Volatility Contest 2024 - Submission Document

Kartik N. Iyer & Parag H. Rughani

December - 2024

Table of Contents

1. Introduction **2**

2. Introducing the Thread Local Storage (TLS) callback analysis plugin **2**

2.1. Overview of TLS callbacks 3

2.2. Purpose and Objective of the Plugin 4

2.3. Core Features and Working of the Plugin **4**

2.3.1. Disassembly of TLS Callbacks 5

2.3.2. Extraction of Callbacks 5

2.3.3. Regex Based Instruction Matching 6

2.3.4. Byte Disassembly Limit Argument 7

2.3.5. Output Format 7

2.3.6. Specification of yara-rule parameter 8

3. Practical Tests **9**

3.1. Analysis of known clean memory sample 9

3.2. Use of Regex and Suspicious Scanning Features 10

4. Future Scope **11**

5. Why we should win the 2024 Volatility Plugin Contest? **12**

Acknowledgements **12**

References **12**

List of Figures

Figure 1: Arguments provided by the TlsCheck plugin 4

Figure 2: Representation of the TlsCheck output format 8

Figure 3: Working of the yara rule parameter shows detection capabilities. 9

Figure 4: Execution of the plugin on a benign windows 10 memory sample 10

Figure 5: Correlation between the disassembly of Malcat and TlsCheck 10

Figure 6: No suspicious indicators detected 11

Figure 7: Indication of a potentially suspicious instruction 11

Abstract

The research presented in this document is intended as a unique submission for the contest. In line with the contest guidelines, it includes a section titled "*Why should we win the 2024 Volatility Contest?"* All code and resources are contributed under the Volatility Foundation, Inc. Individual Contributor Licensing Agreement.

1. Introduction

Memory forensics plays a critical role in digital investigations, offering a unique perspective into the volatile memory of a system. By analyzing a memory snapshot, often referred to as a memory dump, investigators can uncover data and runtime artefacts that are not stored on disk—such as active network connections, running processes, and encryption keys. This requires a deep understanding of system internals, including hardware interactions, kernel structures, and software behaviour.

The Volatility framework, developed by the Volatility Foundation [4] has been a cornerstone in memory forensics since its inception in 2007. Initially released as Volatility 2, the tool underwent a major overhaul in 2020, leading to the creation of Volatility 3. This modernized framework has become a vital resource for researchers and practitioners alike, enabling comprehensive memory analysis across diverse platforms.

Each year, the Volatility Foundation hosts a contest that encourages the development of innovative plugins and extensions for the framework. This initiative not only sparks creativity in the memory forensics community but also pushes the boundaries of what can be achieved with Volatility.

This document presents our contribution to this year's contest: a specialized plugin designed to detect, analyze and disassemble TLS (Thread Local Storage) callbacks in memory samples. The plugin offers features such as regex-based instruction matching, suspicious instruction detection, yara-rule signature matching and customizable disassembly options, making it a powerful tool for detecting suspicious behaviours in malware and advanced threats. This project reflects our passion for cybersecurity and our commitment to advancing the field of memory forensics.

2. Introducing the Thread Local Storage (TLS) callback analysis plugin

Abstract

TLS callbacks, often overlooked in memory analysis, can serve as a gateway for both legitimate operations and malicious activities within executables. This section introduces a dedicated plugin designed to detect, analyze and disassemble TLS callbacks, helping investigators uncover hidden behaviours that might go unnoticed. The plugin offers features like instruction disassembly, regex-based matching, suspicious instruction detection and yara-based signature matching,

making it an essential tool for memory forensics. By focusing on these often-misused entry points, the plugin bridges a critical gap in current forensic capabilities and provides a deeper layer of analysis for researchers and practitioners.

During my academic studies in malware analysis and memory forensics, I first encountered TLS (Thread Local Storage) callbacks and was immediately intrigued by its unique functionality. I brought up this concept to my professor and had an in-depth discussion on it. That's where we made the decision to create **TlsCheck**, our volatility plugin.

An important point to note is that these callbacks execute even before the main program thread starts, meaning they run prior to the actual AddressOfEntryPoint of an executable [2]. This early execution capability makes TLS callbacks valuable for legitimate purposes, but it also renders them an attractive target for exploitation by malware authors.

Originally designed by Microsoft to facilitate thread-specific initialization, TLS callbacks are used for tasks like setting up thread-local storage or preparing resources for multi-threaded applications. However, this legitimate feature has been repeatedly misused by malware creators seeking to execute code stealthily, bypassing conventional detection mechanisms.

A notable example of this misuse is the Ursnif malware [1], which leveraged TLS callbacks to perform process injection directly into memory. By employing the TLS Anti-Analysis Evasion Trick, Ursnif was able to execute its malicious code covertly, evading various security and analysis tools. This dual role of TLS callbacks—both as a legitimate functionality and a potential vector for malicious activity—highlights their significance in research and practical forensic analysis.

2.1. Overview of TLS callbacks

In computer programming, thread-local storage (TLS) is a memory management technique that allocates static or global memory unique to each thread. This approach enables threads to store data that behaves like global memory but remains isolated within the context of each individual thread [3].

TLS callbacks are specialized functions in a program that are automatically executed by the operating system during specific stages of a thread's lifecycle. These callbacks are invoked when a thread is created, terminated, or when a process is loaded or unloaded. Their primary purpose is to allow developers to perform custom initialization or clean-up tasks related to thread-specific resources.

These callbacks are commonly used in scenarios where certain data or functionality needs to be prepared for each thread, such as initializing thread-local variables, configuring logging systems, or managing security settings. They are implemented in the '**.tls**' section of an executable file and are registered in the program's header. Although TLS callbacks can be powerful, they should be used cautiously, as improper use can lead to performance issues or bugs, particularly in multithreaded applications.

The TLS callback injection is a type of injection attack technique similar to the other known techniques. The MITRE ATT&CK framework [5] identifies TLS callback

injection as a sub-technique of process injection, listed under ID: T1055.005, to execute malicious code within the address space of a legitimate process. By manipulating pointers in the Portable Executable (PE) structure, attackers can redirect program execution to their malicious payload before reaching the program's legitimate entry point.

2.2. Purpose and Objective of the Plugin

The primary purpose behind creating this plugin (TlsCheck) is to detect and display TLS (Thread Local Storage) callbacks for the processes in memory. This approach is novel because, to date, no other plugin in the Volatility framework specifically targets the disassembly and analysis of TLS callbacks. As mentioned in previous sections, despite their legitimate uses, TLS callbacks have been increasingly exploited (or can be exploited) by malware authors for stealthy process injections and anti-analysis techniques. The plugin aims to bridge this gap by providing a dedicated tool to investigate TLS callbacks in memory dumps, enabling researchers and analysts to uncover potential misuse. This focus on a niche but critical area of malware behaviour makes it a valuable addition to the field of memory forensics.

The key objective of this plugin is to provide users with an easy-to-use interface for disassembling TLS callback instructions, detecting suspicious patterns using regex-based matching, and highlighting potentially suspicious behaviours. By empowering analysts to identify and investigate these subtle attack vectors, the plugin seeks to enhance forensic capabilities and address a previously underexplored area of memory analysis.

```
kali@kali:~/Tools/volatility3$ python3 vol.py -f  ../../TLS/NOP.vmem windows.TlsCheck --help
Volatility 3 Framework 2.11.0
usage: volatility windows.TlsCheck.TLSCheck [-h] [--pid [PID ...]] [--disasm-bytes DISASM-BYTES] [--scan-suspicious] [--regex REGEX] [--yara-file YARA-FILE]

options:
  -h, --help            show this help message and exit
  --pid [PID ...]       Process IDs to include (all other processes are excluded)
  --disasm-bytes DISASM-BYTES
                        Bytes to disassemble (Default: 64)
  --scan-suspicious     Displays suspicious TLS Callback instruction(s) along with the disassembly
  --regex REGEX         Custom regex pattern to match against disassembled instructions
  --yara-file YARA-FILE
                        Path to custom YARA rule file
```

Figure 1: Arguments provided by the TlsCheck plugin

2.3. Core Features and Working of the Plugin

The **TlsCheck** plugin is specifically designed for the Windows operating system and is fully compatible with the latest version of Volatility 3. This tool focuses on detecting, analysing and disassembling TLS callbacks for processes present in memory.

TlsCheck offers a clean and user-friendly output, presenting detailed information and incorporating practical features to streamline forensic workflows. By disassembling TLS callbacks and detecting potentially suspicious patterns, the plugin empowers investigators to identify subtle threats effectively. This section dives into the core features of TlsCheck (see Figure 1 for provided argument options), highlighting the

plugin's disassembling feature, extraction of callbacks, regex-based suspicious instruction detection, byte disassembly based on user specifications, format of output and yara based signature matching.

2.3.1. Disassembly of TLS Callbacks

Disassembling is the process of breaking down machine code into human-readable assembly language, allowing analysts to understand what a program or code is doing at a low level. Disassembling these callbacks from memory is crucial for understanding their true purpose and identifying potentially suspicious patterns during memory analysis.

The TlsCheck plugin handles TLS callback extraction differently for 32-bit and 64-bit executables, though both approaches follow similar principles. For 32-bit binaries, it locates the TLS directory, reads the AddressOfCallBacks field, and calculates the correct file offset by converting the RVA using the PE file's section information. For 64-bit binaries, it follows a similar process but accounts for the different address sizes and PE format specifications of 64-bit executables. Once the callback locations are identified in either case, the plugin uses the Capstone disassembly engine to decode the instructions at those addresses, displaying them in a readable format with proper memory addresses. It lists the names of processes with a TLS RVA greater than 0. If the callback procedure disassembly is identified, the plugin proceeds to disassemble the code for that process. Normally, the plugin stops disassembling at the *ret* instruction, but this limit can be extended with the *--disasm-bytes* (discussed in section 2.3.4) argument to show instructions beyond the 'ret' instruction. The extracted callbacks are then analyzed for suspicious instruction patterns that might indicate doubtful behaviour.

2.3.2. Extraction of Callbacks

The plugin begins its execution by first dumping out the process from the memory and then parsing the PE file of each process to locate the TLS Directory, which is part of the PE header. The directory contains a pointer to the AddressOfCallBacks, a list of function pointers that define the TLS callbacks. The relevant section of the code that identifies the TLS Directory is as follows:

```
# Access Data Directory from Optional Header
data_directory = pe_obj.OPTIONAL_HEADER.DATA_DIRECTORY

# Check for TLS Directory (10th entry)
tls_directory_entry = data_directory[9]  # TLS Directory entry is at index 9
tls_rva = tls_directory_entry.VirtualAddress

'''redacted '''
```

If the TLS Directory RVA (Relative Virtual Address) is 0, it indicates that no TLS callbacks are defined for the process, and the plugin skips further processing for that process. Once the AddressOfCallbacks is identified, its RVA is converted to a file offset using the following function defined:

```
tls_offset = self.rva_to_file_offset(pe, tls_rva)
```

The *rva_to_file_offset* function ensures accurate mapping between the RVA and the corresponding location in the file. This step is crucial for accessing the actual memory content of the TLS callbacks. After determining the file offset, the plugin reads the callback addresses stored in memory:

```
with open(exe_file, "rb") as f:
    f.seek(tls_offset + 12)       # Move to the location of AddressOfCallbacks
    address_bytes = f.read(4)  # Read 4 bytes
    last_address = int.from_bytes(address_bytes, "little")
```

Here, the plugin extracts and validates the callback address to ensure it points to executable code within the process's memory space. If a valid callback is found, its disassembly begins. This is especially useful for detecting potentially malicious patterns, such as unusual jumps or function calls. For example:

```
self.disassemble_bytes_at_address32 (
    exe_file,
    first_4_bytes_value,
    image_base,
    proc_name
)

self.disassemble_bytes_at_address64 (
     exe_file_64,
     callback_address,
     image_base,
     proc_name,
     callback_rva
   )
```

The *disassemble_bytes_at_address32* and *disassemble_bytes_at_address64* functions are responsible for interpreting the machine code at the specified address and presenting the instructions in a human-readable format.

2.3.3. Regex Based Instruction Matching

The regex-based instruction matching feature in the TlsCheck plugin is designed to enhance the analysis of TLS callbacks by identifying patterns of interest in disassembled instructions. The plugin includes a predefined list of regex patterns that target common suspicious behaviours, such as:

- Indirect jumps
- Privileged instructions
- Anti-analysis techniques
- Stack pivoting and manipulation
- Common obfuscation techniques
- Anti-Debugging techniques and more.

Users can expand the list by adding their own patterns, which will be automatically included whenever the *--scan-suspicious* argument is used. In support and addition to the pre-built patterns, the plugin offers two arguments for users.

- **--scan-suspicious:** By enabling this flag, users can scan disassembled instructions for matches against the predefined suspicious patterns mentioned above.
- **--regex:** Alternatively, users can define their own custom regex using the argument to search for specific patterns or strings that align with their unique analysis requirements.

2.3.4. Byte Disassembly Limit Argument

Although not directly related, we've noticed that the malfind [6] plugin disassembles and displays only a fixed number of assembly bytes to users. If more disassembly is needed (especially in the case of Header Stomping attacks), users have to switch to volshell and write a simple Python script to view additional bytes. Keeping this limitation in mind, and since our plugin also focuses on disassembly, we've included the *--disasm-bytes [bytes]* argument in our plugin. By default, it disassembles and displays 64 bytes of code (stopping immediately if a 'ret' instruction is encountered – which indicates the completion of a TLS section), but users can easily adjust this value to meet their requirements.

2.3.5. Output Format

The output (see Figure 2) of the **TlsCheck** plugin is designed to be straightforward and user-friendly, providing critical information about processes and their associated TLS callbacks in memory. Here's an explanation of each column and its purpose:

1. PID (Process ID):
    - Displays the unique identifier for each process. This ID helps users identify and refer to specific processes in the memory dump.
2. PPID (Parent Process ID):
    - Shows the ID of the parent process that spawned the current process. This information is useful for understanding process relationships and detecting anomalies, such as suspicious parent-child connections.
3. Process Name:
    - Lists the name of the executable associated with each process. This provides an immediate reference to what the process is and its potential role in the system.
4. Offset(V):
    - Indicates the virtual memory offset of the process in the memory dump. This is valuable for further analysis or debugging.
5. TLS RVA(V):
    - Relative virtual address of the TLS
6. Architecture:
    - Specifies whether the process is running in a 32-bit (x86) or 64-bit (x64) architecture. This distinction is important for compatibility and disassembly.
7. Path:

- Displays the full path of the executable associated with the process, helping users locate the file on disk (if accessible) and verify its legitimacy.

8. TLS-Callback Instructions (Detailed Section):
    - For processes with TLS callbacks, this section disassembles and lists the instructions executed within the callback. Each line shows:
        - Hexdump: Displays the hexadecimal representation of the contents.
        - Address: The virtual memory address of the instruction (e.g., 0x7ff75dc01700).
        - Instruction: The operation being performed (e.g., cmp, mov, push).
        - Operands: The data or registers involved in the operation (e.g., edx, 2 or qword ptr [rsp + 8]).

```
kali@kali:~/Tools/volatility3$ python3 vol.py -f  ../../TLS/windowstls2.vmem windows.TlsCheck --pid 7124
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     PPID    Process Name    Offset(V)       TLS RVA(V)      Architecture    Path

7124    5088    TLSCallbackExa  0xd90d6db8a080  0x2600  x64     C:\Users\FlareVM\Desktop\TLSCallbackExample.exe

----------------------------------------------------------------------
TLS-Callback Found in Process: TLSCallbackExa (PID: 7124)
Address range: 0x7ff7523c1070 - 0x7ff7523c10b0
----------------------------------------------------------------------
83 fa 01 75 1b 44 8d 4a 3f 33 c9 48 8d 15 2e 12 ...u.D.J?3.H....
00 00 4c 8d 05 07 12 00 00 48 ff 25 f0 0f 00 00 ..L......H.%....
c3 cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc ................
83 fa 01 75 1b 44 8d 4a 3f 33 c9 48 8d 15 7e 12 ...u.D.J?3.H..~.
Disassembly:
0x7ff7523c1070: cmp     edx, 1
0x7ff7523c1073: jne     0x1090
0x7ff7523c1075: lea     r9d, [rdx + 0x3f]
0x7ff7523c1079: xor     ecx, ecx
0x7ff7523c107b: lea     rdx, [rip + 0x122e]
0x7ff7523c1082: lea     r8, [rip + 0x1207]
0x7ff7523c1089: jmp     qword ptr [rip + 0xff0]
0x7ff7523c1090: ret
----------------------------------------------------------------------
```

Figure 2: Representation of the TlsCheck output format

2.3.6. Specification of yara-file parameter

The plugin supports advanced detection capabilities through custom YARA rules using the *--yara-file* argument. One can write their own custom YARA rules in a file and pass its path to the plugin to scan the TLS callback instructions. This is particularly useful for identifying specific patterns, malicious behaviours, or code sequences of interest within TLS callbacks. For example, one could create rules to detect specific shellcode patterns, API abuse, or known malicious code structures. The plugin will evaluate the disassembled TLS callback instructions against the specified YARA rules and highlight any matches found. When a match is found, the plugin displays the name of the matched rule, helping analysts' quickly pinpoint suspicious code patterns within the TLS callbacks.

```
kali@kali:~/Tools/volatility3$ python3 vol.py -f  ../../TLS/NOP.vmem windows.TlsCheck --yara-file ~/Desktop/NOP.yar --pid 4044
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     PPID    Process Name    Offset(V)       TLS RVA(V)      Architecture    Path

4044    3324    nopAssemblyTLS  0xb203764f2080  0x2500  x64     C:\Users\FlareVM\Desktop\nopAssemblyTLS.exe

----------------------------------------------------------------------
TLS-Callback Found in Process: nopAssemblyTLS (PID: 4044)
Address range: 0x7ff6e6561070 - 0x7ff6e6561093
----------------------------------------------------------------------
83 fa 01 75 28 90 44 8d 4a 3f 90 4c 8d 05 fe 11 ...u(.D.J?.L....
00 00 90 48 8d 15 16 12 00 00 90 33 c9 90 90 90 ...H.......3....
90 90 90                                        ...
Disassembly:
0x7ff6e6561070: cmp     edx, 1
0x7ff6e6561073: jne     0x109d
0x7ff6e6561075: nop
0x7ff6e6561076: lea     r9d, [rdx + 0x3f]
0x7ff6e656107a: nop
0x7ff6e656107b: lea     r8, [rip + 0x11fe]
0x7ff6e6561082: nop
0x7ff6e6561083: lea     rdx, [rip + 0x1216]
0x7ff6e656108a: nop
0x7ff6e656108b: xor     ecx, ecx
0x7ff6e656108d: nop
0x7ff6e656108e: nop
0x7ff6e656108f: nop
0x7ff6e6561090: nop
0x7ff6e6561091: nop
0x7ff6e6561092: nop
----------------------------------------------------------------------
[*] YARA Rule Matches:
[YARA Match] Rule: NOP_Instructions
----------------------------------------------------------------------
```

Figure 3: Working of the yara-file parameter shows detection capabilities.

3. Practical Tests

To showcase the functionality of the plugin, we ran practical tests on windows 10 memory samples. These tests demonstrate the plugin's ability to detect and analyze legitimate processes, detect malware using TLS callbacks, and handle various use cases like suspicious instruction detection and customizable disassembly.

3.1. Analysis of known clean memory sample

A clean Windows 10 memory sample was created to test the plugin's functionality. As shown in the output (Figure 4), the plugin lists processes where the Relative Virtual Address (RVA) of the TLS is non-zero. Processes with a zero RVA are automatically filtered out and excluded from the output. Note that in Figure 4, the process with PID 2640 does not contain a 'ret' instruction within the 64-byte disassembly limit. As a result, no 'ret' instruction is displayed in the disassembly

```
└─$ python3 vol.py -f  ../../TLS/windowstls.vmem windows.TlsCheck
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     PPID    Process Name    Offset(V)       TLS RVA(V)      Architecture    Path

684     572     winlogon.exe    0x9d0c3d332080  0x9ca90 x64     C:\Windows\system32\winlogon.exe
508     684     dwm.exe 0x9d0c3d2e8080  0xe2c8  x64     C:\Windows\system32\dwm.exe
2024    736     spoolsv.exe     0x9d0c3e7b30c0  0x96468 x64     C:\Windows\System32\spoolsv.exe
2640    736     OfficeClickToR  0x9d0c3eaed080  0xaeee68        x64     C:\Program Files\Common

----------------------------------------------------------------------
TLS-Callback Found in Process: OfficeClickToR (PID: 2640)
Address range: 0x7ff7d35c1700 - 0x7ff7d35c1740
----------------------------------------------------------------------
83 fa 02 75 60 48 89 5c 24 08 57 48 83 ec 20 8b ...u`H.\$.WH.. .
0d 33 3a 8e 00 65 48 8b 04 25 58 00 00 00 41 b8 .3:..eH..%X...A.
64 00 00 00 48 8b 14 c8 42 80 3c 02 01 74 2c 42 d...H...B.<..t,B
c6 04 02 01 48 8d 1d 35 7b 50 00 48 8d 3d 66 7b ....H..5{P.H.=f{
Disassembly:
0x7ff7d35c1700: cmp     edx, 2
0x7ff7d35c1703: jne     0x371765
0x7ff7d35c1705: mov     qword ptr [rsp + 8], rbx
0x7ff7d35c170a: push    rdi
0x7ff7d35c170b: sub     rsp, 0x20
0x7ff7d35c170f: mov     ecx, dword ptr [rip + 0x8e3a33]
0x7ff7d35c1715: mov     rax, qword ptr gs:[0x58]
0x7ff7d35c171e: mov     r8d, 0x64
0x7ff7d35c1724: mov     rdx, qword ptr [rax + rcx*8]
0x7ff7d35c1728: cmp     byte ptr [rdx + r8], 1
0x7ff7d35c172d: je      0x37175b
0x7ff7d35c172f: mov     byte ptr [rdx + r8], 1
0x7ff7d35c1734: lea     rbx, [rip + 0x507b35]
----------------------------------------------------------------------
```

Figure 4: Execution of the plugin on a benign windows 10 memory sample

To verify the disassembly performed by **TlsCheck**, the process dump was analyzed using the Malcat tool to cross-check the TLS callback disassembly. The results confirmed that the plugin's disassembly was accurate, with no faults in the instruction disassembly, demonstrating that this feature is functioning effectively (see Figure 5).

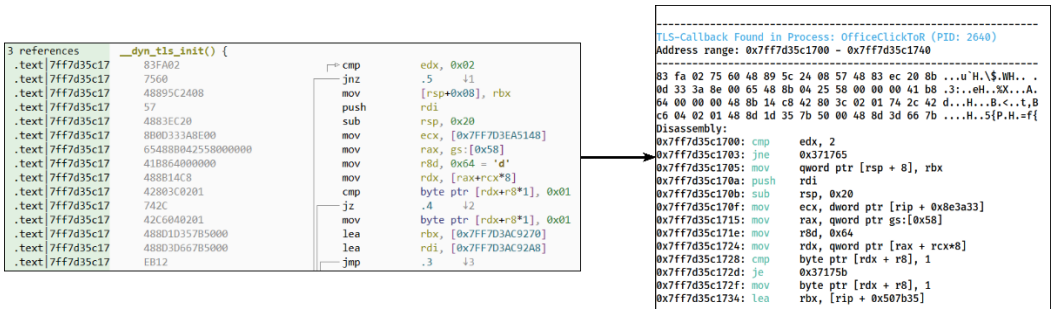

Figure 5: Correlation between the disassembly of Malcat and TlsCheck

3.2. Use of Regex and Suspicious Scanning Features

The **--scan-suspicious** option of **TlsCheck** was tested on a Windows 10 memory sample. As shown in Figure 6, if no suspicious instructions are detected, the plugin simply returns nothing. However, if it finds something suspicious, it lists the specific instruction for further review (see Figure 7). It's important to note that these flagged instructions are only marked as *suspicious*, not necessarily *malicious*. Additional investigation is always required to verify any findings.

```
┌──(kali㉿kali)-[~/Tools/volatility3]
└─$ python3 vol.py -f  ../../TLS/TLS.vmem windows.TlsCheck --scan-suspicious --pid 4268
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     PPID    Process Name    Offset(V)       TLS RVA(V)      Architecture    Path

4268    1552    TLS.exe 0x8c85a4a77080  0x2180  x86     C:\Users\Flare\Desktop\Samples\Legit\TLS.exe

----------------------------------------------------------------
TLS-Callback Found in Process: TLS.exe (PID: 4268)
Address range: 0x401000 - 0x401040
----------------------------------------------------------------
64 a1 2c 00 00 00 6a 00 68 10 21 40 00 68 10 21 d.,...j.h.!@.h.!
40 00 8b 00 6a 00 c7 40 04 1f 02 00 00 ff 15 98 @...j..@........
20 40 00 c2 0c 00 cc cc cc cc cc cc cc cc cc cc  @..............
64 a1 2c 00 00 00 8b 00 ff 70 04 68 20 21 40 00 d.,......p.h !@.
Disassembly:
0x401000:       mov     eax, dword ptr fs:[0x2c]
0x401006:       push    0
0x401008:       push    0x402110
0x40100d:       push    0x402110
0x401012:       mov     eax, dword ptr [eax]
0x401014:       push    0
0x401016:       mov     dword ptr [eax + 4], 0x21f
0x40101d:       call    dword ptr [0x402098]
0x401023:       ret     0xc
----------------------------------------------------------------
```

Figure 6: No suspicious indicators detected

```
└─$ python3 vol.py -f  ../../TLS/windowstls3.vmem windows.TlsCheck --scan-suspicious --pid 4004 --disasm-bytes 36
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     PPID    Process Name    Offset(V)       TLS RVA(V)      Architecture    Path

4004    5020    AssemblyWhat.e  0xa6846d181080  0x2500  x64     C:\Users\FlareVM\Desktop\AssemblyWhat.exe

----------------------------------------------------------------
TLS-Callback Found in Process: AssemblyWhat.e (PID: 4004)
Address range: 0x7ff70bb81070 - 0x7ff70bb81094
----------------------------------------------------------------
83 fa 01 75 2a 90 4c 8d 05 03 12 00 00 90 48 8d ...u*.L.......H.
15 1b 12 00 00 90 41 b9 40 00 00 00 90 33 c9 90 ......A.@....3..
90 90 90 90                                     ....
Disassembly:
0x7ff70bb81070: cmp     edx, 1
0x7ff70bb81073: jne     0x109f
0x7ff70bb81075: nop
0x7ff70bb81076: lea     r8, [rip + 0x1203]
0x7ff70bb8107d: nop
0x7ff70bb8107e: lea     rdx, [rip + 0x121b]
0x7ff70bb81085: nop
0x7ff70bb81086: mov     r9d, 0x40
0x7ff70bb8108c: nop
0x7ff70bb8108d: xor     ecx, ecx
0x7ff70bb8108f: nop
0x7ff70bb81090: nop
0x7ff70bb81091: nop
0x7ff70bb81092: nop
0x7ff70bb81093: nop
----------------------------------------------------------------
[*] Potentially Suspicious Instruction(s) Identified:

[D] Suspicious: NOP sled detected starting at nop
----------------------------------------------------------------
```

Figure 7: Indication of a potentially suspicious instruction

Tip: Try increasing the bytes to scan more instructions for suspicion (use --disasm-bytes option).

4. Future Scope

While TlsCheck already offers powerful tools like regex-based instruction matching and YARA rule integration, there's plenty of potential to make these features even more effective. Expanding the library of built-in regex patterns to cover more complex and stealthy techniques would help uncover a broader range of suspicious behaviours. Similarly, enhancing YARA rule support by allowing updates or access to a shared

repository of community-created rules could make it easier to adapt to new and evolving threats. Adding options for users to fine-tune regex searches or manage YARA rules more intuitively would also make the plugin even more practical and user-friendly. These improvements would ensure TlsCheck stays ahead as a go-to tool for forensic investigators.

5. Why we should win the 2024 Volatility Plugin Contest?

At its core, memory forensics is about uncovering hidden stories within the volatile layers of a system, and we've always been passionate about this fascinating field. It's a blend of technical depth and the detective work of piecing together clues. For us, creating the TlsCheck plugin wasn't just a technical challenge—it was a mission to address a gap in memory analysis tools and make a meaningful contribution to the forensics community.

Over the past few months, we have been designing **TlsCheck**. It started with a simple observation: TLS callbacks, while powerful, are often overlooked in memory analysis, yet they hold immense potential for both legitimate use and exploitation by attackers. Recognizing this, we set out to create a plugin that not only detects these callbacks but dives deep into their workings to help analysts identify patterns that could signal malicious activity.

What makes TlsCheck special is its focus on usability and practicality. We didn't want to just create another tool; we wanted to build something analysts would actually rely on. With features like regex-based instruction matching, YARA rule support, and user-friendly outputs, we aimed to simplify and enhance the investigative process. The results speak for themselves—the plugin has successfully detected and analyzed TLS callbacks, flagged suspicious instructions, and even uncovered nuanced behaviours that traditional tools often miss.

Winning this contest would be an incredible validation of our hard work and vision. But more than that, it would shine a spotlight on the importance of addressing overlooked attack vectors like TLS callbacks. It would inspire us to continue pushing boundaries and contribute to the ever-evolving world of digital forensics.

We believe TlsCheck bridges a critical gap in memory forensics, and we're excited to share it with the community. Whether or not we win, we're proud of the work we've done—and we hope it will make a difference for analysts and researchers everywhere.

Acknowledgements

We would like to thank Harsh Upadhyay, for helping us creating small programs and memory dumps to test out our plugin. His assistance was practical and made the testing process much more manageable. We appreciate his support during this project.

References

[1] **Balaji** Ursnif Malware Variant Performs Malicious Process Injection in Memory using TLS Anti-Analysis Evasion Trick [Online] // gbhackers. - 29 November 2017. - 2024. - https://gbhackers.com/ursnif-malware-variant/.

[2] **Bocchetti Andrea** TLS Callbacks to bypass debuggers [Online] // Medium. - 31 January 2024. - https://medium.com/@andreabocchetti88/tls-callbacks-to-bypass-debuggers-60409195ed76.

[3] **Thread-local storage [Online] // Wikipedia**. - 21 October 2024. - December 2024. - https://en.wikipedia.org/wiki/Thread-local_storage.

[4] **Volatility** The Volatility Foundation [Online] // Volatility Foundation. - December 2024. - https://volatilityfoundation.org/.

[5] **Process injection:** Thread Local Storage, Sub-technique T1055.005 - Enterprise | MITRE ATT&CK®. (n.d.). https://attack.mitre.org/techniques/T1055/005/

[6] **Volatility3.Plugins.Windows.Malfind Module - volatility 3.2.11.0 Documentation,** n.d. https://volatility3.readthedocs.io/en/latest/volatility3.plugins.windows.malfind.html