International Journal of Economics, Commerce and Management
United Kingdom ISSN 2348 0386 Vol. IX, Issue 5, May 2021

http://ijecm.co.uk/

ANALYSIS OF HEAP MANAGER FOR WINDOWS 7 & 10 FROM AN EXPLOITATION PERSPECTIVE

Ștefan Nicula
The Bucharest University of Economic Studies, Romania
niculastefan21@gmail.com

Răzvan Daniel Zota
The Bucharest University of Economic Studies, Romania
zota@ase.ro

Abstract

In order to understand and successfully exploit a heap memory corruption vulnerability on Windows, multiple concepts such as the Windows Heap Manager internal routines should be grasped. The article aims to narrow down some important concepts related to the Heap Manager on both Windows 10 and Window 7 operating systems. PE applications that are making use of the heap memory can choose to implement their own heap manager by using the VirtualAlloc function or can opt for the Windows direct implementation by invoking Win API specific functions for different routines such as allocation, reallocation and heap free. Key differences can be noted between the two analyzed Windows versions and some of them are restricting exploitation methods that worked on older operating systems. An important aspect of the research was focused on analyzing the heap memory layout after consecutive or adjacent allocations using allocators and free primitives equally on both operating systems. A chapter is dedicated to the analysis of different protection mechanisms enforced and how it affects the exploit development process. The results indicate that Windows 7 Heap Manager is more deterministic and can be leveraged better in comparison with the randomization introduced by Windows 10. Additionally, the internals of the new Segment Heap introduced in Windows 10 and the NT Heap are largely different however, precise heap manipulation is still possible on both operating systems.

Keywords: Heap manager, Windows 7, Windows 10, Exploitation, LFH, BEA, Heap spraying, Read write primitives, Heap memory corruption exploits

INTRODUCTION

The basics of Heap Memory Management on Windows are divided between the kernel space and userland. For the kernel space, memory management is part of the Windows Executive which is a kernel-mode component that provides a variety of services to device drivers, including object management, memory management, process and thread management, input/output management, and configuration management.[1] The ntoskrnl.exe contains the implementation of the Windows Executive. Virtual Memory Management handles a big portion of the functionalities including taking care of translation, mapping the virtual memory to physical memory, dividing the memory into pages, and attributing page memory contents to disk. There are also additional services like Memory Mapped files, copy-on-write memory, and large address spaces such as Address Windowing Extension. [2]

In the userland, memory management is divided between stack and heap. The stack is a fixed piece of memory that is attributed to every process thread. It is mainly used for local variables, register operations, saved return pointers, Structure Exception Handling information storage, and other information that have a short lifespan. The heap is the managed memory that can be used in a dynamic manner as requested by the process. Being a dynamic on-demand memory area, functionalities such as freeing, allocating or resizing the memory are needed in order to manage the data in transit. [3]

The presented concepts are key elements in the exploit development process that target a heap-related vulnerability on a Windows desktop application. The concept of heap exploitation is getting more traction in the modern days. Browsers are one of the most targeted software for heap memory corruption vulnerabilities and exploits. This is because of numerous entry-points and complex engines and routines.

A desktop application running on a Windows environment can opt for a self-heap implementation by asking direct memory access using the VirtualAlloc function or have the Windows Heap Manager take over the routines and implementations by using the Windows API function for heap access. [4]

In the case of a Windows-based heap management solution, internals of the heap routines are the same as the ones that can be used by a command line application. This greatly helps in debugging and understanding the heap dynamics as any created application that uses the Windows heap API can be used in order to analyze the internals. An interesting example of an application that uses the Windows Heap manager is the Internet Explorer browser. It is arguably an outdated and increasingly unpopular browser however, with the recent introduction of Internet Explorer mode in the Edge Chromium-based system, we are seeing some possible attack vectors opening and creating opportunities in terms of new attacks and exploitations.

Regardless of the targeted application, the concepts of heap memory layouts, routines and protection mechanisms are key aspects in determining the exploit development process. If we take a broad overview at different heap manager implementations we can note specific common concepts and an overall common ground in terms of routines, allocations and complex object manipulation. By studying the behavior of the Windows Heap Manager, we can extrapolate the concepts and draw parallels to other proprietary implementations as well.

INTERNAL HEAP MANAGEMENT MECHANISMS

Applications running on the Windows environment have two mechanisms available to allocate and use heap memory. The first choice is to use the default Windows Heap Manager that is provided directly by the operating system, this includes all the needed routines such as creating a heap, allocation, deallocation and reallocation. The second option provided is to use direct memory access with the help of **VirtualAlloc** function. [7] The latter offers applications the opportunity to implement their own Heap Management routines and mechanisms. We often see this implementation on standalone complex applications such as browsers or PDF readers that request a large portion of heap memory which will later be managed internally by their own implemented routines for allocating, deallocating, and resizing the heap.

The default Windows Heap Manager makes use of multiple libraries that contain such routines that ultimately end in ntdll.dll library. As such, the **RtlAllocateHeap**, **RtlReAllocateHeap**, and **RtlFreeHeap** are the functions used by the Heap Manager in order to manage the heap memory. Techniques for debugging the heap memory often rely on hooking these functions and analyzing the local variables passed to them using the stack in order to identify allocations and frees, intercept the returned memory handle and memory address zone that was allocated. [8]

Kernel32.dll heap management functions:

- HeapCreate/HeapDestroy: used for creating or removing a Heap
- HeapAlloc/HeapFree: main functions designated for allocating or freeing a heap chunk
- HeapReAlloc: function used for resizing specific allocations
- VirtualAlloc: assigns a large heap area not managed by the Windows Heap Manager, used for internal Heap Management implementation

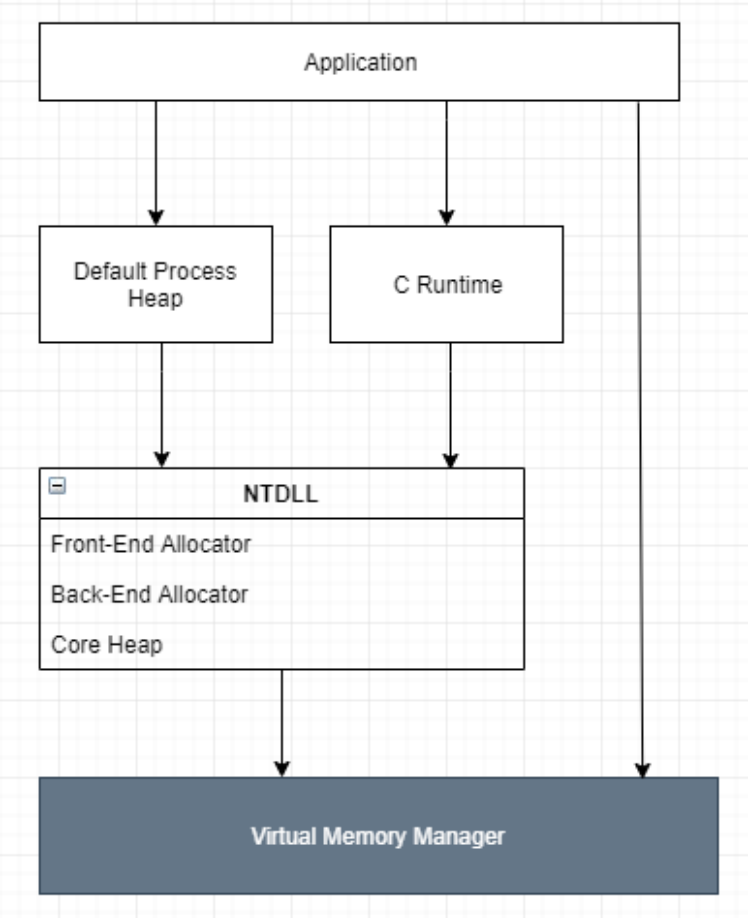

Figure 1: VMM access options for Windows application when using heap memory

Backend Allocator vs Frontend Allocator

Backend allocator is the first mechanism to be used by the Heap Manager. It keeps track of the heap chunks that can be allocated inside the heap segment. However, for optimization purposes, the Frontend allocator has been introduced. After successful repetitive allocations of more than 18 times for a fixed allocation size of no more than 0xF471, then the FrontEnd allocator is being executed.

The Frontend allocator creates the so-called Low Fragmentation Heap (LFH), which basically asks the Heap Manager for a big chunk inside the Heap Segment. In some aspects, it can be compared with the Virtual Alloc behavior however, the Frontend allocator chunk is still kept inside the Heap Segment. These allocations are called buckets. Buckets are formed for specific allocation with fixed sizes. For example, if the user code asks for multiple allocations with size 0x40, the LFH will check if other allocations of the same size are present and will map them accordingly.

An important aspect to keep in mind is that, once activated, the LFH will stay active until the process terminates. This means that, if we are building an exploit that activates the LFH, we should develop the exploit knowing that our future allocations will be dependent on size and their place on the heap will be dictated by the LFH. [9]

When speaking of differences between Windows 7 and 10, we can note a considerable one in terms of allocations for the LFH functionality. One key aspect is that in Windows 7, the LFH allocations are done linearly. For a certain bucket of a fixed size allocation, the frontend allocations will allocate and deallocate in a linear order, each value will be placed next to another one. This behavior allows for a convenient way of spraying the heap, mainly due to the predictability of the allocations.

In Windows 10, the LFH is not linear anymore, instead, allocations in the LFH are made in different locations, even in different Heap Segments. As such, LFH in Windows 10 is way harder to use for heap spraying but not impossible. Techniques such as LFH allocation exhaustion can be used to make a precise allocation. [10]

Another important virtual allocation is the **VirtualAllocdBlocks**. It represents large chunks of data that can't be stored inside a normal heap manager like the frontend or the backend allocators. Instead, these chunks are allocated by directly requesting new virtual memory allocation from the kernel and providing the handle to the user. The offset of the allocated block will be aligned depending on the bitness of the operating system and can have different offsets. [11] We can notice specific allocations gaps between 2 chunks, larger than Windows 7 ones. This automatically translates into bigger holes between allocations and the memory locations of the holes become unpredictable as well. [12]

From the following, we can conclude that for Windows 10 Heap Manager specifically, the key would be to avoid routines such as LFH and VirtualAllocdBlocks.

Heap allocations are automatically rounded to a multiple of 8 bytes, which defines the heap allocation granularity. Both operating systems are using the Process Environment Block to store heap metadata information.

For Windows 7:

- Default Process Heap, offset 0x18
- Number of Heaps, offset 0x88
- List with heaps, offset 0x90

For Windows 10:

- Default Process Heap, offset 0x30
- Number of Heaps, offset 0xe8
- List with heaps, offset 0xf0

The offsets are important because they can provide values and memory addresses for payload size, functionality and can help during a debugging process.

HEAP MEMORY PROTECTIONS AND LAYOUT

Another aspect that we need to take into consideration is the memory layout protection mechanisms such as the Address Space Layout Randomization (ASLR) protection and their corresponding limitations. We also want to keep in mind the Data Execution Prevention (DEP) mechanism as it is a very important aspect that will dictate what mindset we need to apply when going for a heap memory corruption exploitation.

Regarding the ASLR protection, we know that in a full ASLR environment, all the address spaces are randomized for each execution. This will mitigate any exploit that relies on hardcoded addresses. However, by taking a high overview of the ASLR process, we can note that a specific pattern is still kept. As such, the following figure shows the high overview of process memory during execution using ASLR enabled:

- Stack
- Heap
- DLL Modules
- OS libraries
- OS modules

This mapping is kept for all the processes, no matter the protections used. Indeed, the offsets may vary depending on the ASLR aggressiveness, for example, it can randomize the base address by 4 bytes, however the structure remains the same. This means that if we somehow control a notable portion of the memory layout, we can estimate an exact location where our data will land in memory. When speaking of the heap, we can achieve this using heap spraying techniques by abusing different allocators, based on the exploited software. [5]
Allocator primitives allow us to basically allocate a huge amount of data in the heap memory segment. If we know the structure of memory layout and the sizes of the heap chunks and segments, we can estimate the position of the data in memory. [6]

HEAP EXPLOITATION TECHNIQUES

From a heap perspective, as a general short classification, some of the memory corruption issues that can affect a Windows application can be:

- Use after free
- Allocators specific attacks
- Heap overwrite

- Double free exploit
- Uninitialized memory usage
- Off by one

Precise heap spraying is a key element in the exploitation chain of a heap memory corruption vulnerability however, we note that depending on the situation, a heap spray is not necessarily a must. This technique allows the researcher to have a certain level of precision in controlling a direct memory address. By investigating the Windows Heap Manager behavior when allocating data to the heap, we can note specific patterns. Based on those patterns and on the internal processes like backend allocator and LFH, we can use allocators to spray certain memory areas like the Default Heap Segment, to obtain a memory address in which content we can control. An example of address used in these scenarios can be the 0xc0c0c0c0, it is situated in the Default Heap Segment and oftentimes, if sprayed correctly, it will contain user controlled data. [13]

ASLR plays an important role in the defense mechanisms used by the targeted application. Additionally, not every primitive exploited offers the possibilities of direct heap layout control or the ease of use for primitives. Some limitations encountered during an exploit development process can be related to the limitation in terms of the allocated buffer size or the restricted control for the number of allocations supported by the environment. By taking a look at the Pwn2Own competition [14] that targets, among others, full browser attack chain exploits, we can note a specific change compared to the past years, a trend in the heap exploitation techniques used. A distancing from the heap spraying can be observed and instead, information disclosure primitives are now primarily created to obtain offsets and memory addresses needed to build payloads. Considering the protection mechanisms implemented by modern browsers at present, a successful exploit targeting a heap memory corruption requires additional support for bypasses of the sandbox environment. Information disclosure primitives are created by the researcher by leveraging heap-specific vulnerabilities and the heap layout. An example of how this can be achieved is by allocating adjacent objects in a use after free scenario, keeping reference to the objects and object attributes and finding ways to modify the object headers and rewrite properties that will result in arbitrary read-write opportunities.

We can note that certain Heap protections implemented in newer versions have not yet been approached. However, if we keep track of what we are activating and what memory zones to use, we can bypass certain protection by avoiding triggering or activating them. For example, if a heap chunk is being smashed in a heap overflow scenario, the Windows Heap Manager will not be aware unless that specific memory region will be used again or traversed. So if we take

care not to trigger any unwanted heap memory usages on that area, then protections against heap header corruption using heap cookies and metadata will not be activated.

The mindset for a heap corruption exploitation situation is much different compared to the stack corruption one. This is mainly due to the fact that in a heap exploitation scenario, the first primary focus should be on obtaining a memory leak. With the help of a memory leak, the target would be to obtain a base address for the exploit chain. If, in the case of a heap overflow scenario, we are going directly for EIP control, then ASLR and DEP modules will not permit any code execution paths. As such, in almost all the heap exploitation scenarios, the first main goal is leaking a proper memory address on which we can build upon. Leaking such data and creating the right heap layout to do so is strongly related to the Heap Manager used. Overwriting adjacent objects allows the possibility to create read-write primitives involved in the memory leak process and used on the exploit chain. [15]

The usage of page heap memory inspection tools is essential for both debugging and creating an exploit. We can activate the heap page when running a native debugger such as WinDBG. This will actually make some interesting modifications to the heap layout. It will create a clone Heap Segment for each Heap Segment created by the Heap Manager. As such, some important changes are happening and each address is mapped on the clone heap, allowing for debugging and crash analysis. On a heap user after free scenario, a simple crash without the page heap enabled will not provide all the needed details in order to successfully backtrace the issue. [16]

The DEPS spraying technique is using the data assigned to objects to spray the heap. What ends up as spray is the data inside the object and not the object itself. Other allocators result in different results. Depending on the situation, a heap spray containing pointers to objects should be used instead of a pointer to user-controlled data. This is often the case with Double Free or Heap Overflow exploits. [17]

CONCLUSIONS

By looking at both implementations of Heap Manager for Windows 7 and Windows 10, we can create specific test cases for allocations, we can inspect the memory layout and draw important conclusions that will dictate the process of exploit development when encountering software that uses the Windows Heap Manager.

Windows 7 heap is deterministic, allowing for a precise and solid heap spraying, greatly increasing the control of the heap layout. Some of the requirements for achieving layout control include multiple allocations in a short time-frame and taking into account potential minimal noise, this will result in adjacent heap objects. Ultimately, in terms of controlling content at a

predictable address, the Windows 7 heap provides reliable mechanisms in order to manipulate contents at specific memory addresses.

The Windows 10 Heap Manager introduces randomization and tries to prevent adjacent memory allocations on the heap. Techniques like using the low-fragmentation heap are to be avoided because the internals are significantly different from the Windows 7 predecessor. Some of the best high profile exploits on Google Chrome or Windows RDP protocol, for example, are the most reliable on older versions of Windows like 7 or XP. Predictable allocations are an important trait for heap exploits.

Modern heap exploitation is a fascinating and a difficult subject to master. The process of reversing and understanding the internals of the affected heap manager is often a tedious process. Fortunately, for heap managers such as the Windows Heap Manager, simple user-controlled C programs can be leveraged in order to debug and understand the internals. They offer direct access to the same Windows API functions used by complex applications and provide support for testing new protection mechanisms, different implementations and behaviors. Future research includes the analysis of Windows kernel memory pools and their comparison with userland heap manager as well as their applicability in Windows kernel driver exploitation.

REFERENCES

[1] Lunchmeat, Exploring Operating Systems: The Windows Executive, April 15, 2018, http://shamrock-security.com/exploring-operating-systems-the-windows-executive

[2] Microsoft MSDN, User mode and kernel mode, 20 April, 2017, https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

[3] Mark E. Russinovich, David A. Solomon, Alex Ionescu, Windows Internals, Part 1 (6th Edition), Microsoft Press, 2012

[4] Mark E. Russinovich, David A. Solomon, Alex Ionescu, Windows Internals, Part 2 (6th Edition), Microsoft Press, 2012

[5] Wei Chen, Heap Overflow Exploitation on Windows 10 Explained, Jun 12, 2019, https://blog.rapid7.com/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/,

[6] Windows Heap Exploitation, Jul 9, 2019, https://www.slideshare.net/AngelBoy1/windows-10-nt-heap-exploitation-english-version

[7] Mario Hewardt, Daniel Pravat, Advanced Windows Debugging, Addison-Wesley Professional, 2007

[8] Tarik Soulami, Inside Windows Debugging, Microsoft Press, 2012

[9] Chris Valasek, Understanding the Low Fragmentation Heap, Blackhat USA 2010, http://illmatics.com/Understanding_the_LFH.pdf

[10] Pavel Yosifovich, Mark Russinovich, David Solomon, Alex Ionescu, Windows Internals, Part 1 76th Edition), Microsoft Press, 2017

[11] Falcon Momot, UNDERSTANDING THE WINDOWS ALLOCATOR: A REDUX, August 13, 2013, https://www.leviathansecurity.com/blog/understanding-the-windows-allocator-a-redux ,

[12] Corelan Team, Windows 10 x86/wow64 Userland heap, July 5, 2016, https://www.corelan.be/index.php/2016/07/05/windows-10-x86wow64-userland-heap/

[13] Daniel Pravat and Mario Hewardt, Advanced Windows Debugging: Memory Corruption Part II—Heaps, Nov 9, 2007, https://www.informit.com/articles/article.aspx?p=1081496,

[14] Dustin Childs, PWN2OWN MIAMI 2020 - SCHEDULE AND LIVE RESULTS, January 21, 2020, https://www.thezdi.com/blog/2020/1/21/pwn2own-miami-2020-schedule-and-live-results

[15] Fermin J. Serna, The info leak era on software exploitation, 2012, https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2012/BH_US_12_Serna_Leak_Era_Slides.pdf

[16] Mark Vincent Yason, Windows 10 segment heap internals Aug 4, 2016, https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf

[17] Dusan Repel, Johannes Kinder, Lorenzo Cavallaro, "Modular Synthesis of Heap Exploits", Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, DOI 10.1145/3139337.3139346, 25–35, October 2017