanned:      62
Skipped:            0
-
Hooked:             1
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   1
---
```

IoC

Shellcode payload written over a section of legit hosting dll

modified code IoC is a trademark of Module Overloading and Module Stomping

2

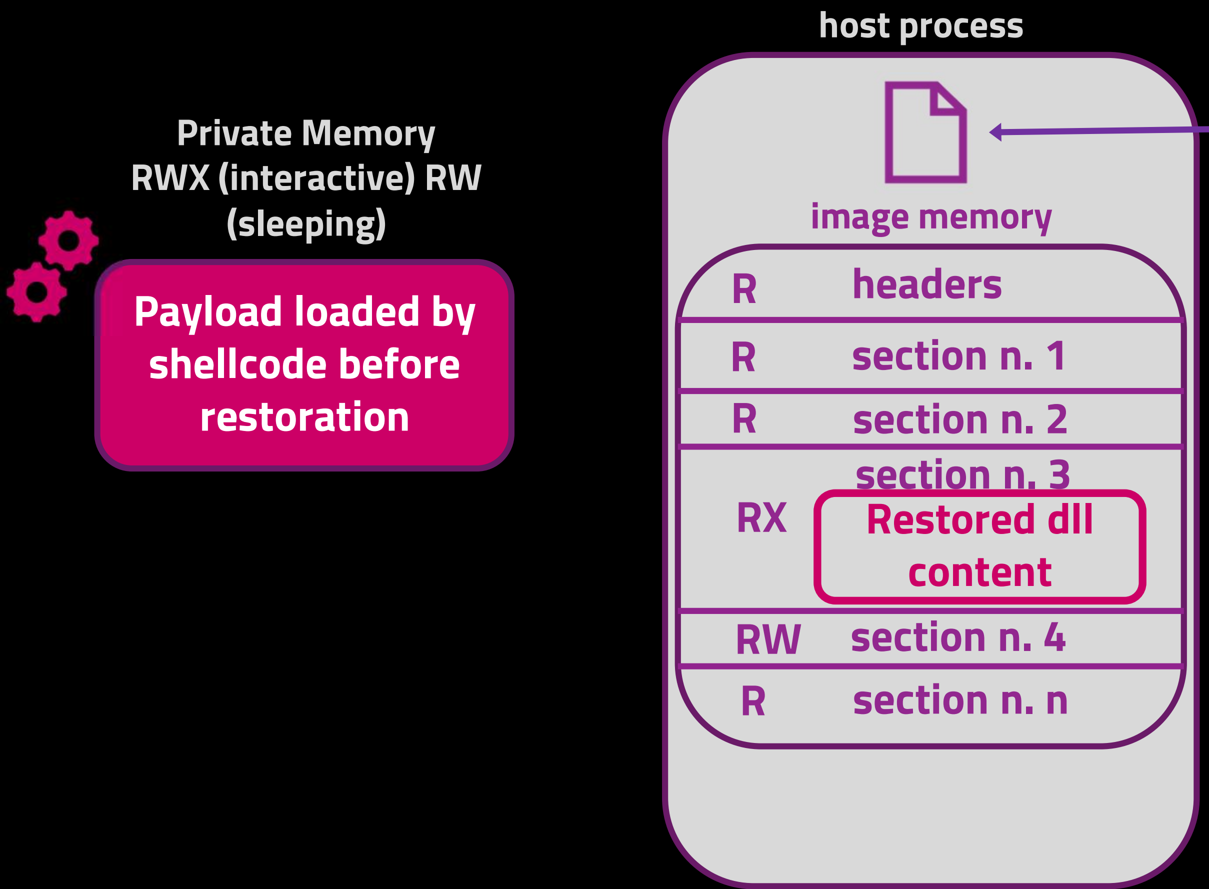
MODULE SHIFTING

Module Shifting

- Implemented in Python ctypes – full-in-memory execution available
- Can be used with PE and shellcode payloads
- Avoids «Modified code» IoC between virtual memory and on disk dll leaving near to zero suspicious memory artifacts, getting no indicators on Moneta and PE-Sieve
- Execution via function pointer avoids callbacks and creating new thread
- better blending into common False Positives by choosing the target section and using padding

Module Shifting

Main Technique Advantage – no resident memory artifact on hosting dll



1. Legit hosting dll is loaded
2. change permissions to RW
3. overwrite shellcode on dll's section
4. padding (optional)
5. change permissions to RX
6. Execute code via function pointer
7. Write original dll content over shellcode

https://github.com/naksyn/ModuleShifting

Module Shifting - Key Points

```
import ctypes
dll_path="C:\\Windows\\System32\\wmp.dll"
hostingdll=ctypes.cdll.LoadLibrary(dll_path)
```

Load

```
for section in self.hostingdllparsed.sections:
   if section.Name.decode().strip('\x00').lower() == self.tgtsection:
       self.tgtsectionaddr= section.VirtualAddress
       self.tgtsectionsize= section.SizeOfRawData
       break
```

Shift

Write

```
self.write_exec_shellcode()
self.restore()
```

Exec

Restore

Module Shifting – Restore modified bytes

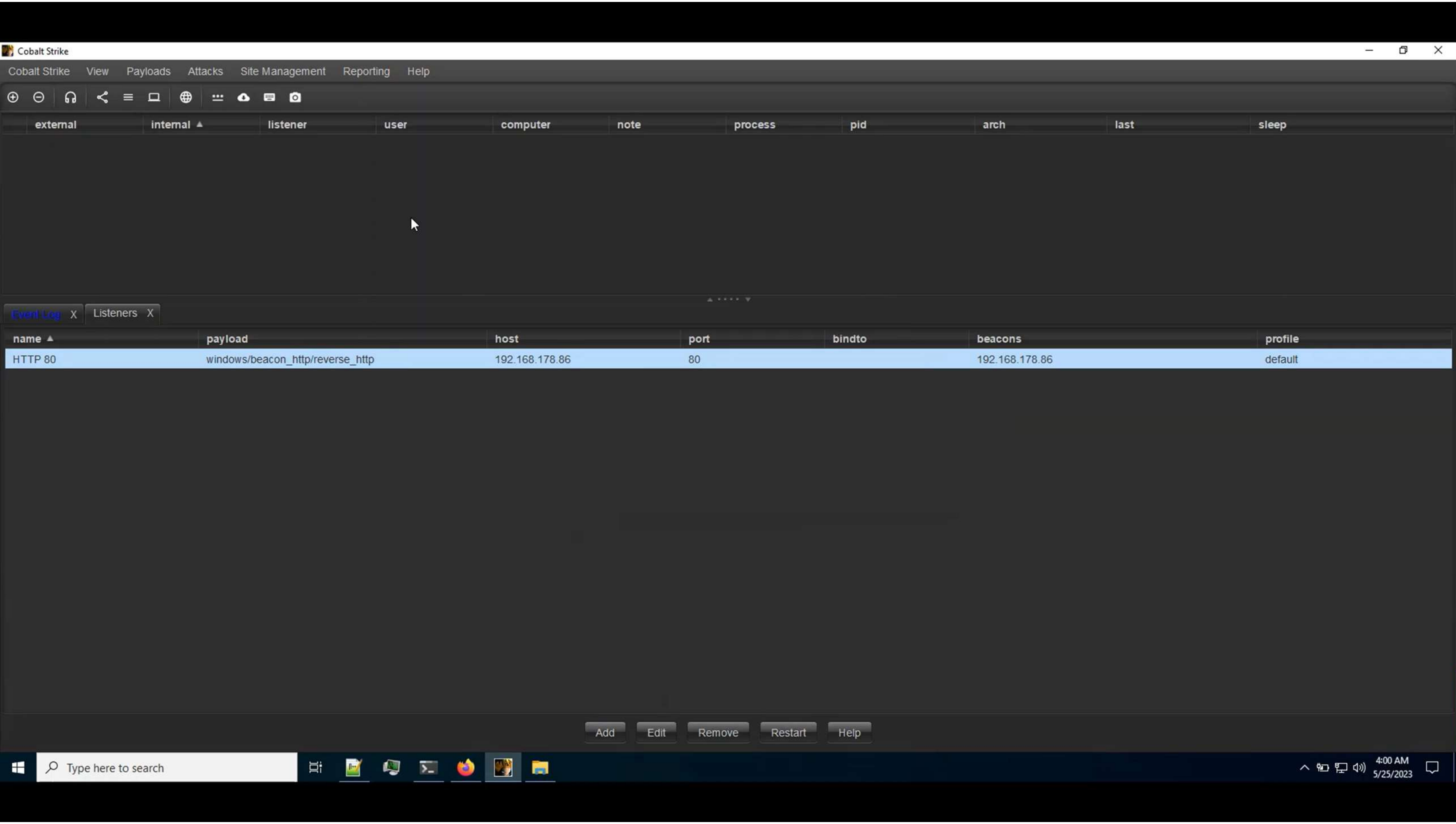
```
def restore(self):

    VirtualProtect(
            cast(tgtaddr,c_void_p),
            mod_bytes_size,
            PAGE_READWRITE,
            byref(oldProtect))

    memmove(cast(tgtaddress,c_void_p), self.targetsection_backupbuffer, mod_bytes_size)
```

Buffer with original dll bytes

copied over the shellcode position


Cobalt Strike
Cobalt Strike View Payloads Attacks Site Management Reporting Help
external internal ▲ listener user computer note process pid arch last sleep
Listeners X
name ▲ payload host port bindto beacons profile
HTTP 80 windows/beacon_http/reverse_http 192.168.178.86 80 192.168.178.86 default
Add Edit Remove Restart Help
Type here to search
4:00 AM
5/25/2023

Module Shifting – IoCs

```
Total suspicious:   0
```

```
... scan completed (1.078000 second duration)
```

- Currently no traces of injection IoCs after Moneta and PESieve scans
- Sleeping AceLdr can be detected by other tools - our focus is on the injection technique

Detection Opportunities

- Module Stomping and Module Shifting need to write shellcode on a legitimate dll
- Module Shifting will eliminate this IoC with cleanup but indicators could be spotted by scanners with runtime inspection capabilities

```
python.exe : 1164 : x64 : C:\Users\naksyn\Desktop\python-3.10.11-embed-amd64\python.exe
  0x00007FFCEC4D0000:0x01600000   | .NET DLL Image      | C:\Windows\assembly\NativeImages_v4.0.30319_64\mscorlib\
c\mscorlib.ni.dll
    0x00007FFCECA0A000:0x0103d000 | RX         | .text     | 0x0004b000 | Modified code
```

307.2 kB of payload

common behaviour for mscorlib.ni.dll is to overwrite 45 kB

Writing more than the usual size can be a malicious indicator

Main Takeaways

- Injection Techniques have several moving parts
- Python can be used as a loader with Pyramid and ctypes to dynamically call windows APIs
- Memory IoCs can be greatly reduced with a proper injection strategy
- Memory scanners can be used by attackers to find False Positives candidates to blend in
- Functionally-independent Shellcode payloads once injected and executed can be overwritten with original dll content
- ModuleShifting improvements can be applied also to other injection techniques

THANKS!

Any questions?

You can find me at:

@naksyn

References

https://github.com/hasherezade/module_overloading

https://github.com/forrest-orr/moneta

https://github.com/hasherezade/pe-sieve

https://www.forrest-orr.net/post/masking-malicious-memory-artifacts-part-ii-insights-from-moneta

https://github.com/kyleavery/AceLdr

https://github.com/boku7/Ninja_UUID_Runner

https://blog.f-secure.com/hiding-malicious-code-with-module-stomping/

https://github.com/ModuleShifting