Hunting In Memory

By Joe Desimone

Published: 2022-06-21 · Archived: 2026-04-05 13:45:02 UTC

Threat Hunters are charged with the difficult task of sifting through vast sources of diverse data to pinpoint adversarial activity at any stage in the attack lifecycle. To be successful, hunters must continually hone their subject matter expertise on the latest attacker techniques and detection methods. Memory resident malware, which presents itself in many forms, is an attacker technique that has existed for over a decade. The popularity of memory resident malware has steadily increased over time, possibly resulting from the proliferation of code and knowledge of in memory techniques. More likely, its popularity reflects the success of memory-based techniques to evade detection by security products and practitioners. Once limited to advanced adversaries, memory resident techniques are now commonplace for all levels of adversary sophistication. I will examine the most common of these memory based attacker techniques, and walk through our team's research to craft a scalable, low noise approach to hunting for adversaries that are hiding in memory.

Attacker Techniques

Before I address memory hunting methods to detect adversaries in your network, it is helpful to understand the common forms of memory resident malware. These techniques include shellcode injection, reflective DLL injection, memory module, process and module hollowing, and Gargoyle (ROP/APC).

SHELLCODE INJECTION

Shellcode injection is the most basic in-memory technique and has also been around the longest. The basic 'recipe' for shellcode injection is a four step process. These steps are: 1) open a target process (OpenProcess); 2) allocate a chunk of memory in the process (VirtualAllocEx); 3) write the shellcode payload to the newly allocated section (WriteProcessMemory); and 4) create a new thread in the remote process to execute the shellcode (CreateRemoteThread). The venerable Poison Ivy malware uses this technique, which is a big reason why so many APT groups were drawn to it over the years.

If you pull up a Poison Ivy samplewith x64dbg and set a breakpoint on VirtualAllocEx, you will soon locate the chunk of code responsible for the injection.

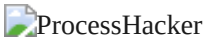

Shellcode Injection


Shellcode Injection

In the first image, the push 40 instruction preceding the call to VirtualAllocEx corresponds to page access protection value of PAGE_EXECUTE_READWRITE. In the following screenshot from ProcessHacker of the memory layout of a Poison Ivy implant, you can see it allocates a number of these RWX sections.


Poison Ivy Implant
Poison Ivy Implant

Typical code sections are of type ‘Image’ and map to a file on disk. However, these are type ‘Private’ and do not map to a file on disk. They are therefore referred to as unbacked executable sections or floating code. Threads starting from these types of memory regions are anomalous and a good indicator of malicious activity. ProcessHacker can also show you the call stack of the malware threads. There are multiple functions in the call stack which do not map to memory associated with loaded modules.


ProcessHacker
ProcessHacker

REFLECTIVE DLL INJECTION

Reflective DLL injection, originally developed by Steven Fewer, is another type of in memory attacker technique. Metasploit’s Meterperter payload was one of the first attempts to fully weaponize the technique, but many malware families use it today. Reflective DLL injection works by creating a DLL that maps itself into memory when executed, instead of relying on the Window’s loader. The injection process is identical to shellcode injection, except the shellcode is replaced with a self-mapping DLL. The self-mapping component added to the DLL is responsible for resolving import addresses, fixing relocations, and calling the DllMain function. Attackers benefit from the ability to code in higher level languages like C/C++ instead of assembly.

Classic reflective DLL injection, such as that used by Meterpreter, is easy for hunters to find. It leaves large RWX memory sections in the process, even when the meterpreter session is closed. The start of these unbacked executable memory sections contain the full MZ/PE header, as shown in the images below. However, keep in mind that other reflective DLL implementations could wipe the headers and fix the memory leak.


Unbacked executable memory sections
Unbacked executable memory sections


unbacked executable memory sections
unbacked executable memory sections

The DLLs loaded in memory also conveniently export a self-describing function called ReflectiveLoader().


Reflective Loader
Reflective Loader

MEMORY MODULE

Memory module is another memory resident attacker technique. It is similar to Reflective DLL injection except the injector or loader is responsible for mapping the target DLL into memory instead of the DLL mapping itself. Essentially, the memory module loader re-implements the LoadLibrary function, but it works on a buffer in memory instead of a file on disk. The original implementation was designed for mapping in the current process, but updated techniques can map the module into remote processes. Most implementations respect the section permissions of the target DLL and avoid the noisy RWX approach.

NetTraveler is one malware family that uses a memory module style technique. When NetTraveler starts, it unpacks the core functionality and maps it into memory. The page permissions more closely resemble a legitimate DLL, however the memory regions are still private as opposed to image.


NetTraveler page permissions
NetTraveler page permissions

The active threads have start addresses at these private regions. The callstack also reveals these malicious sections.


endgame-callstack-malicious-sections.png
endgame-callstack-malicious-sections.png

Winnti is yet another malware sample that uses the Memory Module technique. They had a minor slip on the section permissions of the first page, as you can see below.


endgame-section-permissions.jpg
endgame-section-permissions.jpg

However, the Winnti sample was notable because the MZ/PE headers in the DLL were erased, making it more difficult to detect.


_endgame-DLL.jpg
_endgame-DLL.jpg

PROCESS HOLLOWING

Process hollowing is another technique attackers use to prevent their malware from being detected by security products and hunters. It involves creating a suspended process, unmapping (hollowing) the original executable from the process, allocating and writing a new payload to the process, redirecting the execution of the original thread to the new payload with SetThreadContext, and finally calling ResumeThread to complete. More stealthy variants use Create/Map section APIs to avoid WriteProcessMemory. Others modify the entry point with a jump instead of using SetThreadContext.

DarkComet is one of many malware families that use process hollowing techniques. Several artifacts can be used to detect process hollowing. One dead giveaway for this activity is a process being spawned with the CREATE_SUSPENDED flag, as shown in the following screenshot from a DarkComet sample.


_endgame-darkcomet-sample.jpg
_endgame-darkcomet-sample.jpg

MODULE OVERWRITING

So far, all techniques discussed have led to the execution of non-image backed code, and were therefore fairly straightforward to detect. Module overwriting, on the other hand, avoids this requirement, making it much more difficult to detect. This technique consists of mapping an unused module into a target process and then overwriting the module with its own payload. Flame was the first widely publicized malware family to use this technique.

More recently, Careto and Odinaff malware families have used module overwriting techniques. Various techniques can be used to reliably detect module overwriting, which involves comparing memory to associated data on disk.

GARGOYLE

Gargoyle is a proof of concept technique for memory resident malware that can evade detection from many security products. It accomplishes this feat by laying dormant with read-only page protections. It then periodically wakes up, using an asynchronous procedure call, and executes a ROP chain to mark its payload as executable before jumping to it. After the payload finishes executing, Gargoyle again masks its page permissions and goes back to sleep. One way to detect this attacker technique is to examine threads and user APCs for evidence of ROP chains.

Detecting In-Memory Attacks

Given the proliferation and accessibility of these techniques, security personnel must be vigilant for memory-based attacker techniques and proactively hunt for them on their networks. However, most products cannot generically detect in-memory attacks at scale, leaving defenders with an enormous gap in their ability to protect against these attacks. Endgame has done significant research to bring low-noise detection capabilities into our product for each method mentioned above.

Given the immense size and impact of this detection gap, it is important to raise all boats, not just those of our customers. For this reason, we collaborated with Jared Atkinson on his powershell tool called Get-InjectedThreads, which implements a relatively low-noise method of detecting in memory threats. It scans active threads on the system for suspicious start addresses. Hunters leverage it to scan hosts in their networks and quickly identify many memory resident malware techniques. The script works by querying each active thread with the NtQueryInformationThread function to retrieve its start address. The start address is then queried with the VirtualQueryEx function to determine the associated section properties. If the memory region where the thread started is unbacked and executable (i.e. not image type and has execute bit set), then the thread is considered injected. The following screenshot shows a sample detection when run on a system infected with a 9002 RAT sample.


_endgame-RAT-sample.jpg
_endgame-RAT-sample.jpg

The script will catch a variety of malware families leveraging the shellcode injection, reflective DLL, memory module, and some process hollowing techniques. However, it is no replacement for security products that comprehensively prevent in-memory attacks, such as Endgame.

Enterprise In-Memory Detection at Scale

Endgame has built detections for each of these techniques (and many more) into our enterprise security platform, offering best in market capabilities to locate in-memory threats. We do not simply rely on naïve approaches like monitoring well-known system call sequences for process injection, but efficiently analyze memory to find all known evasion capabilities. This provides our users with thread-level visibility on injected code, as well as sophisticated follow-on actions like examining the injected code and suspending only a malicious injected thread

to remediate the threat. Our platform is effective both in stopping injection as it is happening in real time as well as locating already resident adversaries hiding in memory, locating threats across tens of thousands of hosts in seconds.

Like any signatureless detection technique, false positives (FPs) are an important consideration. As we researched and implemented our technique-based preventions for each adversary technique described above, we initially encountered FPs at every step of the way. Handling these correctly in our product is of paramount importance.

Most FPs are related to security software, Just-In-Time (JIT) compiled code, or DRM protected/packed applications. Security products sometimes inject code to some or all processes on the system to enhance their behavioral detection capabilities. The downside is if the product is sloppy in its methods, it can actually harm the security of the system and make hunting for real in memory threats more difficult. JIT code, another potential area for false positives, generates assembly code at runtime which lives in unbacked or floating memory regions. .NET or Java applications are a couple of examples which use JIT techniques. Fortunately, this type of code is easier to identify and filter than rogue security products. Lastly, applications packed or protected with Digital Rights Management (DRM) schemes should be kept in mind. These applications may decrypt or deobfuscate their core functionality in memory to deter debugging and reverse engineering. However, the same techniques are used by malware to evade detection and deter analysis from security practitioners.

Through careful design decisions and extensive testing, we have managed to achieve very low false positive rates, allowing Endgame users to root out in-memory threats rapidly.

Conclusion

Adversaries will continue to innovate new techniques to avoid detection and accomplish their objectives. Memory resident techniques are no exception, and have been a thorn in the side of endpoint security defenders for over a decade. Fortunately, by understanding the latest techniques, we can turn the tables and use this knowledge to develop new high fidelity detection methods. At Endgame, our comprehensive approach to these attacks have led us to a market leading position for fileless attack detection (adding to our other key technologies). For more on hunting for in-memory attacks, check out our slides from our SANS Threat Hunting and IR Summit presentation.
