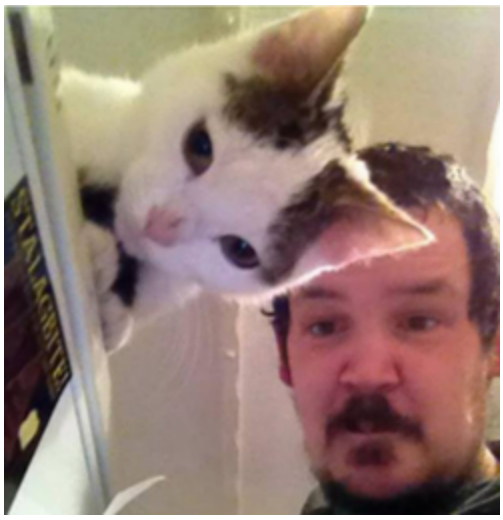
Improving the stealthiness of memory injections techniques

naksyn.com/edr evasion/2023/06/01/improving-the-stealthiness-of-memory-injections.html

Naksyn June 1, 2023



The topic has been presented at x33fcon 2023 Talk - Improving the Stealthiness of Memory Injection Techniques (slide deck is available here)

TL;DR

Injection techniques can be grouped in three main categories:

1. Code Injection
2. PE Injection
3. Process Manipulation

This post focuses on improving Module Stomping and Module Overloading, part of the PE injection techinques, that have been chosen as candidates because they avoid the creation of dynamic memory allocation and perform a common operation (LoadLibrary) that is the cornerstone of the technique.

The public implementation of Module stomping till date are getting “Modified Code” IoC by Moneta because of the stomped code living on the hosting dll.

Moneta will compare the dll bytes on disk with in-memory bytes and the output will be the “Modified Code” IoC. This outcome can be avoided by looking at injection techniques from a higher level and thinking about a proper improvement strategy. In fact, there are several moving parts in an injeciton techniques:

1. the loader

2. the injection technique
3. the payload

If we can keep the payload functionally independent from the stomped bytes, we can restore the stomped bytes and get rid of the "Modified Code" IoC that some module stomping public implementations bring.

Module Overloading, on the other hand, requires having a PE payload living on a "hosting dll" and we cannot revert the copied bytes back to their original value, otherwise this will impair the payload execution.

However, Module Overloading can be improved by choosing the right hosting dll section where to write the payload, and by mimicking some seemingly "strange" behaviour held by windows and third party libraries that overwrite some of their very same PE section, leading to the "Modified Code IoC" with Moneta.

All these improvements led to a modified Module Stomping and Module Overloading technique that has been dubbed Module Shifting. To connect these concepts to my previous Python research I developed the PoC in Python ctypes such that it can be used dynamically with Pyramid.

Credits

A huge thank you to the amazing people that published knowledge and tools instrumental to this work:

- Aleksandra Doniec (hasherezade) for Module Overloading, PE-Sieve, PE-Bear and for technical discussions
- Forrest Orr for Moneta and his Memory Evasion blog series.
- Kyle Avery for AceLdr
- Fsecure and Bobby Cooke for their public Module Stomping implementation (1)(2)

Intro

The purpose of the post is to improve some injection techniques, so to better understand the process involved we'll try to answer the following questions:

1. What's important to know about an injection and how can we choose between the myriad of available techniques
2. How can we test the stealthiness and define a benchmark
3. How can we improve an injection technique.

In the realm of Offensive Cybersecurity, injection techniques play a pivotal role in various malicious activities.

These techniques involve the insertion of code or payloads into the memory space of legitimate processes, often enabling attackers to execute arbitrary actions covertly.

Among the various techniques, three main categories stand out: Code Injection, PE Injection, and Process Manipulation.

In this post, we will delve into the domain of PE Injection, focusing specifically on two advanced techniques: Module Stomping and Module Overloading. Module Stomping and Module Overloading are intriguing techniques within the realm of PE Injection due to their ability to sidestep dynamic memory allocation and rely on a fundamental operation known as LoadLibrary.

These techniques, while effective, have been scrutinized for leaving traces that can be detected by advanced security tools like Moneta. Moneta's detection mechanism involves comparing on-disk DLL bytes with in-memory bytes, effectively flagging modified code as an Indicator of Compromise (IoC). This post addresses the challenges posed by these techniques and presents an innovative approach to enhance their stealth and effectiveness.

Injection Categories

Since our aim is to improve the stealthiness of injection techniques, we'll try to group the injection technique in categories and having a focus on the IoCs that are most commonly left by techniques in a same group. This is by far not a comprehensive description of every injection techniques but the purpose is to provide some high-level overview so that we can better identify promising injection techniques to improve. If you need a more detailed overview, the Blackhat 2019 presentation - Process Injection Techniques: Gotta Catch Them All can be beneficial.

Code Injection

techniques included in this group insert and execute malicious code within a target process's memory, typically involving dynamic memory allocation. Some of the most common techniques in this group are:

1. 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
2. APC injection:
    - Allocate memory in the target process
    - Write malicious code into it
    - queue APC
    - Resume thread execution

3. Hook Injection
    - Intercept API calls made by the target process
    - Redirect the intercepted API calls to the malicious code
4. Thread Local Storage injection
    - modify the target process' PE header (TLS callback function)
    - Execute the injected code as a TLS callback
5. Exception-Dispatching Injection
    - Allocate memory in the target process
    - write malicious code into it
    - modify the target process' exception handler
    - Trigger an exception in the target process

The most prevalent IoC for the techniques listed in this group is the **Dynamic memory Allocation, usually made by VirtualAlloc and HeapAlloc API calls, and subsequent changes in memory permissions** (RWX, RW then RX, etc.) There are also technique-specific IoCs that are generated by some techniques, but they are very peculiar and can generally be fingerprinted by security vendors once a technique becomes public, so for that matter we are mostly interested in the common IoCs shared by most of the techniques in a group, so that we have a simpler map of an injection category and traces left by most of the techniques.

PE Injection

Techniques included in this group inject a Portable Executable (PE) file such as dlls or exes into the address space of a running process. Some of the most common techniques in this group are:

1. Classic dll injection:
    - Drop dll on disk
    - allocate memory to target process and write malicious dll
    - Load dll using LoadLibrary or similar method
2. Reflective dll injection
    - Reflective loader is part of the malicious dll
    - the loader loads and map the malicious dll into target process without actually calling LoadLibrary or other Windows API.
    - Resolve dependencies and perform relocations
3. MemoryModule
    - similar to reflective dll injection but the loader code is external and not embedded in the dll itself.
    - this technique is more flexible since it allows the loading of unmodified dlls.
4. Module Stomping
    - Load a dll into the target process
    - Overwrite dll's section/s with shellcode and execute it

5. Module Overloading
    - Load a dll into the target process
    - Overwrite loaded dll memory space with malicious PE

By injecting a PE, we are requiring that PE to run on the overwritten dlls' bytes and this would typically mean that the PE is a "final" payload that does not load or execute further stages. On the other hand, by using shellcode (i.e. Module Stomping) an attacker can craft a more stealthier approach by using a shellcode that is loading a final payload in another area of memory. As we'll see later in the post, this is a key property that enables some improvements in the injection technique.

Injection techniques in this group mostly leverage, or mimic, a normal dll loading operation such as LoadLibrary. This is a key element that can provide an avenue for attackers to better blend into environments while injecting.

Process Manipulation

Techniques included in this group are used to manipulate or modify the memory and execution context of running processes, libraries, or creating new processes with malicious payloads. Some of the most common techniques in this group are:

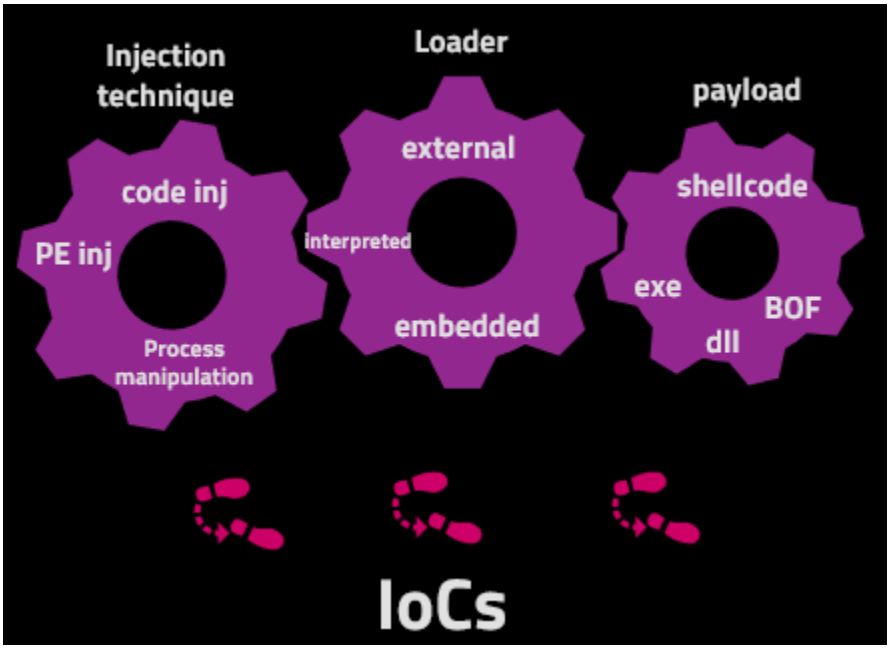
1. Process Hollowing:
    - Create process in suspended state
    - Replace memory contents with malicious executable
    - Resume execution
2. Process doppelgänging
    Abuse NTFS transactions to load a malicious executable within the context of a legitimate process
3. Sideloading
    - Drop dll on disk
    - Abuse windows dll search order or missing dlls to load a malicious dll into a legitimate process
4. Thread Execution Hijacking
    - Suspend a thread in the target process
    - Modify instruction pointer to execute malicious code

The most prevalent IoCs generated by these techniques are **alterating the context or normal execution flow of a PE** (suspend execution state, abuse dll search order).
While this category contain some very powerful techniques, such as sideloading, we might want to first look for techniques that leverages mostly legitimate process' operations and do not alter execution flow, in order to get more chances of blending into an environment without standing out as odd behaviour.

Improvement Strategy

Before diving into the improvement phase, we should have a proper strategy under our sleeves since the injection technique is not a single element but it is part of a **chain composed by the injection technique, loader and the payload as their main moving parts**.

The most prevalent IoC for these techniques is that **the PE (dll or exe) or shellcode, is residing in memory of a (legitimate) loaded dll**. This will lead to a mismatch between in-memory bytes and on-disk dll's bytes caused by the overwriting of the loaded dll memory space with malicious code.



Moving parts of an injection Technique

Moving Parts - Injection technique

The injection technique should not be seen as an isolated element, because its choice can be influenced by the payload or the loader. For example, if your payload to be injected is a PE, you'll basically limit your injection options to the PE injection category. Similarly, if you choose to use an embedded loader to load a dll, you're narrowing down to reflective dll injection.

An attacker should choose an injection technique primarily based on operational considerations, some common drivers might be:

- use an injection to emulate a predefined Threat Actor.
- choose an injection that is more likely to blend into an environment
- use an injection that can bypass a the security solution the attacker is up against (not necessarily blending into the environment).

We are mostly interested in **blending into an environment**, because this can bring the broadest operational depth. For this reason, two key features that the Injection technique should have are:

1. Avoidance of dynamic memory allocation (via VirtualAlloc or HeapAlloc).
2. Usage of a legitimate process operation

Looking for injection techniques techniques with these characteristics we can recall from the Introductory overview that Module Stomping and Module Overloading are two injection techniques that leverage the legitimate LoadLibrary operation to avoid dynamic memory allocation such that malicious shellcode or PE can be written over the loaded dll memory space.

For this reason we chose to **target Module Stomping and Module Overloading** and look for ways to improve them.

Moving Parts - Loader

The purpose of the loader is to execute the injection technique itself, eventually loading and executing a payload. There are mainly three types of loaders:

1. **embedded** - the loader is part of the payload (usually a PE). For example, reflective dll injection uses an embedded loader that is coded in the dll and bootstraps the loading process of the dll itself.
2. **external** - the loader is not part of the payload, it's typically a standalone PE that gets a shellcode, BOF or PE as input payload and kicks off the injection technique. The payload can be written within a section of the loader itself or can be downloaded/read from disk/pipe.
3. **interpreted** - this loader is coded in an interpreted language and executed by the code interpreter. This kind of loaders do not need a purposely compiled PE to run and can be executed in memory by the interpreter that need to be present or dropped on the target.

Building upon my previous Python research, our strategy is adopting an interpreted loader because we'll want to avoid the generation of suspicious PE loaders that generally have a very short life-span can be easily fingerprinted and leverage the powerful evasion properties that Python brings to the game:

1. Python embeddable package comes with a signed interpreter that can be dropped on the target
2. Coding the loader using Python ctypes allows to **dynamically execute wrapped C language** code via Python. We can essentially execute any Windows API using Python via the signed interpreter.

3. Combining Python with Pyramid allows to in-memory import Python modules and execute complex operations entirely in memory.
4. We can avoid the usage of compiled PE for injection.
5. We can avoid AMSI inspection (there's no AMSI for Python) and AV/EDR inspection of dynamic Python code (there's no introspection for dynamic Python code).

Moving Parts - Payload

The final stage of an injection technique is to achieve payload execution, that's essentially code to be run on a target machine. In the context of Memory Injection, payloads can come in the form of:

1. PE (executables or dlls)
2. Position Independent Code (Shellcode, BOFs, etc.)

PE payloads are usually less flexible than shellcode because of their size (PE Header and sections' overhead) and they're also rarely used to stage further malware, instead they're often intended as "final" payloads containing the core of the malware. Furthermore, the size constrait make PE an unviable candidate for injection techniques where little space is available.

On the other hand, shellcode has more flexibility and evasion properties:

- Shellcode can be used to load further stages payloads (even a PE) and can be made independent from final payloads, meaning that once the shellcode loaded and started the final payload, it can be erased without impairing the functionality of the final payload itself.
- Shellcode can be shrank (using stagers for example) to fit small space constraints.
- Position Independent Code payloads can be obfuscated at the assembly level

For this reasons we'll choose shellcode as payload and to make it independent from further stages we'll use a stageless Cobalt Strike generated with AceLdr shellcode.

AceLdr shellcode will load a copy of Beacon on the Heap and it'll apply advanced in-memory evasion techniques. The scope of this blogpost is improving the injection technique rather than the payloads, so we'll be focusing on the artifacts that the injection technique is leaving behind.

Testing with memory scanners

In the realm of cybersecurity, understanding and mitigating novel threats is paramount. For this purpose, great professionals like Aleksandra Doniec and Forrest Orr published Moneta and Pe-Sieve, that are state-of-the-art publicly available memory scanners designed to detect sophisticated memory-based attacks.

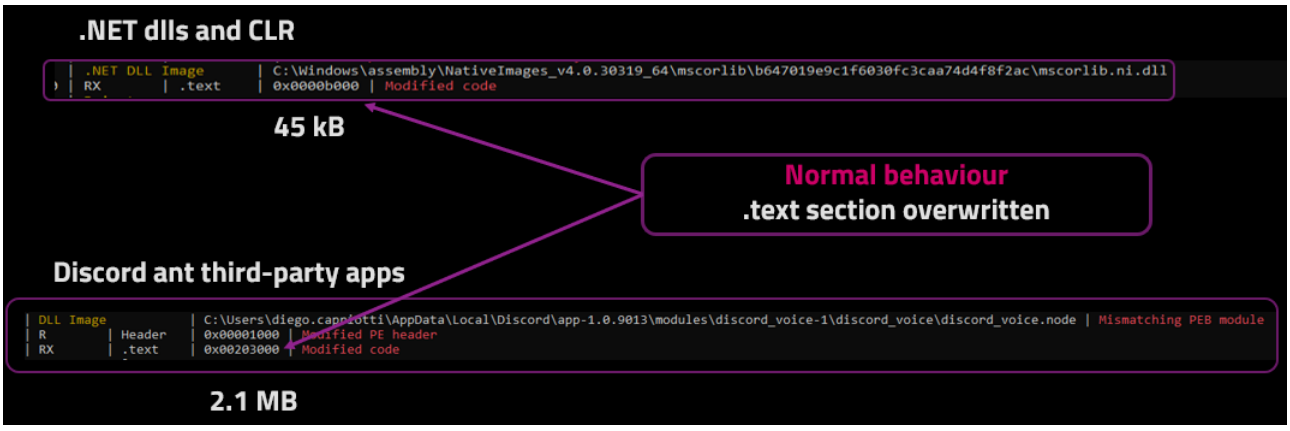
Moneta excels in identifying the presence of dynamic/unknown code and suspicious characteristics of the mapped PE image regions, which are often telltale signs of an attack. On the other hand, Pe-Sieve is designed to identify suspicious memory regions based on malware IOCs and uses a variety of data analysis tricks to refine its detection criteria. These tools were originally designed for defenders, but could be also used by attackers to improve their craft.

When we delve into the intricacies of memory injection techniques like Module stomping and Module overloading, both these tools become instrumental. By utilizing these scanners, we can identify the improvement opportunities in these injection techniques, making it possible to enhance their efficiency when deploying shellcode and ensuring they remain undetected by modern defense mechanisms.

Having these tools at hand is also beneficial infinding some weird common behaviours that we can use to our advantage to better blend in. For example, running Moneta on all processes on a Windows 10 Operating system and inspecting its results, can lead to interesting findings.

In fact, some .NET dlls are known to do self-modifications on their .text section, leading to the Moneta's "Modified Code" IoC. Third-party apps like Discord and Signal also have the same behaviour, it's interesting to note that the size of the bytes that they're overwriting is bigger in the latter cases.

Generally, the bigger the size the dll is self-modifying, the better, since an attacker can smuggle a bigger payload and mimick the exact same behaviour of the legitimate applications. In particular, security solutions would probably whitelist this behaviour otherwise they'll be overwhelmed by false positives and customers will be unhappy.

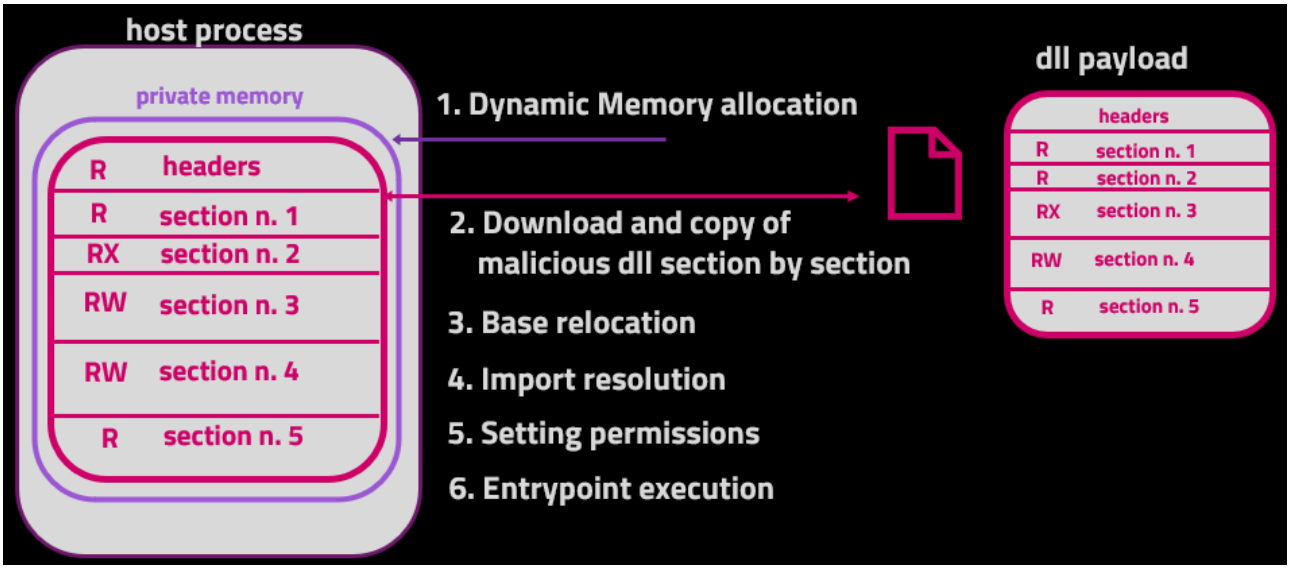


False Positives - self-modifying behaviours done by legitimate applications

Starting Point - PythonMemoryModule

After defining the strategy, we should start somewhere and iterate to improve. Our starting point is the MemoryModule technique, that is instrumental to the Module Overloading injection that we'll target later on.

MemoryModule is a technique firstly published by Joachim Bauch and is used to map and load a dll in memory without calling the LoadLibrary Windows API. This is achieved by executing the same operations done by the Windows Loader when issuing the LoadLibrary API call The following image depicts its basic steps:



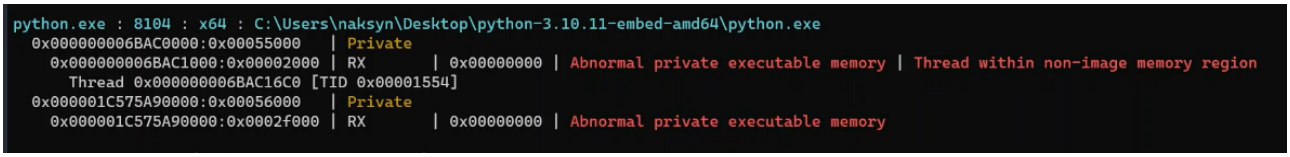
MemoryModule technique

In order to use the MemoryModule technique with a Python interpreted loader, the technique has been ported to Python ctypes and is available on my PythonMemoryModule github project.

Combining the PythonMemoryModule project with Pyramid we can achieve the injection of a Cobalt Strike dll with MemoryModule technique using a full in-memory Python loader. In the following video we'll demostrate the injection and the scanning results of Moneta and PE-Sieve on the injected process.

In summary, PythonMemoryModule used with a Cobalt Strike dll is producing the following IoCs:

```
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
```



IoCs generated by MemoryModule and Cobalt Strike dll Artifact

The Abnormal Private executable Memory IoC detected at 0x6bac1000 is due to the MemoryModule injection technique that copied the .text section at that address and changed its permissions to RX subsequently.

The other abnormal private executable memory IoC is generated because Cobalt Strike dll is self-bootstrapping Beacon in another area of memory (0x1c575a90000) so we basically here have two PEs in memory that are generating IoCs but only one is running Beacon. Dynamic memory allocation would necessarily nead to “Abnormal Private Executable Memory” IoC at some point, se we would want to get rid of this IoC in the first place.

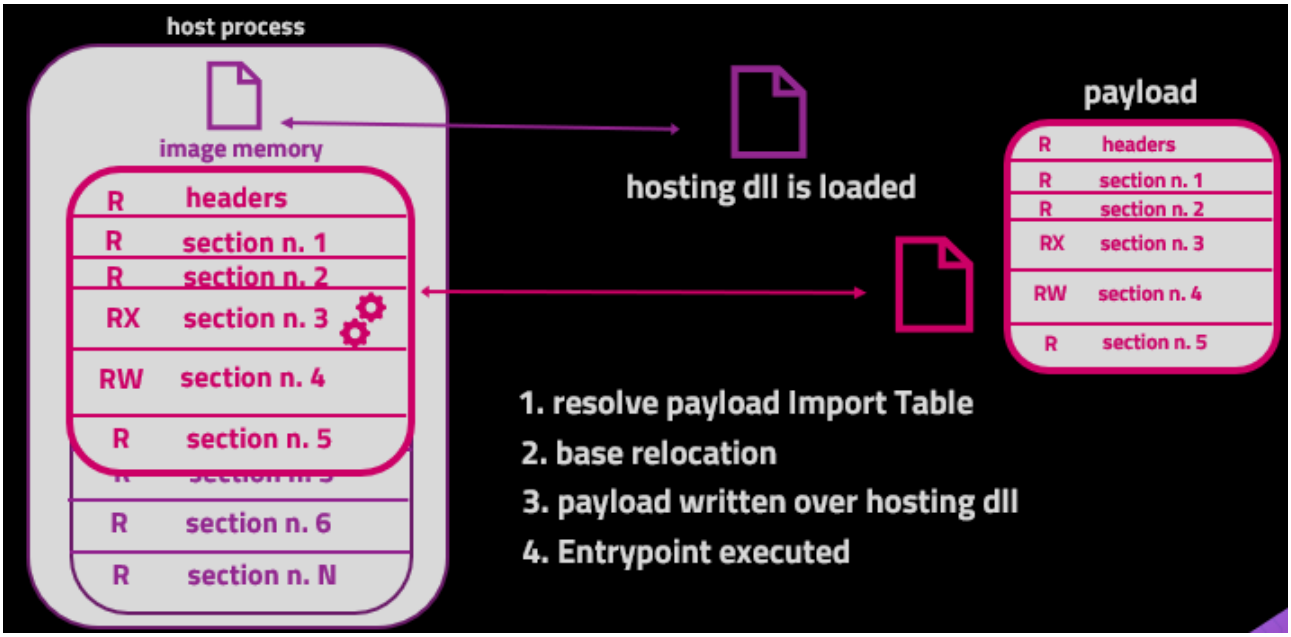
Module Overloading and Module Stomping techniques can provide us a way to avoid dynamic memory allocation.

Module Overloading

Module Overloading technique, firstly published by Aleksandra Doniec, aims at avoiding the creation of dynamic memory allocation by firstly loading a hosting dll using LoadLibrary API, overwriting malicious content (PE) onto it, and loading it using the same Memory Module steps we saw earlier.

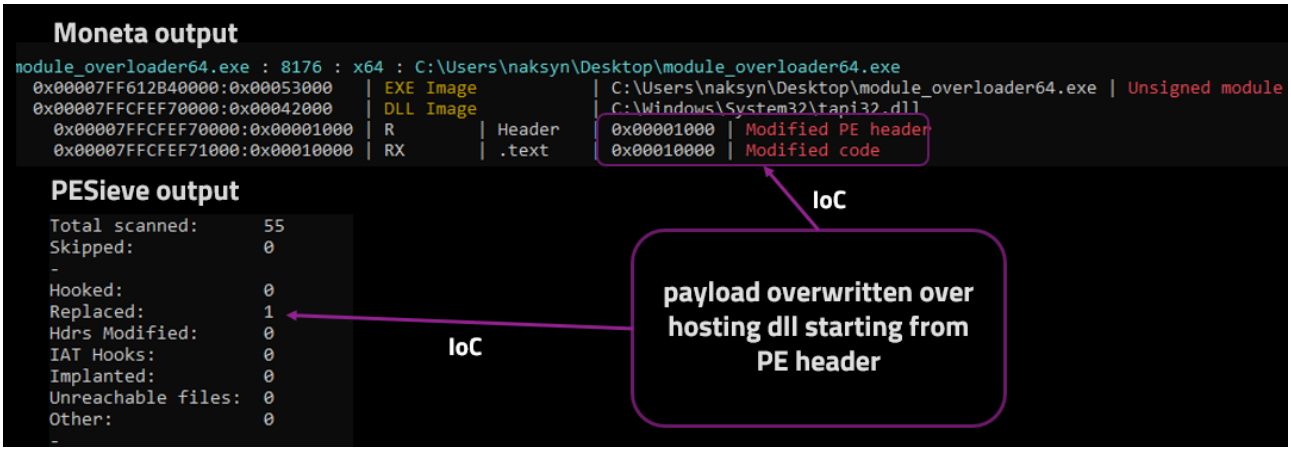
In this way the legitimate hosting dll is loaded via LoadLibrary API, but malicious content is loaded using the Memory Module technique over the memory space of the hosting dll that is legitimately loaded. This clever mix makes the Module Overloading Technique.

At a high level, Module Overloading steps (as implemented by Aleksandra Doniec) look like this:



Module Overloading injection technique

Even though this technique is stealthier than Memory Module, we still have some IoCs to work on. Specifically, Moneta will identify “Modified Code” and “Modified Header” as IoCs after executing the injection.



Module Overloading IoCs

This result stems from the fact that we overwrote the hosting dll memory space with malicious content, so when Moneta and PE-Sieve are doing a comparison between on-disk bytes of the hosting dll with its memory counterpart this will mismatch and fire the “Modified Code” and “Replaced” IoC if the overwriting happen to come across the hosting dll’s mapped .text section.

The “Modified Header” IoC is generated because this technique implementation starts overwriting from the very top of the hosting dll memory space, thus overwriting the PE header that commonly happens to reside in the first 0x1000 bytes.

All things considered, we got rid of the MemoryModule’s “Abnormal Private Executable memory” IoC but we introduced other IoCs related to the hosting dll byte-by-byte comparison between on-disk and memory space.

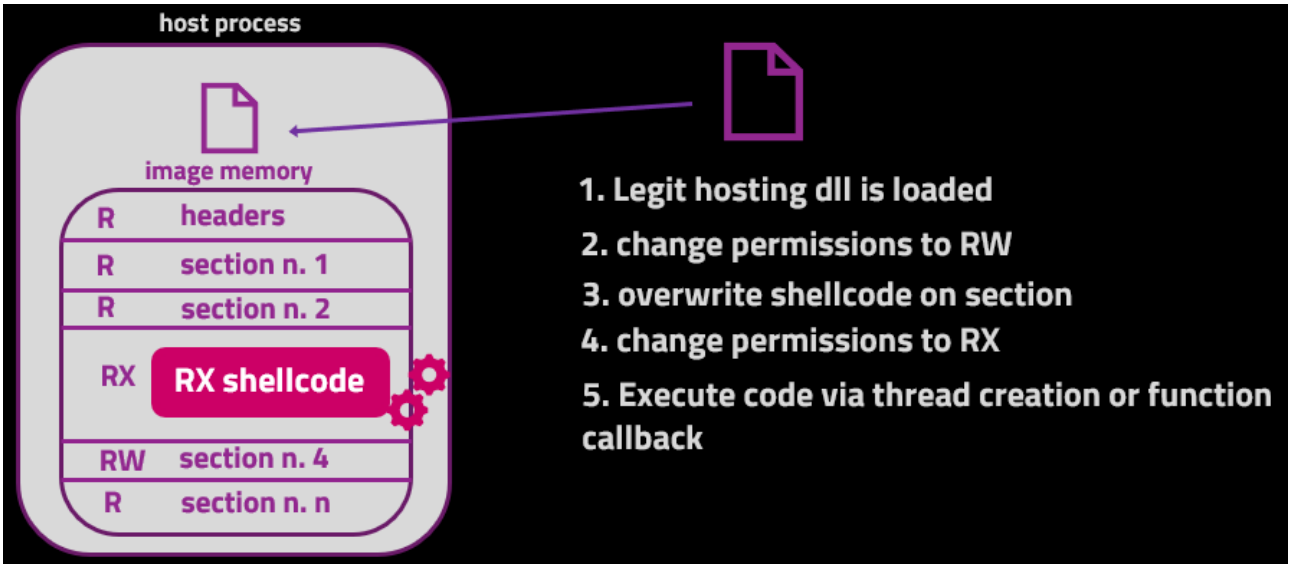
However, we can improve a bit this outcome by introducing Module Stomping injection technique.

Module Stomping

Module stomping provides the same Module Overloading benefit of avoiding dynamic memory creation through the loading of a hosting dll to be used as “disposable space” onto which overwrite malicious content. The main difference is that Module Stomping is way more simpler than Module Overloading because its aim is writing and executing shellcode, not a PE. So we don’t need the Windows Loader steps that both Memory Module and Module Overloading adopted, with Module Stomping we just need to write and directly executing shellcode.

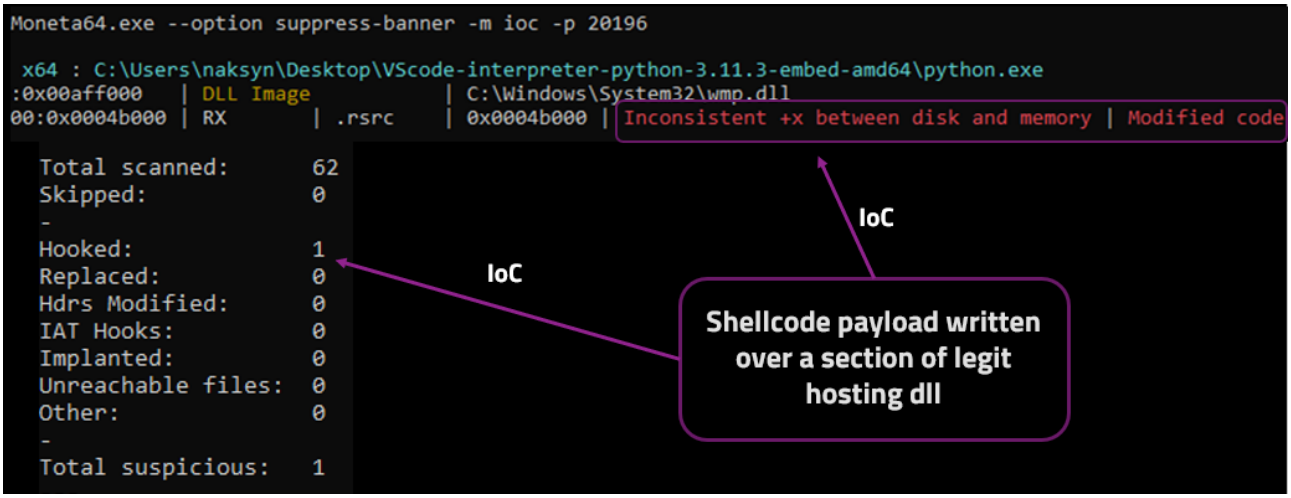
Some Module Stomping implementations have been made publicly available by F-Secure and Bobby Cooke

At a high level, Module Stomping steps look like this:



Module Stomping injection technique

After injecting via Module Stomping using wmp.dll as hosting dll and writing the malicious shellcode over the .rsrc section we obtain the IoCs depicted in the following image.



Module Stomping IoCs

We gradually reduced the generated IoCs but “Modified Code” is still haunting us because it’s a trademark for both Module Stomping and Module Overloading technique. The “inconsistent +x between disk and memory” is obtained because of the shellcode written over the .rsrc section and subsequent +RX permission set. Moneta is complaining about the fact that .rsrc section originally does not have executable permission.

Both of these IoCs can be finally avoided with some improvements that are implemented in a technique dubbed “Module Shifting”.

Module Shifting

Till now we observed how some injections behave in memory and gained a bit of knowledge of how and why memory scanners identifiy suspicious memory anomalies.

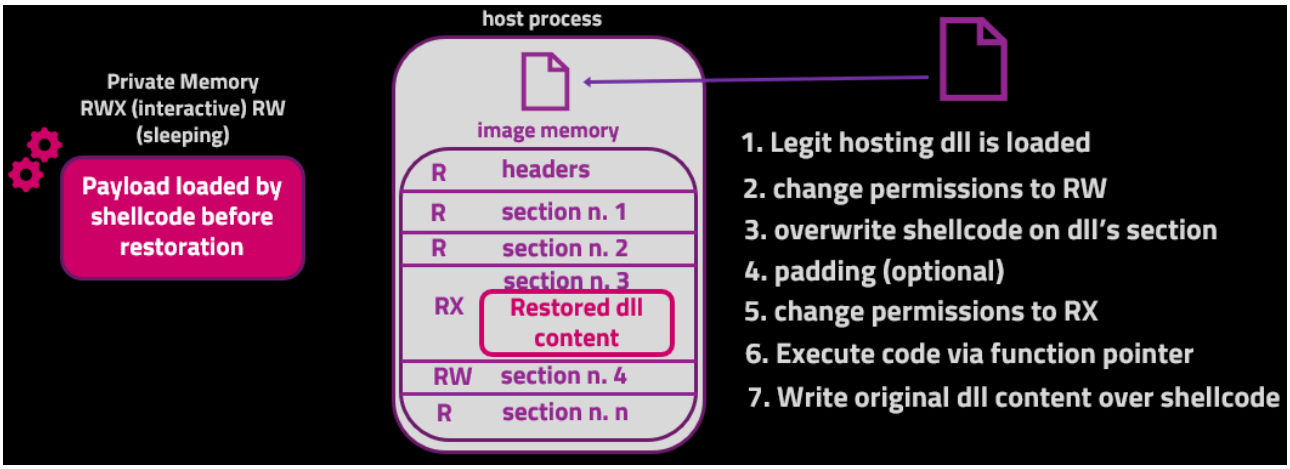
We can use this knowledge to our advantage by asking ourself few what-if questions:

1. what if the writing of the shellcode is shifted to a section of a dll that is normally self-modifying the exact section?
2. what if we inject using a self-modifying dll as host with enough space to write our shellcode and we apply some padding to look exactly as the self-modifying behaviour?
3. what if we use a shellcode payload that is functionally independent from further stages and we overwrite the executed shellcode with the dll’s original bytes?

After experimenting and answering all these questions we came up with the Module Shifting technique that aims at improving Module Stomping and Module Overloading by providing the following advantages:

1. Avoids “Modified code” between virtual memory and on disk dll leaving near to zero suspicious memory artifacts, getting no indicators on Moneta and PE-Sieve
2. better blending into common False Positives by choosing the target section and using padding
3. Can be used with PE and shellcode payloads
4. Implemented in Python ctypes – full-in-memory execution available

At a high level, Module Shifting steps look like this:



Module Shifting Injection technique

The restore operation is quite simple and is done after executing the initial shellcode.

```
# Restore operation
        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)
```

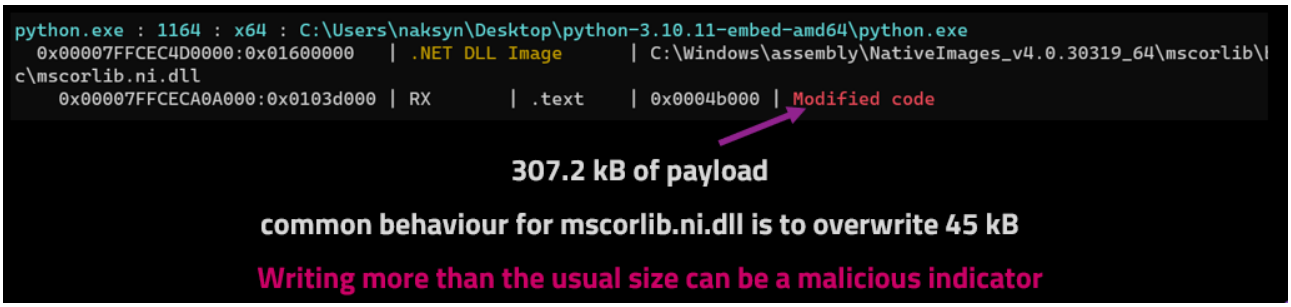
After setting the shellcode memory area permissions to RW the content of **targetsection_backupbuffer**, containing a copy of the original dll for the same exact amount of shellcode bytes and position, gets written over the shellcode. This effectively restores the stomped bytes to the original ones, leaving no traces of the written shellcode anymore. In this way, Moneta and PE-Sieve will do a byte-by-byte comparison as usual and will find no mismatch between the hosting dll on-disk bytes and in-memory ones.

There won't be also any inconsistent executable permissions because we set the permissions back to the section's original values.

Following is a demonstration of a self-process injection with Module Shifting technique using a Cobalt Strike Beacon shellcode generated with AceLdr. **After executing Moneta and PE-Sieve we get no IoCs detected** because there are no artifacts left by Module Shifting injection technique (payload is not our focus), that was our initial aim.

Even though Moneta and PE-Sieve did not generate IoCs, a runtime inspection scanner could identify some anomalies. In fact, overwriting a 307,2 kB payload over the .text section of mscorlib.ni.dll can be a malicious indicator because the common behaviour for this dll is to overwrite 45 kB.

However, this anomaly could not be spotted by scanners without runtime inspection capabilities, because Module Shifting does not leave artifacts floating around after having restored the stomped bytes.



Detection Opportunities

Outro

Concluding this exploration, we dove deep into the intricacies of injection techniques, honing in on Module Stomping and Module Overloading as part of the PE injection arsenal.

The objective was clear: to improve these techniques, aiming for more operational stealthiness. We delved into the journey of improving memory injection techniques While traditional approaches like Module Stomping faced challenges with "Modified Code" IoC due

to the stomped code's residence in the hosting dll, we've delineated a strategy to finally circumvent these obstacles. The newly introduced Module Shifting technique encapsulates these enhancements, offering a more nuanced way to to operate with a greater stealthiness.

The key takeaways for this blog post are:

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

The future of injection techniques is always evolving, and the landscape will continually shift towards greater sophistication and precision.

←