A new way to bypass Windows heap protections

Nicolas Falliere 2005-09-01

Windows heap overflows have become increasingly popular over the last couple of years. Papers like, "Third Generation Exploitation" [ref 1] or, "Windows Heap Overflows" [ref 2] introduced the internal structure and handling mechanisms of Windows heaps, and presented ways to exploit heap-based buffer overflows. Techniques to make highly reliable exploits were presented in the paper, "Reliable Windows Exploits" [ref 3]. Heap exploitation is now mastered for systems such as Windows XP, Windows XP SP1 and Windows 2000.

However, the introduction of Windows 2003 -- and later, Windows XP SP2, brought another level of protection hackers would have to bypass in order to exploit heap overflows on these systems.

In this paper, we'll remind readers of the principles of classic heap overflow exploitation, and explain why these techniques do not work with the newest Windows platforms. Then, we'll present a way to bypass a first level of protection, to trigger a memory overwrite.

A quick overview of Windows heap overflows

A heap is a collection of contiguous chunks of memory, as shown below in Figure 1. When one decides to allocate dynamic memory in a program, the allocation occurs in a heap. Functions like malloc(), GlobalAlloc(), LocalAlloc() or HeapAlloc() are only wrappers of the core function RtlAllocateHeap(); this API, exported by ntdll.dll, is in charge of allocating memory in a heap. Other Rtl*Heap() functions exist as well, to create and destroy heaps, and manipulate chunks.

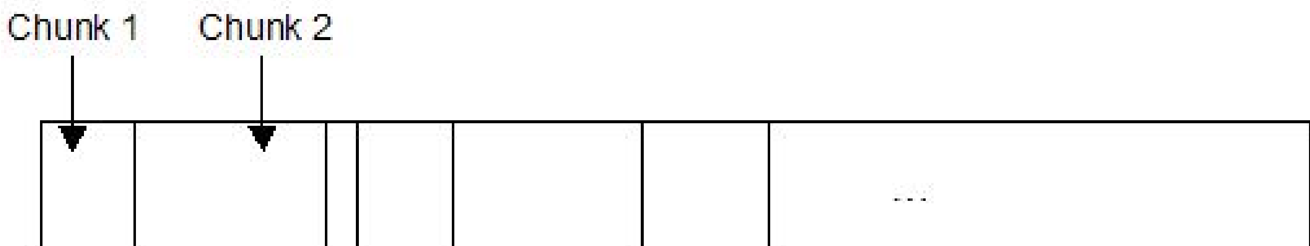


Figure 1. A heap is a collection of contiguous heap chunks.

Each chunk contains a header (shown in Figure 2), detailing its size, the size of the previous block, if it's busy or not, in which memory segment it is located, and so on. This header is usually 8-byte long, and following it begins the real storage area of the chunk. If the chunk is free, two pointers will be concatenated to this classic header, referencing the previous and next (not necessarily adjacent) free blocks of the same size. These pointers are called FLink and BLink, which respectively stand for "Forward" and "Backward" links.

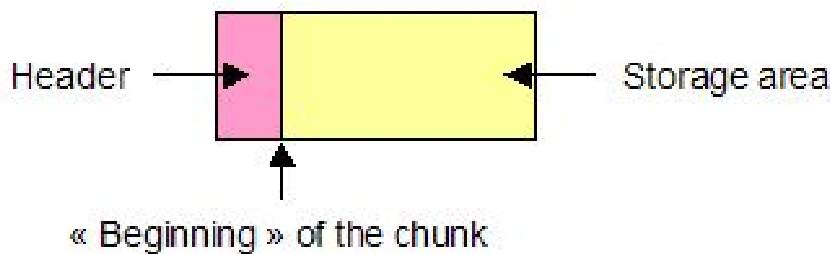


Figure 2. A regular heap-chunk.

When an overflow occurs in a chunk, the header structure of the next adjacent chunk is overwritten. By forging "malicious" values, a subsequent heap operation can trigger an arbitrary 4-byte memory overwrite. Of course, this is not the result of voodoo magic, but simply the exploitation of a heap mechanism called "unlinking".

Several exploitation scenarios exist; the author will remind readers only of the simplest one. Suppose we overflow the contents of a free block. When this block is allocated, it will be removed from its doubly-linked list; this process takes place in two steps. First, the FLink pointer of the previous chunk will be updated to reference the next chunk; then, the BLink pointer of the next chunk will be updated to reference the previous chunk. This is the unlinking process, which is achieved in two assembly instructions:

```
mov [reg1], reg2         ; reg1=FLink
mov [reg2+4], reg1       ; reg2=BLink
```

Thus, forging FLink and BLink will lead to a 4-byte memory overwrite. Gaining control is the next step; please see the References section to learn more about this.

Introducing heap protections

These techniques worked well with Windows XP (SP0, SP1) and Windows 2000 operating systems. (Un)fortunately, things changed with the arrival of Windows 2003. Microsoft modified heap management routines and heap structures in order to check the validity of a chunk before allocating or freeing it.

- A security cookie was introduced in chunk headers. When the chunk is allocated, this cookie is checked to ensure no overflow has occurred.
- Forward and backward link pointers are verified, before the unlinking process happens, for any reason (allocation, coalescence). The same check is performed for virtually allocated blocks. This check is the real obstacle one has to face to exploit a heap overflow.

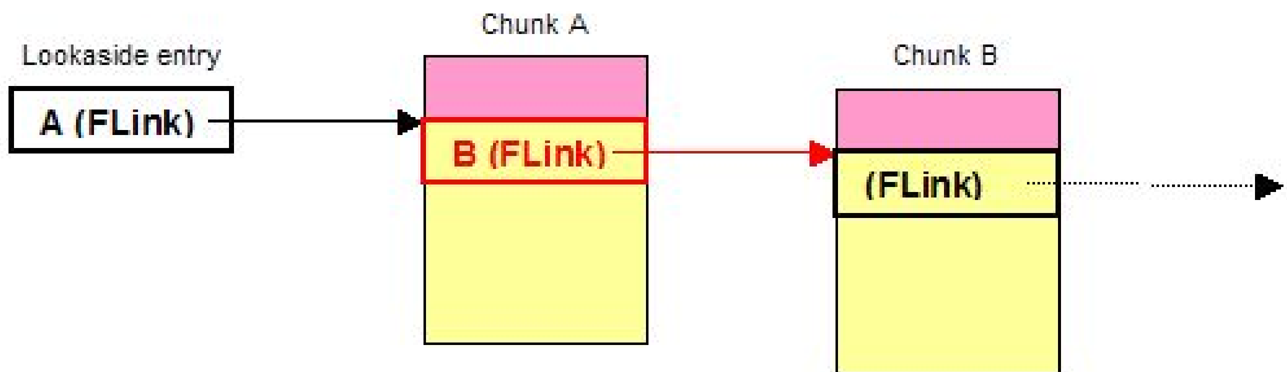
Others protections have been introduced as well, mainly PEB randomization, and exception pointers encoding. The goal is to minimize the amount of fixed and well-known function

pointers, used globally by the process. These locations were priviledged targets to exploit a heap overflow the old way.

The protection was flawed

Unfortunately, the protection was not 100% heap-overflow-proof, as Alexander Anisimov showed at the beginning of 2005.

This first public method to bypass the new heap protections consists of exploiting the inexistent checks on the lookaside list (refer to the paper, "Defeating Windows XP SP2 Heap protection and DEP bypass" [ref 4] if you want to learn more about lookaside lists). 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 3.



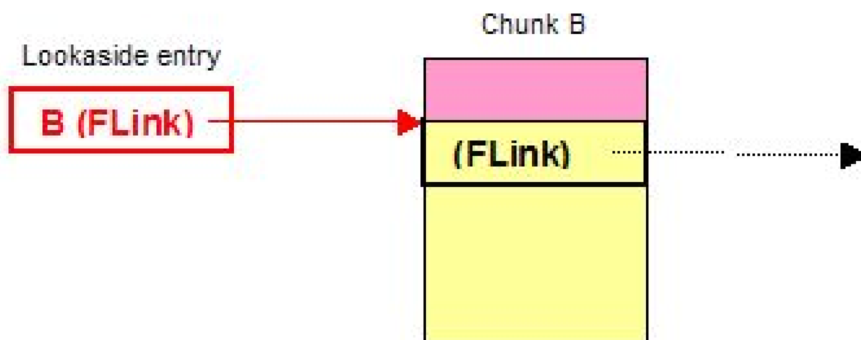


Figure 3. Allocation of a block A from the lookside table..

This new technique is good in theory, but practically, it is hard to use. 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.

Another way to bypass heap protections

This method presents a way to overwrite at least 4 bytes of memory, by overflowing special structures stored in the process default heap.

The process default heap, as well as others system-created heaps, is used by many Windows APIs to store information concerning the process and its environment. When a dynamically-linked library (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?

The fact that even the simplest program, like Widows Notepad, needs so many libraries to run is particularly interesting. If we examine the default heap, before the main thread even starts to execute, we'll notice that a fair amount of heap chunks have been allocated. A quick look at the default heap reveals that many of these chunks have a length of 40 bytes (including 8 bytes for the header) and have the structure described in Figure 4:

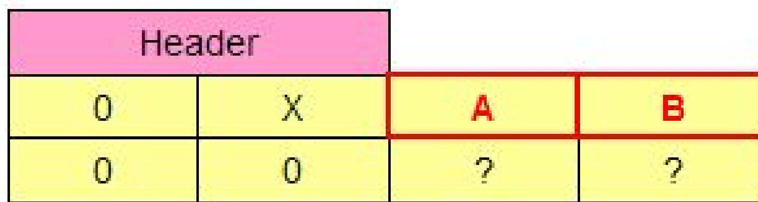


Figure 4. A 40-byte long heap chunk, found in the process default heap.

Where: A is the Address of the next "40-byte long structure". B is the Address of the previous "40-byte long structure".

Note: If you create the process with a debugger, these structures will be 56-bytes long. By default, a heap trailer of 16 bytes is added by the system when the process is created with the "DEBUG_PROCESS" flag.

The first noticeable thing is that A and B play the roles of backward and forward pointers. It also 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 5 shows how all critical sections of a process are linked together.

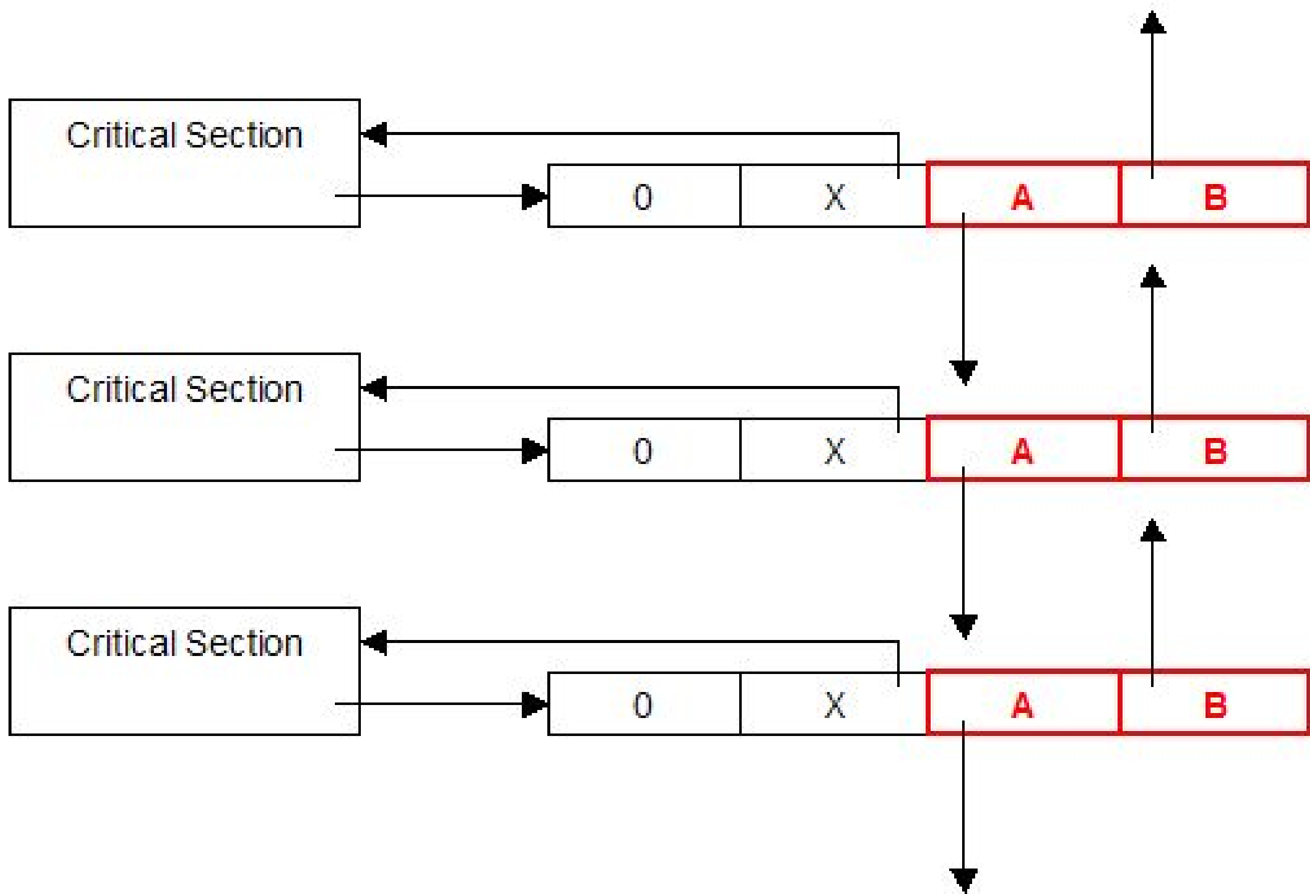


Figure 5. Critical sections and linking structures.

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. And in fact, we can easily find the code in charge of the unlinking process (with a debugger, just replace A and B by invalid addresses, and destroy the critical section to trigger a memory access violation exception).

The following assembly lines are executed by RtlDeleteCriticalSection (ntdll.dll version 5.1.2600.2180):

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

These lines probably remind you what we've discussed earlier. And in fact, the principle is the one of classic heap overflow exploitation: if we can't use the chunk pointers anymore, let's use the pointers of another linked list! We are very lucky here because these structures are very common in the process heap, and absolutely no sanity checks are performed on them. Moreover, critical sections are often destroyed during process termination, which ensures that the overwriting will occur.

The upcoming issue: gaining control

Overwriting memory is the first step of heap-based buffer overflows exploitation; the second and third steps are to choose which value (A) and place (B) to overwrite.

The old couples (fixed "CALL" instruction, Exception handler pointer) and (Pointer to payload, PEB function pointer) do not work anymore, mainly because of memory protections introduced with the Service Pack 2 of Windows XP:

- Vector exception handlers and the final handler pointers, though located at fixed positions for a given version of kernel32.dll, are no longer useful, because encoded using RtlEncodePointer function of ntdll.dll. The real address is "xor-ed" with a system-generated value, using NtQueryInformationProcess.
- The location of the Process Environment Block is randomized, which diminishes the chances of success if we try to overwrite one of its global and often-called function pointers, like AcquireFastPebLock or ReleaseFastPebLock.

These protections were also ported in the Service Pack 1 of Windows 2003 -- Windows 2003 SP0 only implemented heap protections. Therefore, new places remain to be found in order to produce reliable exploits.

Conclusion

The major drawback is that critical sections are often destroyed at process termination, which would force a potential exploiter to crash the program in order to trigger the overwrite. Moreover, the overflow must occur in the process default heap, and a minimal flexibility is required to overwrite a linking structure.

Nonetheless, producing a memory overwrite on the newest Windows systems is possible, and this is the first step towards exploitation of heap overflows.

Proof of concept

The author has provided a proof-of-concept to demonstrate an implementation of his technique.

References

[ref 1] Halvar Flake
"Third Generation Exploitation"
http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt

[ref 2] David Litchfield
"Windows Heap Overflows"
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

[ref 3] Matt Conover, Oded Horowitz
"Reliable Windows Exploits"
http://cansecwest.com/csw04/csw04-Oded+Connover.ppt

[ref 4] Alexander Anisimov
"Defeating Windows XP SP2 Heap protection and DEP bypass"
http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

Privacy Statement
Copyright 2006, SecurityFocus