Bypassing Windows heap protections

Nicolas Falliere
nicolas.falliere@gmail.com

History

Windows heap-based buffer overflows can be summarized in two categories. The first one covers overflows for Windows 2000, Windows XP and Windows XP SP1 platforms. The heap management code for these systems, located in ntdll.dll, do not perform any sanity check on heap chunks. When an overflow occurs, the next adjacent chunk can be overwritten, and if good values are forged, a subsequent heap operation (alloc, free…) can result in an arbitrary four-byte overwrite in memory. New techniques emerged recently, but the principle remains the same: overwriting a specific portion of memory with specific values, to gain control and execute a payload later on.

The second category includes Windows XP SP2 and Windows 2003 operating systems. Microsoft modified heap structures and heap manipulation functions; two checks on the chunks were added. The first check is to verify the integrity of a security cookie in the chunk header, to ensure no overflow has occurred when this same chunk is allocated; the second check, extremely efficient, verifies the forward and backward link pointers of a free chunk being unlinked, for any reason (allocation, coalescence). The same check is performed for virtually allocated blocks.
Others protections have been introduced as well, mainly PEB randomization, and exception pointers encoding. These protections are there to minimize the amount of fixed and well-known function pointers, used globally by the process. These locations were priviledged targets to exploit a heap overflow the old way.

The first public paper detailing a method to bypass the new heap protections was published at the beginning of year 2005 by Alexander Anisimov. It consists of exploiting the inexistent checks on the lookaside list. The first dword of a lookaside entry is the start of a simply-linked list of chunks, marked as busy, but ready for allocations. When an allocation occurs, the first block of a matching lookaside list may be returned: It is simply removed from the list by replacing the forward link pointer (FLink) in the lookaside entry by the FLink pointer of the newly allocated block. This process is explained in Figure 1.

This new technique is good in theory, but seems pretty hard in practice. The following heap operations must occur, by forging good input values, if we want the N-byte overwrite to happen:

1 – Allocation of a block of size N (<0x3F8 bytes)

2 – Freeing of this block: the block gets referenced in the lookaside table
3 – The overflow occurs in a previous adjacent block: we can manipulate the FLink pointer of the previously freed block
4 – A block of size N is allocated: our fake pointer is written in the lookaside table
5 – A second block of size N is allocated: our fake pointer is returned
6 – A copy operation from a controlled input to this buffer occurs: these bytes are written to our chosen location

As you can see, these conditions can be hard to produce in practice, especially in complex programs. The heap must also have an active and unlocked lookaside table for the operation to succeed.

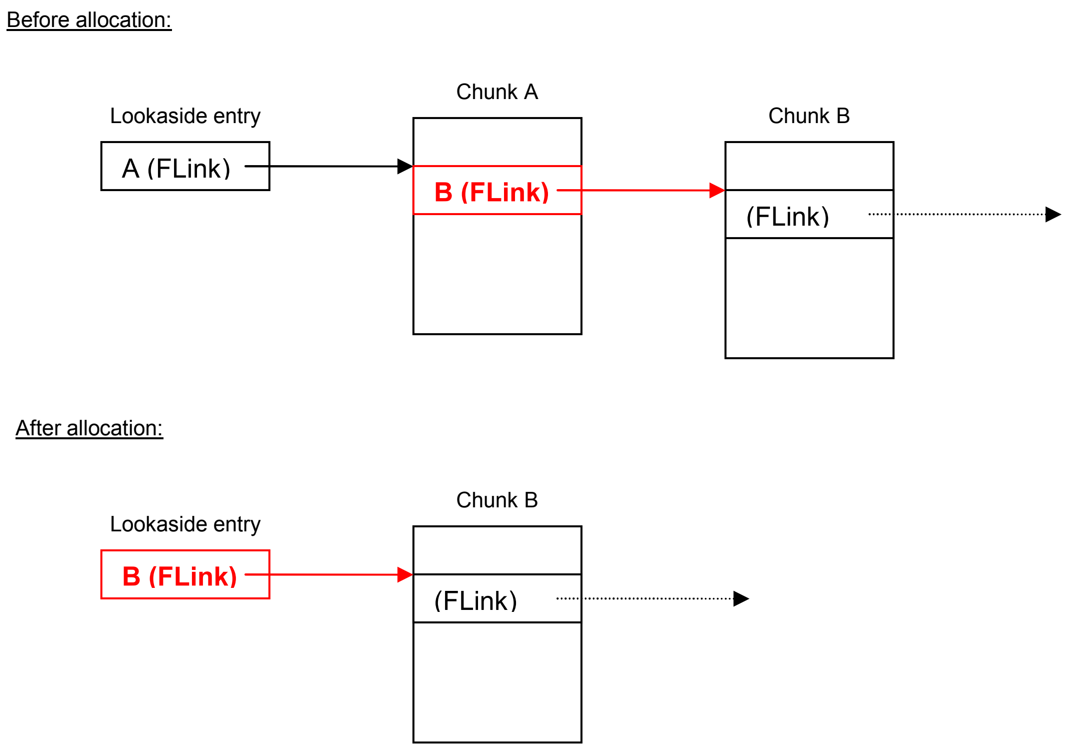

Fig. 1: Allocation of a block A from the lookaside table

A new way to bypass heap protections

The method I introduce here does not use the overwriting of heap-management structures to produce a four-byte overwrite.

The process default heap, as well as others system-created heaps, is used by many APIs to store information concerning the process and its environment. When a DLL is loaded, its main function is executed (DllMain, or similar) and often, data can get stored on the process heap. What if these pieces of data are overwritten?

Let’s take a basic program, such as Windows notepad. We can notice that even this program needs a lot of dynamic libraries to run. If we examine the default heap, before the main thread starts to execute, we’ll notice that a fair amount of heap chunks have been allocated by these DLLs. Many of these chunks have a length of 40 bytes (including 8 bytes for the header) and have the structure described in Figure 2:

Chunk header	
0	X
A	**B**
0	0
?	?

Fig.2: A 40-byte long heap chunk, found in the process default heap

A: Address of the next “40-byte long structure”
B: Address of the previous “40-byte long structure”

It happens that the structure pointed by X is in fact a critical section. When a critical section is initialized, an associated “40-byte long structure” – we will call it a linking structure, is also created to keep track of the critical section. A few of these structures are located in the data section of ntdll.dll; when all of them are used, the linking structures are created in the default heap. Figure 3 shows the relation between linking structures and critical sections.

This doubly-linked list reminds us the way free chunks are handled by heap management routines. During the destruction of a critical section, the associated linking structure will be removed from its list. If we replace A and B, we should then be able to overwrite a 4-byte portion of memory:

From RtlDeleteCriticalSection (ntdll.dll version 5.1.2600.2180):

```
…
mov [eax], ecx        ; eax=B
mov [ecx+4], eax      ; ecx=A
…
```

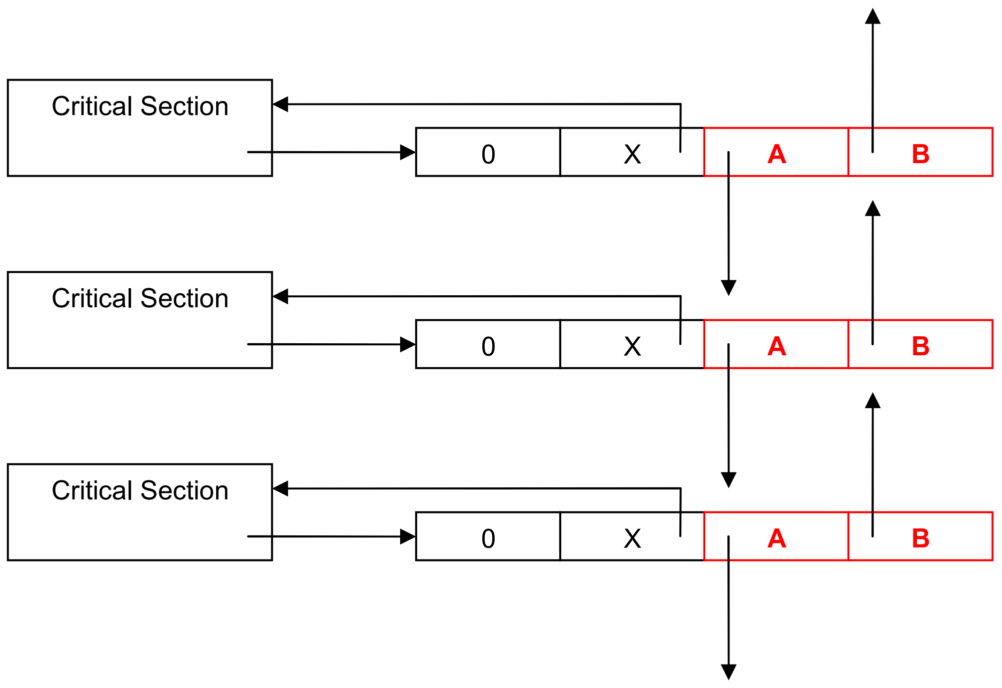

Fig.3: Critical sections and linking structures

The technique works because:

- No sanity checks are performed on these particular backward and forward pointers.
- The critical sections are destroyed during process termination; this ensures the overwriting will occur.
- The linking structures can easily be found in the default heap; if we control the size of the chunk in which we overflow, we can adjust it in such a way that a linking structure lies a few bytes after. Of course, a major drawback is the limitation to this process heap.

You will find a proof-of-concept code demonstrating the technique in Annex 1.

Conclusion

The technique was used to successfully exploit an unpatched heap-overflow located in a standard Windows utility of Windows XP SP2.

However, several problems remain partially solved: Though we have the possibility to overwrite at least 4 bytes of memory, we have to choose good values for the backward and forward pointers. The classic use of pointers to exception handlers is compromised, as well as global function pointers located in the PEB.

Thus, exploiting heap overflows on the newest Windows systems is possible, but the issues now are to increase their portability and their reliability.

Annex 1: Proof-of-Concept code

```
//---------------------------------------------------------------------
// This code demonstrates how an overflow in critical section related
// structures stored in heap chunks can be used to produce an
// arbitrary memory overwrite
// (c) 2005 Nicolas Falliere
// nicolas.falliere@gmail.com
//---------------------------------------------------------------------

#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>


VOID GetChunkList(DWORD *pChunks, INT *nbChunks)
{
    DWORD          pid;
    HANDLE         snapshot;
    HEAPLIST32     list;
    HEAPENTRY32    entry;
    BOOLEAN        bNext;
    INT            cnt = 0;


    pid = GetCurrentProcessId();

    snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST, pid);
    if(snapshot == INVALID_HANDLE_VALUE)
    {
        printf("[Error] Cannot take a heap snapshot\n");
    }
    else
    {
        ZeroMemory(&list, sizeof(list));
        list.dwSize = sizeof(HEAPLIST32);
        bNext = Heap32ListFirst(snapshot, &list);

        while(bNext)
        {
            ZeroMemory(&entry, sizeof(entry));
            entry.dwSize = sizeof(HEAPENTRY32);
            bNext = Heap32First(&entry, list.th32ProcessID,
list.th32HeapID);

            while(bNext)
            {
                pChunks[cnt] = entry.dwAddress;
                cnt++;

                ZeroMemory(&entry, sizeof(entry));
                entry.dwSize = sizeof(HEAPENTRY32);
                bNext = Heap32Next(&entry);
            }

            ZeroMemory(&list, sizeof(list));
            list.dwSize = sizeof(HEAPLIST32);
            bNext = Heap32ListNext(snapshot, &list);
        }

        CloseHandle(snapshot);
        *nbChunks = cnt;
    }
}


int main(void)
{
    HANDLE   hHeap;
    DWORD    pChunks[500];
    INT      nbChunks;
    INT      i;
    HMODULE  hLib;
    DWORD    *p;


    hHeap = GetProcessHeap();
    printf("Default heap: %X\n", hHeap);

    hLib = LoadLibrary("oleaut32.dll");
    printf("LoadLibrary : oleaut32.dll\n");

    GetChunkList(pChunks, &nbChunks);

    for(i = 0; i < nbChunks; i++)
    {
        // Chunk size is 40 bytes
        if(*(WORD *)(pChunks[i] - 8) == 5)
        {
            p = (DWORD *)(pChunks[i]);

            // Check if FLink and BLink are there
            if(p[2] && p[3])
            {
                printf("Structure found at address: %8X\n", p);
                printf("Before modification : A=%8X B=%8X\n", p[2], p[3]);
                memcpy(p + 2, "AAAABBBB", 8);
                printf("After modification  : A=%8X B=%8X\n", p[2], p[3]);
                break;
            }
        }
    }

    printf("Press Enter to terminate the program and trigger the access
violation\n");
    getchar();

    return 0;
}
```