**ThreadDread**

**Author:** Gary “kd” Contreras

**Version:** Beta 1.4

**Language:** C++

**Dependencies:** Win32 API, Standard C Library, libyara-3.5.0

**Project:** Temporarily “closed-source”

**Background**

This project originally began as a way to hunt and eradicate injected threads, hence the name “ThreadDread”. It has since evolved as a way to perform more complete memory analysis workflows on a running system without having to create a full memory dump first. We would want to do this for multiple reasons:

1. Improve detection/analysis/response times
2. Fill capability gaps in currently used toolsets
3. Enable the ability to respond and hunt with a single tool
4. Maintain a standalone, portable tool for such purposes

**Use Cases**

In some cases, an adversary may be injected into a system service process; depending on how dependent the rest of the operating system is on that process, the blue team may not be able to remove the adversary without rebooting the machine. ThreadDread solves this problem by being able to kill the malicious thread without harming the rest of the process.

In many cases, there will be a user with a desktop session on the infected machine, giving the analyst responding to the incident only two options; spawn an interactive/remote shell session, or kick the user out of their desktop session via RDP. ThreadDread solves this problem by being entirely command-line driven.

It has been found that certain “free” memory analysis toolsets, such as the Volatility Framework are maintained by the freelance community developers and as such come packaged with plugins that may no longer work properly. In addition, the system profiles in many cases are outdated, which tends to produce improper output for a lot of the plugins that ARE, otherwise, still functional. Finally, transferring a memory dump from an overseas network can be very time consuming, allowing the adversary ample time to pivot away from the infected host. ThreadDread solves all of these problems by using real-time data on the running host, which does NOT require a specific system profile to run.

**Capabilities**

ThreadDread allows for many memory analysis workflows to be completed on a running system without the need to collect a full memory dump first. Below is a comprehensive list of such workflows:

1. Listing running processes on the system
2. Listing running threads on the system, or within a specific process
3. Listing modules (binaries) that have been mapped within a specific process
4. Yara scanning all virtual memory within a process, or all processes on the system
5. Querying specific virtual memory addresses within a process
6. Dumping specific (or all) virtual memory allocations within a process
7. Automatic detection of injected threads based on their start address
8. Automatic detection of injected binaries based on their memory protections
9. Suspending/Resuming/Killing of interesting threads, based on their unique Thread ID
10. The ability to specify a central collection server for scanner output (for threat hunting)
11. Printing a memory legend which aids in understanding the output generated by the tool

**Terminology**

In order to analyze the output of the tool known as ThreadDread, it is imperative that the analyst using the tool has a good understanding of memory analysis and the artifacts that can be found in volatile memory. The following terminology should help remove confusion:

Pages:

The Windows operating system allocates memory by pages; each page of memory is 4096 bytes or 4 kilobytes in size.

Virtual Memory:

Most modern operating systems use a virtual memory scheme, where each process thinks it has all the physical RAM to itself, however, the Operating System is handling the mapping between Virtual and Physical memory addresses behind-the-scenes. When we work with process memory, we are dealing with virtual memory.

Physical Memory:

The physical hardware address of memory where virtual pages are mapped. An important distinction here is that memory pages that are “contiguous” (adjacent to each other) in virtual memory are highly unlikely to be contiguous in physical memory.

Protection:

In terms of memory, “protection” is synonymous with the “permission” applied to a page of memory, such as “read”, “write”, or “execute”. These permissions are applied at the page level, so one entire page of memory MUST have the same protection applied to each byte within the page.

Allocation:

When a program requests memory from the Operating System, it will either “Reserve” or “Commit” the request. The difference is that a reserved region is unusable until committed.

Region:

A Region is a “subdivision” within a memory allocation, where all adjacent pages share the same memory protection. For example, the allocation might be 12KB, but the first page only has “read” permission, while the next two pages have “read/execute” permission. In this case, there are two regions within the allocation; one that is 4KB and one that is 8KB; together they make up the full allocation of 12KB.

Address:

All data that exists in memory will exist at a specific address. Each allocation will have an allocation base address (where the allocation begins). Each region will also have a region base address (where the region begins).

Offset:

An offset is a relative position from the start of data. For example, if the allocation base address is 0x11220000 and a PE signature of “MZ” is at address 0x11221000, then the signature is found at offset 0x1000, because 0x11221000 – 0x11220000 = 0x1000.

Threads:

A thread is a computing resource; the Operating System assigns processing time to threads so that they can execute the instructions within a program. Threads execute instructions, processes do not!

Processes:

A process is a container of resources allocated by the operating system that describe a running program, as well as maintain its running state. For example, a thread is a resource assigned to a running process, just like an open file or a network connection.

Binaries:

Binaries are typically files, such as executables or DLLs, commonly known as PE files in Windows. Binaries typically exist in an “unmapped” state on disk and a “mapped” state in memory

“Unmapped” Binaries:

An unmapped binary typically has its internal sections aligned to sectors on disk, where each sector is typically 512 bytes large. This is referred to as “raw” alignment.

“Mapped” Binaries:

A mapped binary will have its internal sections aligned to pages in memory, where each page, as we know, is commonly 4096 bytes large. This is known as “virtual” alignment. Some binaries, which are meant to be injected, will often have their “raw” and “virtual” alignments be the same, which makes injection easier.

“Dumping”:

When we mention dumping data, we typically refer to extracting the contents of one container and outputting them to another container; for example, extracting the contents of a region of memory and outputting them to a file on disk.

“Carving”:

The difference between dumping and carving data is a matter of precision. When we dump data, we are considering the start of readable data and the total length of the readable data, not accounting for anything specific in between. When we carve, we are considering a specific subset of that data, typically beginning at a particular offset and often extending to a length that does not necessarily match the end of the larger data dump.

**Usage Information**

Print the usage information to find the various options with the “-h” option:

.\ThreadDread.exe -h

Optionally, include the memory constant legend (works with scanner outputs too):

.\ThreadDread.exe -h -v

**ThreadList Output**

Run a thread listing with the “-l” switch and, optionally, a target process id “-p <pid>”.

To find all running threads on the system:

.\ThreadDread.exe -l

To find all running threads within a specific process:

.\ThreadDread.exe -l -p 6688

Whenever you run a thread listing (process-specific or system-wide) you will see lines of output similar to the following:

ThreadList DESKTOP-BF2Q2B7,6692,2021-10-25 03:50:40 Z,6688,2021-10-25 03:50:40 Z,00007FF7E5084E80,00007FF7E5084000,0x4000,0x20,svchost.exe

In order from left-to-right, this is the flow of information:

1. The type of output
2. The originating host’s name
3. The Thread ID
4. Thread start time (in Zulu)
5. The Process ID in which the thread is operating
6. Process start time (in Zulu)
7. The memory address where the thread began its execution
8. The memory region to which the previous memory address belongs
9. The size of the memory region
10. The current protection applied to the region
11. The module (EXE or DLL) mapped at region

One interesting method of hunting injected malware is by running such a thread listing and searching for lines of output where the module is marked “blank” at the end of the line. This indicates that the thread was not started inside of a legitimately loaded module (EXE or DLL) and often can be used to find injected shellcode or binaries!

**ProcessList Output**

Run a process listing with the “-L” switch. The “-p” makes no difference and will not be used.

.\ThreadDread.exe -L

Whenever you run a process listing, you will see lines of output similar to the following:

ProcessList DESKTOP-BF2Q2B7,1104,1688,1,2021-10-20 10:44:17 Z,notepad.exe

In order from left-to-right, this is the flow of information:

1. The type of output
2. The originating host’s name
3. The Process ID
4. The Parent Process ID
5. The number of threads executing inside the process
6. The process start time in Zulu
7. The process name

**ModuleList Output**

Run a module listing with the “-m” switch and must specify a target process id “-p <pid>”.

.\ThreadDread.exe -m -p 1104

Whenever you run a module listing, you will see lines of output similar to the following:

ModuleList DESKTOP-BF2Q2B7,1104,notepad.exe,00007FF696160000,0x3A000,C:\Windows\system32\notepad.exe

In order from left-to-right, this is the flow of information:

1. The type of output
2. The originating host’s name
3. The Process ID
4. The name of the module (exe or dll) found
5. The memory allocation base address where the module was loaded
6. The size of the module in memory
7. The full path to the module on disk

**MemQuery Output**

Run a memory query with the “-q <address>” switch and a target process id “-p <pid>”.

.\ThreadDread.exe -q 7FF7E5084000 -p 1104

Whenever you run a memory query, you will see lines of output similar to the following.

In order from left-to-right, this is the flow of information:

MemQuery 00007FF7E5084000,00007FF7E5080000,0x1000,0x2,0x1000000,0x1000,00007FF7E5080000,0x80

1. The type of output
2. The memory address that was queried
3. The base of the memory region for the address that was queried
4. The size of the region
5. The current protection of the region
6. The type of allocation associated with the region (mapped, image, or private)
7. The state of the region (0x1000 means “committed” and therefore “useable”
8. The memory allocation base address
9. The initial protection applied to the allocation

MemQuery will continue to display multiple lines of the similar information for each region that is found within the same memory allocation. If the optional “-d” switch is applied, then committed regions will be dumped automatically to a “.dmp” file, along with the memory address mappings into a “.txt” file for easier analysis.

**PE Injection Scanner**

Run a binary injection scan with the “-B” switch; scans all possible processes on system.

.\ThreadDread.exe -B

Whenever you run an injected PE file scan, you will see lines of output similar to the following.

In order from left-to-right, this is the flow of information:

PEScan DESKTOP-BF2Q2B7,2068,00000229A0940000,0x31000,0x40,0x40

1. The type of output
2. The originating host’s name
3. The Process ID where the suspected injection was found
4. The allocation base address where the suspected injection begins
5. The size of the region
6. The initial protection set on the allocation
7. The current protection set on the region

It is important to note that the PE scanner may not be 100% accurate and should, for the most part, be used as a supporting scanner once something interesting has already been found. It looks for multiple PE header attributes in memory entirely because some malware authors like to clear the signatures out after loading; instead, it looks for memory allocations with RWX protection that are not mapped from a binary on disk and reports those allocations. This scanner also scans ALL possible running processes to do its work.

**Safety Kill Switch/Timeout**

By default, ThreadDread spawns an additional thread with a sleep timer. If this sleep timer expires before ThreadDread finishes its work, the secondary thread is meant to make ThreadDread kill itself so that it does not continue consuming resources longer than necessary to complete its work. The default timeout is 15 minutes, but can be set to some arbitrary minute value via the command line option “-T <minutes>”. This is meant so that ThreadDread can be safely deployed to the enterprise with less concern about its stability.

**Full Process Dump**

Run a full process memory dump with the “-d” switch and “-p <pid>” to specify which process:

.\ThreadDread.exe -d -p 1104

This command will capture all committed regions of memory to a single dump file, and capture all the relevant mappings to a text file. The mappings basically state the base address of the memory region that was dumped, the size of the region, the type of region, the protection applied to the region, and that region’s offset in the dump file. Verify it with a memory query!

**Thread Suspend/Resume/Kill**

ThreadDread’s core functionality allows you to manipulate threads. The suspend, resume, and kill functionality is called with the “-s”, “-r”, and “-k” switches, respectively, along with the Thread ID that you want to perform the action on. The purpose of this is to be able to safely remove malicious threads from important system service processes without needing to kill the process or reboot the machine once the malicious thread has already been identified.

To suspend a thread:

.\ThreadDread.exe -s 864

To resume a thread:

.\ThreadDread.exe -r 864

To kill a thread:

.\ThreadDread.exe -k 864

**Hunting With Netcat**

ThreadDread was initially meant to hunt injected threads and report them to the analyst on the command line. It can now also send that output across the network to a persistent listener, such as Netcat. Specifying an IP address with “-A <ip address>” and an arbitrary port number with “-P <remote port>” allows things like the thread scanner, yara scanner, and binary inject scanner to redirect their output to the remote listener for capture. Below are some examples:

To redirect the injected thread scanner output to the remote listener:

.\ThreadDread.exe -A 127.0.0.1 -P 8000

To redirect the binary injection scanner output to the remote listener:

.\ThreadDread.exe -B -A 127.0.0.1 -P 9000

To redirect the Yara scanner output to the remote listener:

.\ThreadDread.exe -Y <rule\_file\_path> -A 127.0.0.1 -P 10000

**Yara Scanner**

To run a Yara scan against ALL running processes on the system:

.\ThreadDread.exe -Y <rule\_file\_path>

To run a Yara scan against a specific process on the system:

.\ThreadDread.exe -Y <rule\_file\_path> -p <pid>

The output will look like the following:

YaraScan DESKTOP-BF2Q2B7,forensicrule,2068,00000229A0940000,C:\Windows\System32\svchost.exe,0x40

The output shows the host’s name, the rule name, the process ID, the memory region, the process’ executable path, and the memory protection for the region.

**Injected Thread Scanner**

This was initially the heart of ThreadDread; its ability to quickly located suspicious threads running throughout the system. As such, it is the simplest scanner to run; you simply supply no arguments! You will receive output such as the following:

ThreadScan DESKTOP-BF2Q2B7,6692, 2021-10-25 03:50:40 Zulu, 00007FF7E5084E80,0x4000,0x20, 6688,2021-10-25 03:50:40 Zulu,svchost.exe

In order from left-to-right, this is the flow of information:

1. The type of output
2. The originating host’s name
3. The Thread ID
4. The thread start time in Zulu
5. The thread start address
6. The type of region
7. The memory protection for that region
8. The process ID to which the thread belongs
9. The process start time
10. The name of the process

All this data helps to correlate a possible thread injection, giving the analyst the ability to quickly correlate the thread’s start time with any readily available SIEM alert times. In addition, the fact that the region’s memory protection is available can also help to identify a thread with an interesting protection applied to it, especially if that region is not part of a valid EXE or DLL mapped into the process’ memory space.

**Using ThreadDread With Response Toolkits**

Organizations may very well elect to incorporate ThreadDread as part of their response toolkits. I personally would recommend running the injected thread scanner as well as the injected binary scanner and redirect output to a text file for later analysis. That’s easy enough to do with the following commands:

.\ThreadDread.exe >> injected\_threads.txt

.\ThreadDread.exe -B >> injected\_binaries.txt

Analysts may (optionally) elect to run the Yara scanner with a custom rule set and capture the results of that scan to a file as well; to do that, do something like the following:

.\ThreadDread.exe -Y <rule\_file> >> yara\_scan.txt

Or specify a specific process to Yara scan:

.\ThreadDread.exe -Y <rule\_file> -p <process\_id> >> yara\_scan\_<process\_id>.txt

**Additional Resources**

Use the following MSDN links to learn more about the hexadecimal “constant” values that apply to memory regions:

<https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information>

<https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants>

To learn about writing Yara rules, visit the following link:

<https://yara.readthedocs.io/en/stable/writingrules.html>

If you want to understand more about memory forensics in general, please refer to the following book:

**Title:** The Art of Memory Forensics

**ISBN-13:** 978-1118825099

**ISBN-10:** 1118825098