Protecting the Heap: Encryption & Hooks

mez0.cc/posts/protecting-the-heap

Table of Contents

As Endpoint Protection gets better, and more of the community build tooling to detect malware in memory, the more evasive implants must become. In this blog I want to look at encrypting the "Heap". More on that in a moment, but for now, the Heap will hold data for a lot longer than the "Stack". The stack will clear as a a function returns. A typical example of Heap usage is a Command and Control (C2) Frameworks configuration; after all, the data to communicate must live somewhere. A generic example of this would be connection strings, and that is what we will use as sample data for this test.

Cobalt Strike introduced Sleep Mask in Cobalt Strike 4.4:

> The **sleep_mask** is Cobalt Strike's ability to mask and unmask itself in memory. The goal of this feature is to push memory detections away from content-based signatures. Although sleep_mask can encode Beacon's data and code (if the agent is in RWX memory), the static stub is still a target for in-memory hunting based on content.

And then updated in 4.5 and is a recommended read... This was then popularised by MDSec again back on July 30th 2021 and is core functionality of their proprietary C2, Nighthawk.

Off the back of this, waldo-irc put together Hook Heaps and Live Free and LockdExeDemo in which he replicates this functionality to further protect a Cobalt Strike Beacon. Similarly, SolomonSklash then produced SleepyCrypt: Encrypting a running PE image while it sleeps which was aimed towards encrypting the sections of a PE in memory.

I've been writing Vulpes since around 2019 and is becoming more stable and the modules I want are almost all there, so now I'm looking into some more evasive behaviour, specifically when the implant is not operational, hence this blog post!

WTF is a Heap

I don't want to turn this into a Computer Science class, so I won't discuss this TOO much. So, what is the heap?

Well, its considered dynamic storage. Meaning it can house large pools of memory which aren't allocated in a contiguous order. Furthermore, the Heap is not managed, and to use it, it must be allocated specifically with functions such as `malloc`, and then freeing with `free`. If this is not done, then a memory leak can occur. This is where it differs from the stack. If something is allocated on the stack, it is cleared when he calling routine returns.

With this in mind, imagine if a configuration for an implant was a big struct. The config would be required quite often, so its likely going to be stored on the heap. This is because if the configuration was completely done at runtime, then the configuration object would be constantly created and deleted. Meaning, if settings are applied at runtime, then they will constantly need re-updating. Thus, the heap is better for this.

Two great references for this:

- Heap Memory in C Programming
- MEMORY IN C – THE STACK, THE HEAP, AND STATIC

Identifying the Heaps

Microsoft have documented this quite well:

- Enumerating a Heap
- Traversing the Heap List

Stringing these two posts together got me 99% of the way there, so lets look at it.

First off, CreateToolhelp32Snapshot is used with the `TH32CS_SNAPHEAPLIST`, `0x00000001`, value:

```
HANDLE hHeapSnap = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,
GetCurrentProcessId());
```

All the snapshot values:

Value	Meaning
TH32CS_INHERIT 0x80000000	Indicates that the snapshot handle is to be inheritable.
TH32CS_SNAPALL	Includes all processes and threads in the system, plus the heaps and modules of the process specified in *th32ProcessID*. Equivalent to specifying the **TH32CS_SNAPHEAPLIST**, **TH32CS_SNAPMODULE**, **TH32CS_SNAPPROCESS**, and **TH32CS_SNAPTHREAD** values combined using an OR operation ('|').
TH32CS_SNAPHEAPLIST 0x00000001	Includes all heaps of the process specified in *th32ProcessID* in the snapshot. To enumerate the heaps, see Heap32ListFirst.
TH32CS_SNAPMODULE 0x00000008	Includes all modules of the process specified in *th32ProcessID* in the snapshot. To enumerate the modules, see Module32First. If the function fails with **ERROR_BAD_LENGTH**, retry the function until it succeeds.**64-bit Windows:** Using this flag in a 32-bit process includes the 32-bit modules of the process specified in *th32ProcessID*, while using it in a 64-bit process includes the 64-bit modules. To include the 32-bit modules of the process specified in *th32ProcessID* from a 64-bit process, use the **TH32CS_SNAPMODULE32** flag.
TH32CS_SNAPMODULE32 0x00000010	Includes all 32-bit modules of the process specified in *th32ProcessID* in the snapshot when called from a 64-bit process. This flag can be combined with **TH32CS_SNAPMODULE** or **TH32CS_SNAPALL**. If the function fails with **ERROR_BAD_LENGTH**, retry the function until it succeeds.
TH32CS_SNAPPROCESS 0x00000002	Includes all processes in the system in the snapshot. To enumerate the processes, see Process32First.
TH32CS_SNAPTHREAD 0x00000004	Includes all threads in the system in the snapshot. To enumerate the threads, see Thread32First.To identify the threads that belong to a specific process, compare its process identifier to the **th32OwnerProcessID** member of the THREADENTRY32 structure when enumerating the threads.

In typical fashion with the snapshotting functions, the setup:

```
hl.dwSize = sizeof(HEAPLIST32);

if (hHeapSnap == INVALID_HANDLE_VALUE)
{
    printf("CreateToolhelp32Snapshot %ld\n", GetLastError());
    return 1;
}
```

At this point, the snapshot is ready to parse. But before that, something needs to actually be put on the heap. This can be done with HeapAlloc , GetProcessHeap and `memcpy` :

```
// Allocate space on the heap
LPVOID pHeapBase = HeapAlloc(GetProcessHeap(), 0, 14);
// copy in the value
memcpy(pHeapBase, "10.10.11.205\0", 14);

// show it!
printf("heapAllocHeap: %p\n", pHeapBase);
```

Next thing is to grab the first heap entry with Heap32ListFirst:

```
BOOL bFirstHeap = Heap32ListFirst(hHeapSnap, &hl);
```

Now loop over with Heap32First and Heap32Next:

```
do
{
    HEAPENTRY32 he;
    ZeroMemory(&he, sizeof(HEAPENTRY32));
    he.dwSize = sizeof(HEAPENTRY32);

    if (Heap32First(&he, GetCurrentProcessId(), hl.th32HeapID))
    {
        do
        {
            printf("Heap Handle: 0x%p\n", he.hHandle);
            he.dwSize = sizeof(HEAPENTRY32);
        } while (Heap32Next(&he));
    }
    hl.dwSize = sizeof(HEAPLIST32);
} while (Heap32ListNext(hHeapSnap, &hl));
```

Working with the heap

Before operating on the heap, it must be locked with HeapLock:

> If the function succeeds, the calling thread owns the heap lock. Only the calling thread will be able to allocate or release memory from the heap. The execution of any other thread of the calling process will be blocked if that thread attempts to allocate or release memory from the heap. Such threads will remain blocked until the thread that owns the heap lock calls the HeapUnlock function.

By doing this, the heap now belongs to the calling thread, meaning no other threads will be accessing the heap whilst we encrypt it:

```
if (HeapLock(he.hHandle)) {
    // locked!
}
```

Walk and Encrypt

Conveniently, HeapWalk allows just that:

```
while (HeapWalk(he.hHandle, &heapEntry) != FALSE) {
    // do something
}
```

This is shown in Enumerating a Heap where a lot of prints are done, we don't care about that here.

When this returns, a PROCESS_HEAP_ENTRY struct will give us allows to all the following information:

```
typedef struct _PROCESS_HEAP_ENTRY {
  PVOID lpData;
  DWORD cbData;
  BYTE  cbOverhead;
  BYTE  iRegionIndex;
  WORD  wFlags;
  union {
    struct {
      HANDLE hMem;
      DWORD  dwReserved[3];
    } Block;
    struct {
      DWORD  dwCommittedSize;
      DWORD  dwUnCommittedSize;
      LPVOID lpFirstBlock;
      LPVOID lpLastBlock;
    } Region;
  } DUMMYUNIONNAME;
} PROCESS_HEAP_ENTRY, *LPPROCESS_HEAP_ENTRY, *PPROCESS_HEAP_ENTRY;
```

`cbData` is the size, and `lpData` is the actual data on the heap. There is all kinds of information here, and the only check we are going to make is that `wFlags` is `PROCESS_HEAP_ENTRY_BUSY`, `0x0004`:

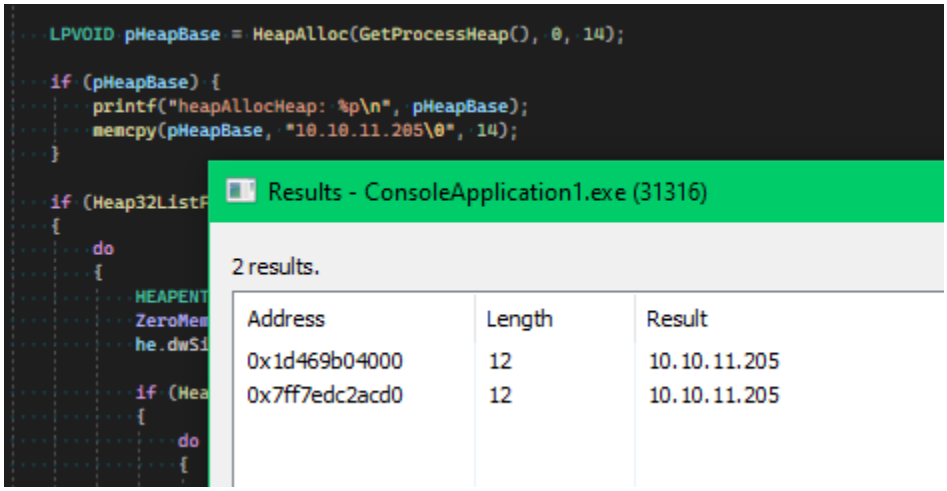
> The heap element is an allocated block.
>
> If **PROCESS_HEAP_ENTRY_MOVEABLE** is also specified, the **Block** structure becomes valid. The **hMem** member of the **Block** structure contains a handle to the allocated, moveable memory block.

This just means that memory is allocated here.

In terms of encrypting, a simple XOR will be used:

```
VOID Xor(unsigned char* lpData, DWORD cbData)
{
    for (int i = 0; i < cbData; i++)
    {
        lpData[i] = lpData[i] ^ 0xff;
    }
}
```

Before actually doing the encryption, lets set a breakpoint on the `Xor` call:

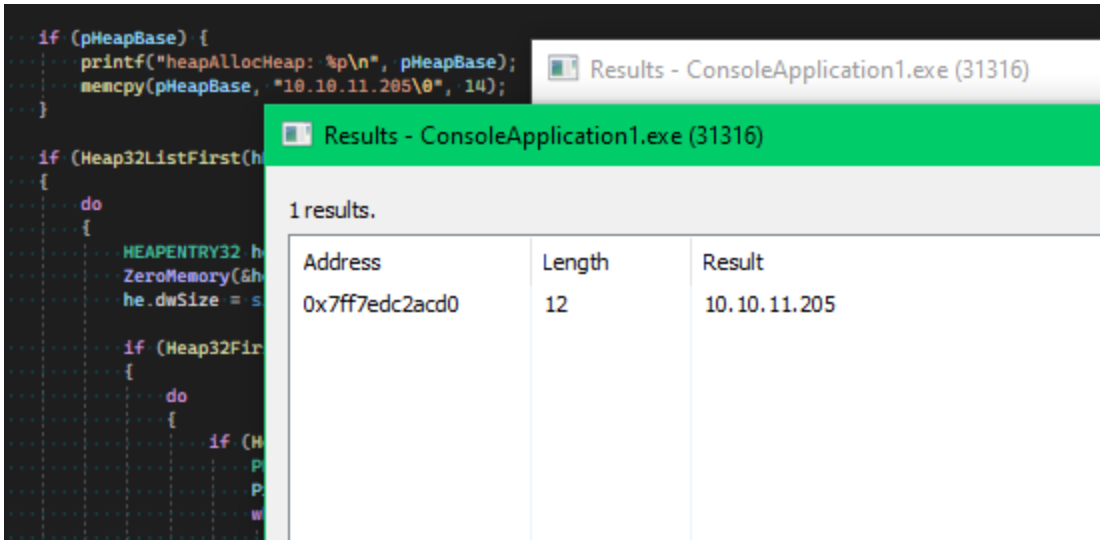


Note, there are two strings. Because the string is just stored in the PE, one of those strings will be from one of the data sections, this is not a focus of the blog for now.

Calling the encryption function:

```
while (HeapWalk(he.hHandle, &heapEntry) != FALSE) {
    if ((heapEntry.wFlags & PROCESS_HEAP_ENTRY_BUSY) && heapEntry.cbData > 0)
    {
        if (pHeapBase == heapEntry.lpData) {
            Xor((unsigned char*)heapEntry.lpData, heapEntry.cbData);
        }
    }
}
```

Rerunning the code:



Now there is only one, remember the other is in the PE data sections.

Before returning, the heap is unlocked with HeapUnlock, releasing ownership:

```
if (HeapUnlock(he.hHandle) == FALSE) {
    printf("HeapUnlock %ld\n", GetLastError());
}
```

Alternatively, everything in the heap can be encrypted, but this could end up giving unexpected results:

```
while (HeapWalk(he.hHandle, &heapEntry) != FALSE) {
    if ((heapEntry.wFlags & PROCESS_HEAP_ENTRY_BUSY) && heapEntry.cbData > 0)
    {
        Xor((unsigned char*)heapEntry.lpData, heapEntry.cbData);
    }
}
```

Hooking Heap Allocations

In order to hook heap allocations effectively, three fnctions are required:

- RtlAllocateHeap
- RtlReAllocateHeap

- RtlFreeHeap

As their names imply, they are responsible for allocating, reallocating, and freeing space on the heap. For hooking, and ease, minhook will be used.

Setting up the hooks

First thing required is to create the three functions as types:

```
typedef PVOID (NTAPI* _RtlAllocateHeap)
(
    PVOID  HeapHandle,
    ULONG  Flags,
    SIZE_T Size
);

typedef PVOID (NTAPI* _RtlReAllocateHeap)
(
    PVOID HeapHandle,
    ULONG Flags,
    PVOID MemoryPointer,
    ULONG Size
);

typedef BOOLEAN (NTAPI* _RtlFreeHeap)
(
    PVOID HeapHandle,
    ULONG Flags,
    PVOID BaseAddress
);
```

Then, three functions that will be used to replace the functionality once hooked:

```
PVOID NTAPI RtlAllocateHeapHook(PVOID HeapHandle, ULONG Flags, SIZE_T Size)
{
    return pRtlAllocateHeap(HeapHandle, Flags, Size);
}

PVOID NTAPI RtlReAllocateHeapHook(PVOID HeapHandle, ULONG Flags, PVOID MemoryPointer,
ULONG Size)
{
    return pRtlReAllocateHeap(HeapHandle, Flags, MemoryPointer, Size);
}

BOOLEAN NTAPI RtlFreeHeapHook(PVOID HeapHandle, ULONG Flags, PVOID BaseAddress)
{
    return pRtlFreeHeap(HeapHandle, Flags, BaseAddress);
}
```

Once that is done, minhook needs to be initialised:

```
MH_STATUS status;

if (MH_Initialize() != MH_OK) {
    return -1;
}
```

Then place the hooks with `MH_CreateHookApi`:

```
status = MH_CreateHookApi(
    L"ntdll",
    "RtlAllocateHeap",
    RtlAllocateHeapHook,
    reinterpret_cast<LPVOID*>(&pRtlAllocateHeap)
);

status = MH_CreateHookApi(
    L"ntdll",
    "RtlReAllocateHeap",
    RtlReAllocateHeapHook,
    reinterpret_cast<LPVOID*>(&pRtlReAllocateHeap)
);

status = MH_CreateHookApi(
    L"ntdll",
    "RtlFreeHeap",
    RtlFreeHeapHook,
    reinterpret_cast<LPVOID*>(&pRtlFreeHeap)
);
```

One final thing is to create three variables to store the original address:

```
_RtlAllocateHeap pRtlAllocateHeap = nullptr;
_RtlReAllocateHeap pRtlReAllocateHeap = nullptr;
_RtlFreeHeap pRtlFreeHeap = nullptr;
```

So now, any call to `RtlAllocateHeap` will be replaced with `RtlAllocateHeapHook` and the original `RtlAllocateHeap` will be stored in `pRtlAllocateHeap`. Then, enable the hooks:

```
if (status == MH_OK)
{
    printf("Hooks Enabled!\n");
}
else
{
    printf("Hooks failed to start!\n");
    return -1;
}
```

Capturing the heap data

In order to do this, a new struct is created:

```
typedef struct _HEAP
{
    HANDLE hHeap;
    PVOID pAllocated;
    ULONG ulFlags;
} HEAP, *PHEAP;
```

This will hold:

1. The handle to the heap
2. The space allocated by the heap
3. The flags used to allocate the heap

As this is a POC, a global vector will be used to hold them:

```
std::vector<HEAP> heaps = {};
```

Updating the `RtlAllocateHeapHook` function:

```
PVOID NTAPI RtlAllocateHeapHook(PVOID HeapHandle, ULONG Flags, SIZE_T Size)
{
    HEAP heap = { 0 };
    heap.hHeap = HeapHandle;
    heap.ulFlags = Flags;
    heap.pAllocated = pRtlAllocateHeap(HeapHandle, Flags, Size);
    heaps.push_back(heap);
    return heap.pAllocated;
}
```

The original `RtlAllocateHeap` is used to allocate space and then stored in the struct, this same value is returned from the hook so the function operates as expected. Once that is done, it is added to the vector. However, `std::vector` is not thread safe, and that `push_back` will cause a crash.

That part is left as a task for the reader, look at:

1. std::deque
2. std::mutex

These two should sort it out, good luck!

Conclusion

In this short blog, I wanted to take a look at working with the heap from both an allocation and hooking perspective. This functionality will be implemented into Vulpes along with the heap tracking functionality.

Full code

```
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>
#include <vector>
#include "minhook/MinHook.h"

typedef PVOID (NTAPI* _RtlAllocateHeap)
(
    PVOID  HeapHandle,
    ULONG  Flags,
    SIZE_T Size
);

typedef PVOID (NTAPI* _RtlReAllocateHeap)
(
    PVOID HeapHandle,
    ULONG Flags,
    PVOID MemoryPointer,
    ULONG Size
);

typedef BOOLEAN (NTAPI* _RtlFreeHeap)
(
    PVOID HeapHandle,
    ULONG Flags,
    PVOID BaseAddress
);

_RtlAllocateHeap pRtlAllocateHeap = nullptr;
_RtlReAllocateHeap pRtlReAllocateHeap = nullptr;
_RtlFreeHeap pRtlFreeHeap = nullptr;

typedef struct _HEAP
{
    HANDLE hHeap;
    PVOID pAllocated;
    ULONG ulFlags;
    SIZE_T Size;
} HEAP, *PHEAP;

std::vector<HEAP> heaps = {};

DWORD GlobalThreadId = GetCurrentThreadId();

VOID Xor(unsigned char* lpData, DWORD cbData)
{
    for (int i = 0; i < cbData; i++)
    {
        lpData[i] = lpData[i] ^ 0xff;
    }
}
```

```
int EncryptHeap()
{
    {
        HEAPLIST32 heapList;

        HANDLE hHeapSnap = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,
GetCurrentProcessId());

        heapList.dwSize = sizeof(HEAPLIST32);

        if (hHeapSnap == INVALID_HANDLE_VALUE)
        {
            printf("CreateToolhelp32Snapshot failed (%d)\n", GetLastError());
            return 1;
        }

        LPVOID pHeapBase = HeapAlloc(GetProcessHeap(), 0, 14);

        if (pHeapBase == NULL)
        {
            return -1;

        }

        memcpy(pHeapBase, "10.10.11.205\0", 14);

        BOOL bFirstHeap = Heap32ListFirst(hHeapSnap, &heapList);

        if (bFirstHeap == FALSE)
        {
            CloseHandle(hHeapSnap);
            return -1;
        }

        do
        {
            HEAPENTRY32 he;
            ZeroMemory(&he, sizeof(HEAPENTRY32));
            he.dwSize = sizeof(HEAPENTRY32);

            if (Heap32First(&he, GetCurrentProcessId(), heapList.th32HeapID))
            {
                do
                {
                    if (HeapLock(he.hHandle)) {
                        PROCESS_HEAP_ENTRY heapEntry;
                        heapEntry.lpData = NULL;
                        while (HeapWalk(he.hHandle, &heapEntry) != FALSE) {
                            if ((heapEntry.wFlags & PROCESS_HEAP_ENTRY_BUSY) &&
heapEntry.cbData > 0)
                            {
                                if (pHeapBase == heapEntry.lpData)
```

```
                                {
                                    //Xor((unsigned char*)heapEntry.lpData,
heapEntry.cbData);
                                }
                            }
                        }
                        if (HeapUnlock(he.hHandle) == FALSE) {
                            printf("HeapUnlock %ld\n", GetLastError());
                        }
                    }
                    he.dwSize = sizeof(HEAPENTRY32);
                } while (Heap32Next(&he));
            }
            heapList.dwSize = sizeof(HEAPLIST32);
        } while (Heap32ListNext(hHeapSnap, &heapList));

        CloseHandle(hHeapSnap);
    }

    return 0;
}

PVOID NTAPI RtlAllocateHeapHook(PVOID HeapHandle, ULONG Flags, SIZE_T Size)
{
    PVOID pAllocated = pRtlAllocateHeap(HeapHandle, Flags, Size);

    HEAP heap;
    heap.pAllocated = pAllocated;
    heap.ulFlags = Flags;
    heap.Size = Size;
    heap.hHeap = HeapHandle;

    return pAllocated;
}

PVOID NTAPI RtlReAllocateHeapHook(PVOID HeapHandle, ULONG Flags, PVOID MemoryPointer,
ULONG Size)
{
    return pRtlReAllocateHeap(HeapHandle, Flags, MemoryPointer, Size);
}

BOOLEAN NTAPI RtlFreeHeapHook(PVOID HeapHandle, ULONG Flags, PVOID BaseAddress)
{
    return pRtlFreeHeap(HeapHandle, Flags, BaseAddress);
}

int main()
{
    MH_STATUS status;

    if (MH_Initialize() != MH_OK) {
        return -1;
```

```
    }

    status = MH_CreateHookApi(
        L"ntdll",
        "RtlAllocateHeap",
        RtlAllocateHeapHook,
        reinterpret_cast<LPVOID*>(&pRtlAllocateHeap)
    );

    status = MH_CreateHookApi(
        L"ntdll",
        "RtlReAllocateHeap",
        RtlReAllocateHeapHook,
        reinterpret_cast<LPVOID*>(&pRtlReAllocateHeap)
    );

    status = MH_CreateHookApi(
        L"ntdll",
        "RtlFreeHeap",
        RtlFreeHeapHook,
        reinterpret_cast<LPVOID*>(&pRtlFreeHeap)
    );

    status = MH_EnableHook(MH_ALL_HOOKS);

    if (status == MH_OK)
    {
        printf("Hooks Enabled!\n");
    }
    else
    {
        printf("Hooks failed to start!\n");
        return -1;
    }

    EncryptHeap();


    return 0;
}
```