Forensic Science International: Digital Investigation 56 (2026) 302060

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2026 - Selected Papers from the 13th Annual Digital Forensics Research Conference Europe

Structural analysis of the Windows NT heap for memory forensics

Daniel Uroz [a,b], Abraham Díaz-Campo Pinilla [1], Ricardo J. Rodríguez [a,*]

[a] Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[b] IndraMind Cybersecurity, Spain

ARTICLE INFO

Keywords:
Memory forensics
Windows NT heap
Volatility 3
Heap forensics
Low fragmentation heap

ABSTRACT

Modern attacks increasingly target user-space memory, leveraging dynamic heap allocations to store payloads, obfuscate runtime behavior, and evade traditional detection mechanisms. These heap-based techniques complicate memory forensics, as existing tools typically treat dynamic memory as a flat, unstructured region. To address this gap, in this paper we present a forensic methodology for the extraction and structural analysis of Windows NT heap entries, implemented in an open-source plugin for the Volatility 3 framework, called HeapList. Our approach supports all major Windows versions, from Vista to Windows 11, on both ×86 and ×64 architectures. We reconstruct the backend and frontend heap layers, decode encoded metadata, and enable navigation and directed extraction of heap entries. We validate our methodology through cross-verification with `WinDbg` and controlled testing using the Windows Heap API. Additionally, we discuss how our plugin can facilitate reverse engineering, the identification of dynamic payloads, heap layout inspection, and memory triage. By providing structured access to user-space heap memory, our work improves forensic visibility into dynamic memory and enables deeper analysis of heap-centric behavior in modern threat landscapes. Finally, we demonstrate the applicability of our approach in real-world scenarios by extracting information relevant to forensic analysis of user-space applications (specifically, from Telegram Desktop) through heap analysis.

1. Introduction

Digital forensics has evolved from the analysis of static artifacts on disk to the capture and interpretation of volatile memory snapshots to uncover traces of malicious activity (Mitropoulos et al., 2006; Ligh et al., 2014). One of the richest, yet underexplored, areas within memory forensics is the (user-space) heap (Block and Dewald, 2017), a dynamic memory region allocated by processes at runtime (Bradley, 2011). While kernel-mode analysis has traditionally been the focus of memory forensics, modern threats are increasingly exploiting user-space memory, using it as a platform for payloads, evasion tactics, and command and control beacons (MITRE, 2025; Repel et al., 2017).

This shift is particularly critical in the Windows operating system, which remains the most widely deployed desktop platform and, therefore, a prime target for adversaries (CrowdStrike, 2024; GlobalStats, 2025). Its complexity, widespread adoption in enterprise and consumer environments, and legacy architectural features make it especially attractive to attackers seeking persistence and stealth. In Windows, the heap plays a fundamental role in dynamic memory management, and its misuse has been a frequent vector for vulnerabilities such as use-after-free, heap spraying, and type confusion (Ding et al., 2010; Bouffard et al., 2015; Haller et al., 2016; Gui et al., 2022; Du et al., 2024). However, the internal structure of the Windows heap, especially that of NT (Yosifovich et al., 2017), remains largely opaque and inconsistently documented, limiting the forensics community's ability to rigorously analyze it.

Despite the increasing sophistication of attacks targeting dynamic memory, current forensic tools offer limited visibility into heap structures (Cohen, 2015; Block and Dewald, 2017; Yun et al., 2020). Many rely on coarse-grained techniques, such as memory carving or signature comparison, which are typically ineffective against polymorphic or novel threats. Furthermore, while reverse engineering initiatives have shed light on the internals of the Windows heap (Yason, 2016; Valasek and Mandt, 2012), these insights have not yet been fully translated into accessible and practical tools for incident response and memory analysis. As a result, security analysts are often forced to treat the heap as an opaque or flat region, lacking the granularity required to trace attacker behavior, reconstruct heap manipulations, or attribute artifacts to

* Corresponding author.
E-mail addresses: duroz@minsait.com (D. Uroz), abraham.dcp@gmail.com (A. Díaz-Campo Pinilla), rjrodriguez@unizar.es (R.J. Rodríguez).
[1] Independent researcher.

https://doi.org/10.1016/j.fsidi.2026.302060

specific heap operations. This gap between threat complexity and forensic capabilities represents a significant obstacle to detecting and responding to heap-centric attacks.

To address this gap, we present a forensic approach for the structurally accurate extraction and analysis of Windows NT heap entries from user-space memory dumps. Our contributions are embodied in an open-source plugin, HeapList,[2] for the Volatility 3 framework. The plugin systematically traverses the backend and frontend layers of the heap, correctly handles hard-coded attributes, distinguishes committed from uncommitted memory, and is compatible with a wide range of Windows versions, from Vista to Windows 11, on both major architectures. Our methodology allows analysts to interpret heap structures in user space beyond byte-level searches, facilitating deeper memory investigations and opening new avenues for behavior-based memory threat detection.

By focusing on the NT heap (still prevalent in most processes that do not adopt the segment heap (Yason, 2016)), we offer the forensics community a detailed roadmap for understanding the inner workings of the Windows heap, visualizing allocation patterns, and detecting potential anomalous or malicious memory usage in incident response scenarios.

1.1. Contributions

In summary, the contributions of this paper are four-fold:

(i) We present a forensic methodology for structurally traversing the Windows NT heap in user-space memory, enabling accurate reconstruction of both backend and frontend heap entries in memory dumps;
(ii) We implement this methodology as an open-source plugin, HeapList, for the Volatility 3 framework, supporting all major Windows versions, from Vista to Windows 11, on both ×86 and ×64 architectures;
(iii) We validate our approach through cross-verification with `WinDbg` and controlled experiments using the Windows Heap API, and discuss its practical utility in advanced threat detection, heap exploitation analysis, reverse engineering, and forensic triage; and
(iv) We demonstrate the ability of heap analysis to recover forensic-relevant artifacts from real-world applications, illustrated through a case study on Telegram Desktop.

1.2. Outline

The remainder of this paper is organized as follows. Section 2 provides necessary background on memory management in Windows. Section 3 delves into the inner workings of the Windows NT heap, focusing on the backend and frontend layers, and details the algorithms we used to analyze these structures. Section 4 presents our implementation and evaluation of the HeapList plugin, including its support for Windows versions and its validation against benchmark data. Section 5 presents a case study demonstrating the extraction of relevant forensic artifacts from Telegram Desktop through heap analysis. Section 6 explores the forensic implications of user-space heap analysis, highlighting both its potential applications in threat detection and its inherent limitations. Section 7 discusses related work in heap forensics and memory analysis, differentiating our approach from previous research. Finally, Section 8 concludes the paper and describes future directions of work.

2. Background

In this section, we first provide an overview of how memory is managed in the Windows operating system (Windows, for short), highlighting the role of user memory and the various APIs exposed for memory allocation and deallocation. Next, we focus on the Windows heap manager, responsible for efficiently managing dynamic memory requests at a granularity finer than the page level. Understanding these fundamentals is necessary for analyzing the internal structure of the NT heap, which underpins our forensic approach to heap traversal and analysis. Let us remark that the technical details presented in this section are adapted from the seminal work of Yosifovich et al. (2017).

2.1. Windows memory overview

Memory management in Windows is based on fixed-size units called *pages*, with a minimum size of 4 KB on all supported architectures (x86, x64, and ARM). This page-based design means that the page is the smallest unit of protection and allocation managed by Windows (Yosifovich et al., 2017).Fig. 1 provides an overview of the various memory management APIs that Windows exposes to user-space applications.

The foundation is the kernel-mode memory manager, which orchestrates physical and virtual memory. User-mode applications interact with this manager indirectly through several APIs. The *virtual API* enables low-level, page-granular memory operations (e.g., `VirtualAlloc`, `VirtualFree`, or `VirtualProtect`), while the *file mapping API* supports access to memory-mapped files and shared memory between processes (e.g, `CreateFileMapping`, `OpenFileMapping`, or `MapViewOfFile`). Above these is the *heap API*, which abstracts and manages memory for small, frequent allocations more efficiently (e.g., well-known examples are `HeapAlloc`, `HeapFree`, or `HeapCreate`). It is used internally by the *local/global API* (a legacy interface from 16-bit Windows) and by the *C/C++ runtime environment*, which translates standard memory functions such as `malloc` and `free` into heap API calls. This layered design allows user applications to manage memory flexibly while maintaining strict boundaries between user space and the kernel.

It is also worth noting that while the C/C++ language runtimes are not part of the Windows memory manager per se, their standard dynamic memory functions (such as `malloc`, `free`, and `realloc`) ultimately depend on the heap API, which improves portability and integration with the underlying operating system.

2.2. Heap manager

While the virtual API is designed for page-level allocations, using it directly for small allocations is inefficient. To optimize memory usage and performance, Windows introduced the heap API, which manages a region of memory (allocated through the virtual API) and subdivides it into smaller chunks, known as *heap entries*. These heap entries are aligned to 8 bytes on ×86 architectures and 16 bytes on ×64 architectures (Yosifovich et al., 2017).

Each process is automatically assigned a default heap at startup.

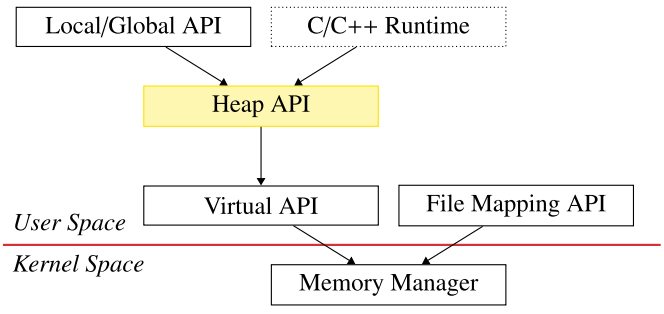

Fig. 1. Windows memory management APIs available to user-space applications (Yosifovich et al., 2017).

[2] Available at https://github.com/reverseame/heaplist.

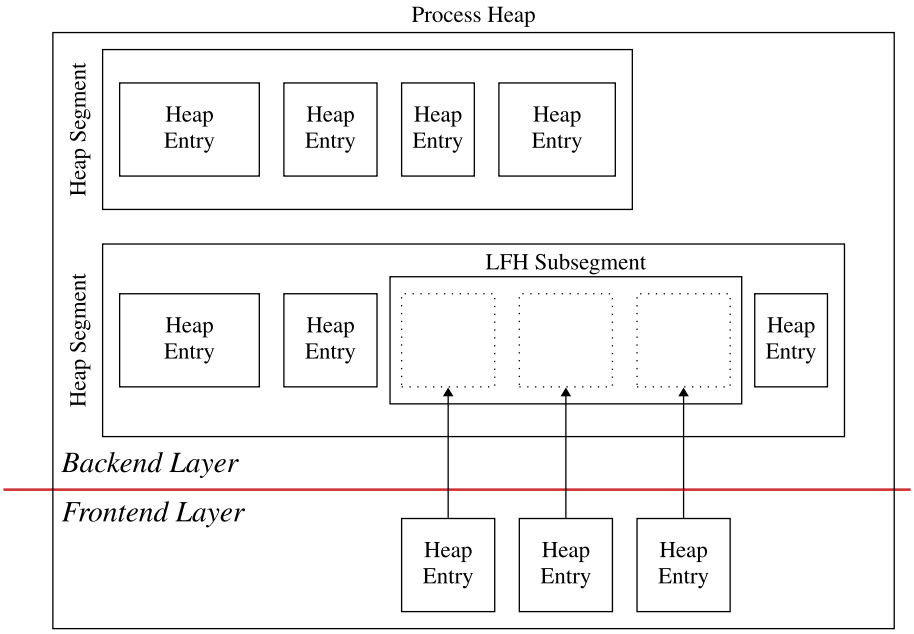

Fig. 2. Simplified Windows NT heap architecture. The backend layer manages segment allocation, while the frontend layer (LFH) organizes uniformly sized allocations within subsegments.

However, the heap API allows the creation and management of multiple heaps, providing flexibility in dynamic memory management by applications. Internally, the Windows memory management system supports two main heap implementations:

(1) The NT heap, the traditional heap mechanism used by most applications. (2) The Segment heap, a more recent design introduced in Windows 10, which is enabled by default for modern Windows applications and some system executables (Yason, 2016). In this paper, we focus on the NT heap, which remains the default and most widespread implementation for legacy and non-UWP processes (Yason, 2016; Yosifovich et al., 2017).[3]

The NT heap is internally divided into two functional layers: the *backend layer*, which manages fundamental memory regions and operations such as allocation, deallocation, and resizing of heap segments; and the *frontend layer*, which introduces optimization strategies for small and frequently requested allocations.

The only frontend implementation supported by modern versions of Windows is the *Low Fragmentation Heap* (LFH) (Yosifovich et al., 2017; Valasek, 2010). When LFH is active, the frontend manages subsegments within the memory provided by the backend, grouping allocations of the same size to reduce fragmentation and improve performance.Fig. 2 illustrates this layered architecture, showing how heap entries are organized into segments and subsegments.

3. Dissecting the windows NT heap structure

Although Microsoft does not publicly document the inner workings of the Windows heap, seminal work by Valasek (2010); Valasek and Mandt (2012) and subsequent research by Momot (2013) has reverse-engineered key aspects of its structure and behavior. Building on these efforts, our work focuses on a practical, forensic analysis of heap structures on modern Windows systems. Rather than treating the heap as a flat, contiguous memory region, our methodology allows analysts to explore its internal structures and extract meaningful, well-defined data.

As mentioned above, we focus on the NT heap, which remains the predominant heap implementation for non-UWP processes (Yason, 2016; Yosifovich et al., 2017) and is supported by all major releases from Windows Vista to Windows 11, with no substantial architectural changes since Windows 8.

In what follows, we detail the two layers that make up the NT heap (see Section 2.2). First, we describe the backend layer and present the algorithm implemented to reliably extract its contents. Next, we examine the frontend layer and explain the decoding and traversal logic required to retrieve its entries.

3.1. Traversing the backend layer

Algorithm 1 describes the traversal logic of the backend layer of the Windows NT heap, which manages memory segments and heap entries, and is applicable to systems from Windows Vista onwards. Each process has at least one heap, accessible via the `ProcessHeaps`array in the Process Environment Block (PEB) (Microsoft Corporation, 2022), an internal data structure used by Windows to maintain process-level information. The PEB resides in user space and is created and initialized by the Windows loader when a process starts. Each heap corresponds to an `ntdll!_HEAP` structure, which contains a list of memory segments represented by the `ntdll!_HEAP_SEGMENT` symbol. These are organized in a doubly linked list under the `SegmentList` attribute.

Each segment consists of several heap entries (`ntdll!_HEAP_ENTRY`), which include metadata such as size and flags, followed by application data. The granularity of the heap entries depends on the architecture: 8 bytes on ×86 systems and 16 bytes on ×64 systems, as indicated in Section 2.2. The `Flags` field indicates the status of the entry: free (`0x0`), busy (`0x01`), or internally managed (`0x08`), along with optional user-defined flags (`0x20`, `0x40`, or `0x80`). Although the exact semantics of the internal flag are undocumented,[4] our analysis suggests that it correlates with entries managed by LFH integrated into

[3] The Universal Windows Platform (UWP) is a Microsoft application development framework introduced with Windows 10. A non-UWP process refers to a Windows application that is not built using the UWP framework.

[4] See Raymond Chen's discussion: https://stackoverflow.com/questions/78900630/analyzing-a-crash-dump-on-windows-most-of-the-memory-is-busy-internal-what-i#comment139141936_78906881 (accessed on Oct 9, 2025).

the backend, typically for large blocks.

Some heap memory may be reserved but unallocated. These unallocated regions are described by the `UCRSegmentList` attribute, using symbols such as `ntdll!_HEAP_UCR_DESCRIPTOR`. Our algorithm identifies and skips these regions to avoid reading invalid memory. To ensure proper traversal, we limit traversal to the valid range defined by the `FirstEntry` and `LastValidEntry` pointers. Heap entry metadata such as `Size` and `Flags` may be encoded; when `EncodeFlagMask` is set to `0x100000`, decoding is required by XORing each attribute with the corresponding heap-level `Encoding.Size` and `Encoding.Flag` values.

In particular, the traversal logic begins by retrieving the target process's PEB (line 2), which contains a list of all active heaps using the `ProcessHeaps` attribute (line 3). Each heap is composed of multiple segments, which are iterated through the `SegmentList` (line 4). For each segment, the algorithm initializes a pointer to the first heap entry and collects the unallocated regions described in the `UCR SegmentList` (lines 5–6).

The main traversal loop (lines 7–17) processes each entry from `FirstEntry` to `LastValidEntry`. At each step, it checks whether the current address lies within an unallocated region (line 8). If not, the algorithm reads the heap entry structure from memory (line 10). If the heap uses encoded metadata (indicated by the `EncodeFlagMask` value being set to 0x100000), the entry's `Size` and `Flags` fields are XORed with the corresponding encoding attributes (lines 12–13). The decoded entry is appended to the result list (line 14), and the pointer is advanced by the entry's size (line 15). If the current address lies within an uncommitted region, it is skipped by incrementing the pointer accordingly (line 17). Finally, the compiled list of heap entries is returned (line 18). This algorithm ensures a complete and accurate reconstruction of committed heap entries at the backend layer.

Algorithm 1. Traversal algorithm for extracting heap entries from the backend layer of the Windows NT heap on systems from Windows Vista onwards

Input: A process object P
Output: A list of heap entries

```
1  heapEntries ← ∅
2  peb ← getPEB(P)
3  foreach heap ∈ peb.ProcessHeaps do
4      foreach segment ∈ heap.SegmentList do
5          heapEntryPointer ← segment.FirstEntry
6          uncommitedRegions ← getUncommitedRegions(segment.UCRSegmentList)
7          while heapEntryPointer < segment.LastValidEntry do
8              uncommittedRegion ← findContainingUncommittedRegion(heapEntryPointer, uncommitedRegions)
9              if uncommittedRegion = ∅ then
10                 heapEntry ← readMemory(heapEntryPointer)
11                 if heap.EncodeFlagMask = 0x100000 then
12                     heapEntry.Size ← heapEntry.Size ⊕ heap.Encoding.Size
13                     heapEntry.Flags ← heapEntry.Flags ⊕ heap.Encoding.Flags
14                 heapEntries = heapEntries ∪ heapEntry
15                 heapEntryPointer ← heapEntryPointer + heapEntry.Size
16             else
17                 heapEntryPointer ← heapEntryPointer + uncommittedRegion.Size
18 return heapEntries
```

3.2. Frontend layer

As explained in Section 2.2, the frontend layer (currently implemented as the LFH), is enabled under specific conditions, typically after 16 allocations of the same size (Valasek and Mandt, 2012). Once active, the LFH organizes allocations into subsegments to reduce fragmentation and improve performance (this behavior can be also observed in the latest version of Windows 11).

The presence of LFH is indicated when the `FrontEndHeapType` field of the `ntdll!_HEAP` structure is set to 2. The heap `FrontEndHeap` pointer references an `ntdll!_LFH_HEAP` structure, which contains an array of subsegment zones (`SubSegmentZones`), each pointing to a `ntdll!_LFH_BLOCK_ZONE`. These zones manage application-oriented heap allocations.

Each block zone contains LFH subsegments (`ntdll!_HEAP_SUBSEGMENT`), which in turn reference a `ntdll!_HEAP_USERDATA_HEADER`. Starting with Windows 8.1, the design of these user blocks is obfuscated: offsets are XOR-encoded using a combination of base addresses and a per-process key stored in `ntdll!RtlpLFHKey`. This obfuscation is intended to mitigate exploitation, but must be reversed to traverse heap entries. Once decoded, the lower bits of `EncodedOffsets.StrideAndOffset` provide the offset of the first entry, while the upper bits represent the block stride (i.e., the heap entry size).

In earlier versions, such as Windows 8, decoding is simpler and relies on direct access to specific attributes. However, in Windows Vista and Windows 7, the offset to the first heap entry is located immediately after the block area structure. Once the stride and starting offset are known, our algorithm iterates through the subsegment to extract all entries from the LFH until reaching `BlockCount` (the total number of LFH entries).

Although the design of the LFH is complex, our traversal logic simplifies recovery. Each subsegment is pre-split into fixed-size entries with flags indicating whether an entry is free or occupied, facilitating systematic extraction for forensic analysis.

Algorithm 2. Traversal algorithm for extracting heap entries from the frontend layer (LFH) of the Windows NT heap on systems from Windows 8.1 onwards.

```
   Input: A process object P
   Output: A list of heap entries
1  heapEntries ← ∅
2  peb ← getPEB(P)
3  ntdll ← getNtdll(P)
   // The key value is set to ntdll!RtlpLFHKey symbol at load time
4  lfhKey ← getLfhKey(ntdll)
5  foreach heap ∈ peb.ProcessHeaps do
6      if heap.FrontEndHeapType = 0x02 then
7          lfhHeap ← heap.FrontEndHeap
8          foreach blockZonePointer ∈ lfhHeap.SubSegmentZones do
9              blockZone ← readBlockZone(blockZonePointer)
10             subsegmentAddr = blockZonePointer + (_HEAP_ENTRY.size ∗ 2)
11             endBlockZone = subsegmentAddr + (_HEAP_SUBSEGMENT.size ∗ blockZone.NextIndex)
12             while subsegmentAddr < endBlockZone do
13                 lfhSubsegment ← readSubsegment(subsegmentAddr)
14                 userBlocks ← lfhSubsegment.UserBlocks
15                 encodings ← userBlocks.EncodedOffsets.StrideAndOffset ⊕ userBlocks ⊕ lfhHeap ⊕ lfhKey
16                 firstEntryOffset ← encodings & 0xFFFF
17                 blockStride ← ((encodings ⊕ userBlocks.EncodedOffsets.BlockStride) ≫ 16) & 0xFFFF
18                 i ← 0
19                 heapEntryPointer ← userBlocks + firstEntryOffset
20                 while i < lfhSubsegment.BlockCount do
21                     heapEntry ← readMemory(heapEntryPointer)
22                     heapEntries = heapEntries ∪ heapEntry
23                     heapEntryPointer ← heapEntryPointer + blockStride
24                     i ← i + 1
25 return heapEntries
```

Algorithm 2 describes the procedure for traversing the frontend layer of the Windows NT heap when the LFH is enabled. This traversal applies to systems starting with Windows 8.1, where heap structures are obfuscated to prevent exploitation, requiring decryption logic to access entry-level data. Although our plugin is fully compatible with the legacy LFH implementation found in Windows systems from Windows Vista to Windows 8.1, we focus our description here on the modern LFH architecture, as it represents the current standard for Windows memory management and involves more complex metadata structures.

The algorithm begins by retrieving the PEB and base address of the loaded `ntdll.dll` module (lines 2–3, specifically). The LFH key, used to encode offsets in the heap metadata, is extracted using the `RtlpLFHKey` symbol (line 4). The algorithm then iterates over each heap in the `ProcessHeaps` list (line 5). It is only executed if the heap uses the LFH frontend, indicated by `FrontEndHeapType` equal to `0x02` (line 6).

On eligible heaps, the `FrontEndHeap` pointer is dereferenced to obtain the LFH heap structure (line 8), from which an array of block zones is accessed using `SubSegmentZones` (line 9). Each block zone contains several LFH subsegments (data structures that group fixed-size allocations). Their address range is calculated using known structure sizes and the zone's `NextIndex` field (lines 10–11). The algorithm then iterates over each LFH subsegment (lines 12–24).

Each LFH subsegment references a `UserBlocks` structure (line 14), which contains encoded metadata. To determine the offset of the first heap entry and the stride (entry size), the `StrideAndOffset` value is XOR-encoded with the addresses of the user block, the LFH heap, and the LFH key (line 15). The offset is extracted from the lower 16 bits (line 16), and the stride is decoded from the upper 16 bits using a second XOR operation with `BlockStride` (line 17). The pointer to the first heap entry is calculated by adding the offset to the base of the user block (line 19).

Finally, a loop iterates over all heap entries in the subsegment (lines 20–24). Each entry is read from memory and added to the result list, with the pointer advancing one step at each iteration. The loop continues until all entries defined by `BlockCount` have been collected.

The function concludes by returning the list of heap entries (line 25). This algorithm allows for the accurate extraction of LFH-managed heap allocations despite their hard-coded design, supporting detailed analysis of user-space memory in modern Windows environments.

Table 1
Windows operating systems for the Intel CPU family tested for the HeapList plugin.

Operating System	Build Number	Architecture
Windows 11 Home 24H2	10.0.26100.1742	x64
Windows 10 Education 22H2	10.0.19045.2965	x86, x64
Windows 8.1 Core	6.3.9600	x86, x64
Windows 7 Professional SP1	6.1.7601	x86, x64
Windows Vista Business SP2	6.0.6002	x86, x64

4. Evaluation and validation

In this section, we first describe the platform coverage and compatibility of the HeapList plugin on different Windows versions and architectures. Next, we detail the validation methodology used to confirm the correctness of the heap walk, including comparisons with `WinDbg` output and controlled testing with custom C/C++ programs. Then, we highlight the plugin's usage and output capabilities. Finally, we make our implementation publicly available to promote transparency, reproducibility, and community collaboration in memory forensics research.

4.1. Platform support

We evaluated our plugin on various Windows operating systems, both legacy and modern. As shown in Table 1, we tested the plugin on Windows versions ranging from Windows Vista SP2 to Windows 11 Home 24H2, and verified its compatibility with ×86 and ×64 architectures where applicable.

Through these tests, we confirmed that HeapList correctly traverses and extracts NT heap entries across the various heap layouts and symbol definitions used in these versions of Windows. This broad coverage demonstrates the practical applicability of the plugin in forensic workflows involving memory dumps from a wide range of real-world systems.

4.2. Cross-validation with WinDbg

To validate the correctness of our NT heap traversal algorithms, we performed cross-validation with `WinDbg`, the official Microsoft

Table 2
Summary of experimental results obtained from heap analysis of Telegram Desktop using HeapList.

Experiment	Recoverability	Result Summary
Access to all conversations	Partial	Messages from unaccessed chats retrievable but unlinked.
Recently accessed conversations	Full	Includes older messages beyond the current chat view.
Media shared	Full/ Conditional	Images, locations, and contacts retrievable; documents only if downloaded.
Edited messages	Full	Original and edited versions retrievable but not correlated.
Deleted messages	Full	Contents remain accessible after deletion.
Deleted conversation	Full	Contents remain accessible after deletion.
Contact list	Partial	Contacts retrievable even without prior interaction.
Unlock password	Full	Password recovered in plaintext.
Log out	Minimal	Very limited remnant data after logout.

Windows debugger (Microsoft, 2025). We used the `!heap` extension with the `-hl` option, which provides a detailed listing of heap entries in the backend and frontend layers.

We ran HeapList with memory dumps from each platform listed in Table 1 and compared the output with the corresponding WinDbg results. In each case, we confirmed that the heap entries extracted by our plugin matched those reported by WinDbg in terms of entry address, size, and status (e.g., busy or free). This consistency validated the robustness of our symbol traversal logic, metadata decoding, and offset calculations for both heap layers.

By aligning our plugin's output with a reliable field-checking tool, we demonstrated that HeapList provides reliable and debugger-consistent visibility into NT heap internals, even with coding or structural variations between Windows versions. Let us remark, however, that WinDbg and HeapList have fundamentally different purposes and operational contexts. WinDbg is a general-purpose debugger designed for use in live environments controlled by developers or systems engineers, while HeapList is designed to operate offline as a forensic analysis tool to help investigators extract and examine heap structures from memory dumps as part of post-mortem investigations. Consequently, our plugin is not comparable with WinDbg, as it does not compete with it, but rather extends heap introspection capabilities to forensic workflows.

4.3. Controlled functional testing

In addition to debugger-based validation, we performed controlled experiments with manually created C/C++ programs that used the standard Windows heap API (e.g., `HeapCreate`, `HeapAlloc`, `HeapFree`). We designed the allocation patterns to activate the frontend and backend layers, taking into account that the latter is normally activated after 16 allocations of the same size (see Section 3.2). These programs were run in virtualized environments using `VMware Workstation 17.0.2 Pro` (build number 21581411), following clean installations of Windows with `WinDbg` preinstalled. Memory snapshots were captured after execution, and heap entries were retrieved using HeapList. We then compared the extracted entries to the expected runtime behavior observed in WinDbg, confirming the accuracy of metadata retrieval, memory alignment, and allocation content.

4.4. Usage and output capabilities

HeapList supports flexible usage scenarios tailored for forensic analysis. It allows filtering by process ID, selective dumping of entries by base address, and complete extraction of a process's heap memory. A representative example of our plugin's output is shown in Listing 1, where it is used to extract all heap entries from a specific process. The output includes entry metadata (addresses, size, flags), memory layer classification (backend or LFH), and a preview of the stored content. In a typical investigation, the workflow involves dumping these entries to disk for offline analysis. Analysts can then leverage prior knowledge to analyze application-specific data structures or use the granular view to uncover previously unknown details.

Listing 1: Example output from the HeapList plugin, showing the heap entries extracted for a given process. Each row includes the entry's address, size, status (e.g., busy or free), memory layer (backend or LFH), and a preview of the contents. The dumped entries are written to individual output files for later forensic analysis.

```
$ vol3.py -f /path/to/windows-memory-image.raw windows.anonymizedplugin --pid 11956 --dump-all
Volatility 3 Framework 2.11.0
Progress:  100.00               PDB scanning finished
PID     Name    Heap    Segment Entry   Size    Flags   State   Layer   Data    File Output

11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b30740   0x20    [01]    busy
backend PUBLIC=C:\Users\        11956.heap.22225b30740.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b30760   0x20    [01]    busy
backend SESSIONNAME=Cons        11956.heap.22225b30760.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b30780   0x20    [01]    busy
backend SystemDrive=C:..        11956.heap.22225b30780.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b307a0   0x20    [01]    busy
backend SystemRoot=C:\Wi        11956.heap.22225b307a0.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b307c0   0x40    [01]    busy
backend USERDOMAIN_ROAMINGPROFILE=DESKTOP-H23BTC        11956.heap.22225b307c0.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b30800   0x20    [01]    busy
backend USERNAME=User...        11956.heap.22225b30800.dmp
[...redacted...]
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b414c0   0x310   [01]    busy
backend AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        11956.heap.22225b414c0.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b417d0   0x2010  [09]    busy internal
backend ...%".....%"...........,.............        11956.heap.22225b417d0.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b41820   0x310   [80]    free    lfh
.....................................        11956.heap.22225b41820.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b41b30   0x310   [90]    busy    lfh
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        11956.heap.22225b41b30.dmp
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b41e40   0x310   [90]    busy    lfh
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        11956.heap.22225b41e40.dmp
[...redacted...]
11956   cmd.exe     0x22225b30000   0x22225b30000   0x22225b437e0   0x4010  [01]    busy    backend ????
Unavailable
[...redacted...]
```

4.5. Availability and licensing

To ensure transparency, reproducibility, and community adoption, we have released the source code for the HeapList plugin under the GNU/GPLv3. The full source code, including documentation and usage

examples, is publicly available at:

https://github.com/reverseame/heaplist

We welcome contributions, issue reports, and feedback from the digital forensics and reverse engineering communities to improve functionality, expand platform support, and explore new forensic use cases – especially since future versions of Windows continue to rely on undocumented internal components.

5. Case study: Telegram Desktop

To demonstrate the applicability of the proposed methodology in a realistic forensic scenario, we analyzed Telegram Desktop using information extracted with HeapList. The case study focuses on the 64-bit version of Telegram Desktop (v6.1.4), installed on Windows 11 Home 24H2 ×64 (build 10.0.26100.1742, 8 GB of RAM) in a virtualized environment running on VMware Workstation 17 Pro (version 17.0.0 build-20800274). This configuration allowed us to generate full memory dumps of the guest operating system after each of the experiments described below.

As described in Section 2, the heap stores dynamically allocated memory, meaning that only data actively used during a session, such as user interactions in Telegram Desktop, resides in memory. The LFH efficiently manages uniformly sized memory blocks, which, in the context of instant messaging applications, may include message-related data structures among other dynamically allocated objects. This behavior facilitates the identification of forensic artifacts, as relevant evidence typically resides in discrete heap entries rather than scattered across different memory regions. Consequently, HeapList allows forensic examiners to selectively extract such artifacts once identified, thus reducing the need for extensive reverse engineering during forensic investigations.

This heap-based approach complements other analytical techniques previously employed in similar studies, such as those presented by Fernández-Álvarez and Rodríguez (2022), by providing a structured, memory-centric perspective on user activity within applications.

5.1. Description of experiments

We designed nine experiments that simulate a forensic scenario in which a suspect recently used Telegram Desktop before their arrest, adopting the scenario proposed by Fernández-Álvarez and Rodríguez (2022). Throughout these experiments, Telegram remains active on the seized device, but is secured by a local password unknown to the investigators, preventing direct interactive inspection.

The primary goal is to recover all artifacts related to the user's Telegram activity. This includes recently started or accessed conversations and exchanged media (e.g., images, documents, locations, and contacts). Other items of interest include previous versions of edited or deleted messages, conversation history, and contact metadata such as names and phone numbers. Given the constraints of the scenario, we also evaluated whether locally stored authentication material (the password or related secrets) can be recovered and what residual information persists if the user logs out before the seizure.

5.2. Experimental results

Table 2 summarizes the results of our experiments. Using HeapList, we were able to extract user messages even from conversations that had not been accessed recently. While the conversation title (i.e., the name of the other user, group, or channel) was retrievable, it could not be reliably linked to individual messages. This limitation arises because heap entries store dynamic data structures, such as strings, without high-level relational context. Consequently, multiple messages could be retrieved, but they could not be associated with specific conversations, and this lack of linking also affects other types of extracted artifacts.

Regarding media files shared through the app, images were fully retrievable because they are automatically downloaded and cached locally by default. The same behavior was observed for shared locations and contact information. In contrast, documents are not automatically downloaded; therefore, their recovery is only possible when the user explicitly downloads them before the memory seizure.

Both the original and modified versions of the edited messages were recoverable, although we were unable to establish correlations between them. This finding highlights a key advantage of our approach over the approach of Fernández-Álvarez and Rodríguez (2022), as it allows access to message content even after edits. Similarly, deleted messages and conversations remained accessible in memory, indicating that their data persisted in the heap after the deletion actions.

In the case of contact information, many phone numbers were discovered in the user's address book, including entries not associated with previous interactions. Interestingly, the local unlock password was found in plaintext within a heap entry. When the seized device remains powered on, the presence of this password in memory allows investigators to directly access the Telegram Desktop application and perform a live analysis of its contents.

Finally, in a complementary experiment in which the user logged out before the system seizure, only a minimal number of messages could be recovered. This observation suggests that heap-resident data associated with active sessions is freed or overwritten upon session closure, reducing recoverable evidence.

Future work is required to correlate retrieved messages and associate them with their corresponding conversations. This task would involve extending the current capabilities of HeapList and performing application-specific analysis, since the relationships between retrieved heap structures depend on each program's internal data models. Approaches similar to those proposed by Fernández-Álvarez and Rodríguez (2022) and Oliveri et al. (2023) could support this direction, providing a basis for reconstructing high-level semantic relationships from low-level memory artifacts.

6. Potential applications and limitations

In this section, we discuss the potential applications of user heap analysis in memory forensics and highlight its limitations.

6.1. Potential applications

6.1.1. Support for reverse engineering and targeted memory analysis

One of the most immediate applications of HeapList is to assist forensic analysts in reverse engineering, enabling the precise inspection of structures residing within the heap. This capability aligns with and extends methodologies used in previous works such as Cohen (2015) and Block and Dewald (2017). Specifically, HeapList complements the research of Fernández-Álvarez and Rodríguez (2022) by focusing the analysis on specific data artifacts stored in the heap. By dividing the heap into individual entries rather than treating it as a contiguous block of memory, this targeted approach reduces noise and improves clarity, offering a more granular alternative to broader analysis techniques based on virtual address descriptors (Dolan-Gavitt, 2007).

6.1.2. Visibility into dynamic payloads

Advanced adversaries and red team implants often rely on dynamic memory allocation to store payloads, obfuscate command and control activity, and evade traditional detection mechanisms (Cheng et al., 2012; Selmanaj, 2024; Hembree et al., 2025). However, most forensic tools lack the structural granularity necessary to reconstruct heap behavior or attribute memory artifacts to specific actions. As a result, dynamic memory is often treated as an opaque region, limiting the effectiveness of incident response and threat attribution. HeapList addresses this deficiency by reconstructing the internal structure of the NT heap. This offers analysts the visibility required to manually inspect and recover payloads that reside in dynamic memory, providing a granular

view that complements signature-based detection.

6.1.3. Heap layout inspection

Attackers typically exploit heap vulnerabilities such as use-after-free, double-free, out-of-bounds access, and type confusion by carefully crafting the heap layout (Ding et al., 2010; Bouffard et al., 2015; Haller et al., 2016; Gui et al., 2022; Du et al., 2024). Due to the random behavior of the LFH frontend, which non-deterministically distributes allocations across subsegments, successful exploitation requires precise heap manipulation. Techniques such as NOP sledding, heap spraying, and heap maintenance (also known as *heap feng shui*, *heap shaping*, or *heap massaging*) are used to make heap allocations more predictable. While traditional memory forensics treats the heap as a flat region, HeapList allows analysts to visualize the resulting memory distribution and inspect the fragmentation. This structural information facilitates post-mortem analysis of failed exploits or anomalous heap states, enabling investigators to validate hypotheses about heap manipulation techniques during forensic reconstruction.

6.1.4. Memory triage and evidence prioritization

In large-scale investigations or urgent incident response scenarios, analysts often need to prioritize memory regions most likely to generate actionable evidence (Casey, 2011; Ligh et al., 2014). HeapList facilitates this process by offering structured access to dynamic memory allocations and facilitating heap navigation, allowing analysts to selectively explore specific heap entries or subregions based on process ID, address range, or allocation metadata (Carvey, 2014). This targeted approach allows investigators to focus on regions containing volatile data, such as credentials, decrypted payloads, or user activity, without needing to scan the entire memory image. As a result, our plugin streamlines memory triage workflows, reduces the burden of extensive analysis, and accelerates evidence acquisition during live and post-mortem investigations.

6.2. Limitations

All information related to a process's heap resides in user space, rather than in the non-paged kernel-mode pool. As a result, memory analysis of heap structures is susceptible to paging and swapping (Silberschatz et al., 2021). Even when heap data is present in memory, extracting it at a granular level requires traversing a chain of user-space structures, from the PEB to heap segment-specific metadata. Each of these traversal points introduces potential failure modes, making it difficult to analyze the heap as a coherent memory region. Furthermore, we acknowledge that partial data loss or malformed structures may affect the accuracy of heap entry recovery, potentially leading to incomplete results or misleading artifacts (Rzepka et al., 2024). This remains a practical challenge for memory forensic robustness (Rzepka et al., 2025).

At the backend layer, heap entries can vary in size, requiring reading the size of each entry to advance the traversal. This requires access to the metadata of the heap entries preceding the application data, data that may be partially missing or paged out. Although a scanner could, in theory, recover heap entries by identifying potential structure boundaries, the fixed size of the metadata fields (8 bytes on $\times$86 and 16 bytes on x64), followed by variable or random data, provides limited entropy for signature-based recovery. This issue does not affect the LFH frontend, where all entries in a subsegment are of a uniform size. In these cases, the lack of heap metadata can be addressed by using known-block stepping to reconstruct the allocation sequence.

Our current implementation focuses on the NT heap, as provided by the Windows heap API (see Section 2.2). Therefore, it does not support custom allocators or application-specific heap implementations, such as those used by modern browsers or runtimes. These allocators are typically based on proprietary structures and algorithms, requiring specialized reverse engineering to reconstruct their internal memory organization. Similarly, we do not yet support the Segment Heap, introduced in Windows 10 and used by default in modern UWP apps. However, based on previous reverse engineering work by Yason (2016), adding Segment Heap support is viable and remains a goal for future development. It's worth noting that this would only affect systems running Windows 10 and later, simplifying integration.

Similarly, the accuracy of our extraction depends on the integrity of the heap metadata structures maintained by the operating system. Since HeapList parses these structures (e.g., segment headers and fragment tags) to traverse the heap, it is susceptible to anti-forensic techniques that deliberately corrupt or alter this metadata. For instance, if an attacker modifies fields such as encoding masks or segment signatures, or if severe heap corruption occurs during an exploit, the plugin might fail to parse the affected regions or completely omit the damaged structures. Therefore, at the present, our tool is best used to inspect structurally valid heaps or to identify inconsistencies where parsing fails. Developing capabilities to resiliently handle such anti-forensic tampering and severe corruption remains an open challenge that requires further research.

Finally, while HeapList enables the extraction of rich application data from user-space memory, reconstructing high-level semantic relationships (such as correlating retrieved messages with specific conversations, as discussed in Section 5.2) requires further investigation. Such extensions would likely involve application-specific reverse engineering to understand how heap-resident structures interrelate, as suggested by approaches such as those of Fernández-Álvarez and Rodríguez (2022) and Oliveri et al. (2023).

7. Related work

In this section, we review the major efforts in heap analysis, their application to forensics, and contrast them with our approach. We also highlight how our approach advances the state of the art.

The seminal works by Valasek (2010) and Valasek and Mandt (2012) provide fundamental insights into the internal design of the Windows NT heap, detailing the main structures and allocation algorithms for both the backend and frontend of the LFH. In the context of memory forensics, Cohen (2015) extended this knowledge to analyze the NT heap in the `Rekall` memory forensics framework (Google, 2017) for Windows 7 x64 systems. Our work differs from these previous efforts in several important ways. First, it generalizes support for all major Windows versions, from Vista to Windows 11, spanning both $\times$86 and $\times$64 architectures, thus accommodating the evolution of heap internals over time. Second, our plugin is integrated with `Volatility 3`, a modern and widely adopted memory analysis framework, allowing for modularity, plugin reuse, and ongoing community support. Unlike previous work that focused on static heap snapshots or specific OS versions, we offer a reusable, open-source platform for structurally accurate NT heap exploration and forensic data extraction.

On the Linux side, Block and Dewald (2017) demonstrated how heap memory can be analyzed structurally, rather than as a raw byte stream, by reconstructing heap layouts to extract sensitive data such as command-line histories and passwords. This approach underscored the forensic value of interpreting allocator structures. Industry research and exploit development have focused extensively on the Windows heap, which remains a major target in real-world attacks. These efforts have explored the heap as an attack vector for memory corruption vulnerabilities, heap cleanup, and post-exploitation persistence techniques. This continued focus underscores the need for detailed, forensic-grade visibility into heap internals, both to understand attacker behavior and to facilitate effective incident response and threat attribution. In parallel, research on Linux heap exploitation has gained momentum in recent years (Yun et al., 2020; Zeng et al., 2022; Zhang et al., 2023), with a focus on improving exploit reliability through fine-grained manipulation of heap allocators.

8. Conclusions and future work

In this work, we have presented HeapList, a Volatility 3 plugin that enables forensic analysis of the Windows NT heap on all major releases from Windows Vista to Windows 11, supporting both ×86 and ×64 architectures. By reconstructing the internal structures of the backend and frontend layers, we have enabled analysts to extract and examine heap entries with structural accuracy and navigation.

Our plugin enables the investigation of heap-based behaviors in a wide range of processes, making it a valuable tool for memory forensics and incident response. Its effectiveness was demonstrated through a practical case study on Telegram Desktop, where HeapList allowed the recovery of message content, multimedia artifacts, contact data, and even the local unlock password directly from the user-space heap memory. This experiment illustrates the practical applicability of our tool for extracting high-value forensic artifacts from real-world applications.

As part of our ongoing work, we are exploring the integration of behavioral signatures to detect previously unknown threats exploiting heap memory, seeking to improve detection reliability and facilitate proactive threat hunting in modern forensic workflows. We also plan to explore the integration of lightweight heuristics and behavioral signatures to help automatically identify suspicious patterns or anomalous heap usage. Additionally, we are looking to extend HeapList with segment heap support, which will allow full coverage of both modern heap implementations in the Windows memory management system.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used ChatGPT-4 to improve readability and language. After using this tool/service, the authors reviewed and edited the content as needed and assume full responsibility for the content of the publication.

Acknowledgments

This research was supported in part by grant PID2023-151467OA-I00 (CRAPER), funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU, by grant TED2021-131115A-I00 (MIMFA), funded by MICIU/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR, by grant *Proyecto Estratégico Ciberseguridad EINA UNIZAR*, funded by the Spanish National Cybersecurity Institute (INCIBE) and the European Union NextGenerationEU/PRTR, by grant *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-23R), funded by the University, Industry and Innovation Department of the Aragonese Government.

References

Block, F., Dewald, A., 2017. Linux memory forensics: dissecting the user space process heap. Digit. Invest. 22, S66–S75.
Bouffard, G., Lackner, M., Lanet, J.L., Loinig, J., 2015. Heap…Hop! heap is also vulnerable. In: Joye, M., Moradi, A. (Eds.), Smart Card Research and Advanced Applications. Springer International Publishing, Cham, pp. 18–31.
Bradley, A.R., 2011. Programming for Engineers: a Foundational Approach to Learning C and Matlab. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 113–136 chapter Memory: The Heap.
Carvey, H., 2014. In: Windows Forensic Analysis Toolkit: Advanced Analysis Techniques for Windows 8, fourth ed. Syngress.
Casey, E., 2011. Digital evidence and computer crime: forensic science. Computers, and the Internet, third ed. Academic Press Inc.
Cheng, T.H., Lin, Y.D., Lai, Y.C., Lin, P.C., 2012. Evasion techniques: sneaking through your intrusion detection/prevention systems. IEEE Commun. Surv. Tutor. 14, 1011–1020.
Cohen, M., 2015. Forensic analysis of windows user space applications through heap allocations. In: 2015 IEEE Symposium on Computers and Communication (ISCC), pp. 237–244.
CrowdStrike, 2024. CROWDSTRIKE 2024 GLOBAL THREAT REPORT techreport. CrowdStrike.
Ding, Y., Wei, T., Wang, T., Liang, Z., Zou, W., 2010. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference. Association for Computing Machinery, New York, NY, USA, pp. 327–336.
Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. Digit. Invest. 4, 62–64.
Du, S., Li, Z., Li, J., 2024. Heap vulnerability exploitation techniques on the linux platform: an overview. In: 2024 9th International Conference on Signal and Image Processing (ICSIP), pp. 371–382.
Fernández-Álvarez, P., Rodríguez, R.J., 2022. Extraction and analysis of retrievable memory artifacts from windows telegram desktop application. In: Selected Papers of the Ninth Annual DFRWS Europe Conference, vol. 40, 301342. Forensic Science International: Digital Investigation.
GlobalStats, 2025. Desktop Operating System Market Share Worldwide. Online. https://gs.statcounter.com/os-market-share/desktop/worldwide. (Accessed 9 October 2025).
Google, 2017. Rekall Memory Forensic Framework. Online. https://github.com/google/rekall. (Accessed 9 October 2025).
Gui, B., Song, W., Xiong, H., Huang, J., 2022. Automated use-after-free detection and exploit mitigation: how far have we gone? IEEE Trans. Software Eng. 48, 4569–4589.
Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., van der Kouwe, E., 2016. TypeSan: practical type confusion detection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp. 517–528.
Hembree, D., Shaffer, A., Singh, G., 2025. Obfuscation, stealth, and non-attribution in automated red team tools. In: Proceedings of the 20th International Conference on Cyber Warfare and Security (ICCWS). Academic Conferences International Limited, pp. 132–141.
Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.
Microsoft, 2025. WinDbg (Online: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/. (Accessed 9 October 2025).
Microsoft Corporation, 2022. PEB Structure (winternl.h) [Online. https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb. (Accessed 9 October 2025).
MITRE, 2025. Mitre ATT&CK [Online. https://attack.mitre.org/. (Accessed 9 October 2025).
Mitropoulos, S., Patsos, D., Douligeris, C., 2006. On incident handling and response: a state-of-the-art approach. Comput. Secur. 25, 351–370.
Momot, F., 2013. Understanding the windows allocator: a redux (Online: https://web.archive.org/web/20171021122823/http://www.leviathansecurity.com/blog/understanding-the-windows-allocator-a-redux. (Accessed 9 October 2025).
Oliveri, A., Dell'Amico, M., Balzarotti, D., 2023. An OS-agnostic approach to memory forensics. In: 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27-March 3, 2023. The Internet Society.
Repel, D., Kinder, J., Cavallaro, L., 2017. Modular synthesis of heap exploits. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. Association for Computing Machinery, New York, NY, USA, pp. 25–35.
Rzepka, L., Ottmann, J., Freiling, F., Baier, H., 2024. Causal inconsistencies are normal in windows memory dumps (Too). Digital Threats 5. https://doi.org/10.1145/3680293.
Rzepka, L., Ottmann, J., Stoykova, R., Freiling, F., Baier, H., 2025. A scenario-based quality assessment of memory acquisition tools and its investigative implications. In: DFRWS EU 2025 - Selected Papers from the 12th Annual Digital Forensics Research Conference Europe, vol. 52, 301868. Forensic Science International: Digital Investigation.
Selmanaj, D., 2024. Adversary Emulation with MITRE ATT&CK. O'Reilly Media, Inc.
Silberschatz, A., Galvin, P.B., Gagne, G., 2021. Operating System Concepts, tenth ed. John Wiley & Sons, Inc.
Valasek, C., 2010. Understanding the low fragmentation heap (Online: https://www.illmatics.com/Understanding_the_LFH.pdf. (Accessed 9 October 2025).
Valasek, C., Mandt, T., 2012. Windows 8 heap internals (Online: https://illmatics.com/Windows%208%20Heap%20Internals.pdf. (Accessed 9 October 2025).
Yason, M.V., 2016. Windows 10 segment heap internals (Online: https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf. (Accessed 9 October 2025).
Yosifovich, P., Russinovich, M.E., Ionescu, A., Solomon, D.A., 2017. In: Windows Internals: System Architecture, Processes, Threads, Memory Management, and More, Part 1, seventh ed. Microsoft Press.
Yun, I., Kapil, D., Kim, T., 2020. Automatic techniques to systematically discover new heap exploitation primitives. In: Proceedings of the 29th USENIX Conference on Security Symposium. USENIX Association, USA, pp. 1111–1128.
Zeng, K., Chen, Y., Cho, H., Xing, X., Doupé, A., Shoshitaishvili, Y., Bao, T., 2022. Playing for K(H)eaps: understanding and improving Linux Kernel exploit reliability. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA, pp. 71–88.
Zhang, B., Chen, J., Li, R., Feng, C., Li, R., Tang, C., 2023. Automated exploitable heap layout generation for heap overflows through manipulation distance-guided fuzzing. In: Proceedings of the 32nd USENIX Conference on Security Symposium. USENIX Association, USA, pp. 4499–4515.