



Windows Heap Overflows



David Litchfield <david@ngssoftware.com>

NGSSoftware

Intelligent solutions
for an evolving world.

Introduction

This presentation will examine how to exploit heap based buffer overflows on the Windows platform. Heap overflows have been well documented on *nix platforms, for example Matt Connover's paper – w00w00 on heap overflows, but they've not been well documented on Windows, though Halvar Flake's Third Generation Exploits paper covered the key concepts.

NGSSoftware

Intelligent solutions
for an evolving world.

Heap based buffers are safe.... ??????

Most developers are aware of the dangers of stack based buffer overflows but too many still believe that if a heap based buffer is overflowed it's not too much of a problem.

One paper on secure coding suggested that to solve the problem of stack based overflows was to move the buffer to the heap!

NGSSoftware

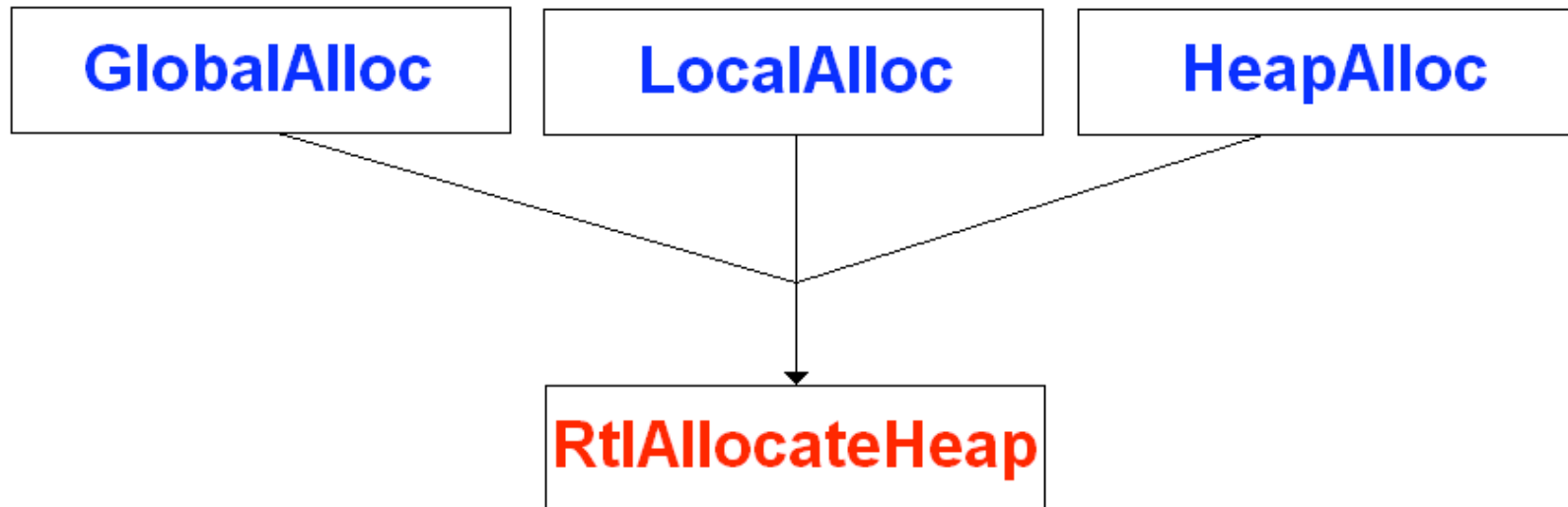
Intelligent solutions
for an evolving world.

What is a heap?

The heap is an area of memory used for storage of dynamic data. Every process has a default process heap but a developer can create their own private heaps. Space is allocated from the heap and freed when finished with.

Windows Heap Overflows

Heap functions



NGSSoftware

Intelligent solutions
for an evolving world.

Heap Design

Each heap starts with a structure. This structure, amongst other data, contains an array of 128 `LIST_ENTRY` structures. Each `LIST_ENTRY` structure contains two pointers – see `winnt.h`. This array can be found at `0x178` bytes into the heap structure – call it the `FreeList` array.

Heap Design

When a heap is first created there are two pointers that point to the first free block set in `FreeList[0]`. Assuming the heap base address is `0x00350000` then first available block can be found at `0x00350688`.

Heap Design

0x00350178 (FreeList[0].Flink) = 0x00350688 (First Free Block)

0x0035017C (FreeList[0].Blink) = 0x00350688 (First Free Block)

0x00350688 (First Free Block) = 0x00350178 (FreeList[0])

0x0035068C (First Free Block+4) = 0x00350178 (FreeList[0])

When an allocation occurs these pointers are updated accordingly. As more allocations and frees occur these pointers are continually updated and in this fashion allocated blocks are tracked in a doubly linked list.

NGSSoftware

Intelligent solutions
for an evolving world.

So where's the problem?

When a heap based buffer is overflowed the control information is overwritten so when the buffer (allocated block) is freed and it comes to updating the pointers in the FreeList array there's going to be an access violation.

So where's the problem?

Example: See code listing A – heap.c

NGSSoftware

Intelligent solutions
for an evolving world.

So where's the problem?

Access violation

```
77F6256F mov dword ptr [ecx],eax  
77F62571 mov dword ptr [eax+4],ecx
```

EAX = 0x42424242

ECX = 0x42424242

If we own both EAX and ECX we have an arbitrary DWORD overwrite. We can overwrite the data at any 32bit address with a 32bit value of our choosing.

NGSSoftware

Intelligent solutions
for an evolving world.

Exploiting Heap Overflows

Repairing the Heap

Unhandled Exception Filter

PEB function Pointer

Vectored Exception Handling

Thread Environment Block

NGSSoftware

Intelligent solutions
for an evolving world.

Repairing the heap

After the overflow the heap is corrupt so you'll need to repair the heap.

Many of the Windows API calls use the default process heap and if this is corrupt the exploit will access violate.

Repairing the heap

Could repair on a per vulnerability/exploit basis.
Time consuming and could run into problems.

Need a generic way to repair the heap which is effective for all exploits. Write it once and reuse it.

Repairing the heap

The best method for repairing the heap is to reset the heap making it “appear” as if it is a fresh new heap. This will keep other heap data intact but allow fresh allocations.

Repairing the heap

We reset our overflow heap control structure with `heap.TotalFreeSize` and set the flags to `0x14` then set `heap.FreeLists[0].Flink` and `heap.FreeLists[0].Blink` to the start of the fake control structure.

See code listing B – `asm-repair-heap`.

Exploit: Using the Unhandled Exception Filter

The Unhandled Exception Filter method is the most common method used. The UEF is the “last ditch effort” exception handler.

Exploit: Using the Unhandled Exception Filter

Location varies from OS to OS and SP to SP.
Disassemble the SetUnhandledExceptionFilter
function.

```
77E7E5A1      mov ecx,dword ptr [esp+4]
77E7E5A5      mov eax,[77ED73B4]
77E7E5AA      mov dword ptr ds:[77ED73B4h],ecx
77E7E5B0      ret 4
```

UEF = 0x77ED73B4

NGSSoftware

Intelligent solutions
for an evolving world.

Exploit: Using the Unhandled Exception Filter

When an unhandled exception occurs the following block of code is executed:

```
77E93114      mov eax,[77ED73B4]
77E93119      cmp eax,esi
77E9311B      je 77E93132
77E9311D      push edi ***
77E9311E      call eax
```

Exploit: Using the Unhandled Exception Filter

Essence of the method is to set our own Unhandled Exception Filter.

EDI was pushed onto the stack. `0x78` bytes past EDI is a pointer to the end of the buffer – just before the heap management control stuff.

Exploit: Using the Unhandled Exception Filter

Set the UEF to an address that points to a

```
CALL DWORD PTR [EDI + 0x78]
```

Many can be found in netapi32.dll, user32.dll, rpcrt4.dll for example.

Exploit: Using the Unhandled Exception Filter

Notes: Other OSes may not use EDI. Windows 2000 for example has a pointer at `ESI+0x4C` and `EBP+0x74`.

Using this method you need to know the target system – i.e. what OS and what SP level.

Exploit: Using the Unhandled Exception Filter

Example: See code listing C – heap-uef.c and code listing D - exploit-uef.c

Exploit: Using Vectored Exception Handling

Vectored Exception Handling is new as of Windows XP.

Unlike traditional frame based exception handling where **EXCEPTION_REGISTRATION** structures are stored on the stack information about VEH is stored on the heap.

Exploit: Using Vectored Exception Handling

A pointer to the first Vectored Exception Handler is stored at `0x77FC3210`. Points to a `_VECTORED_EXCEPTION_NODE`.

Exploit: Using Vectored Exception Handling

```
struct _VECTORED_EXCEPTION_NODE
{
    DWORD   m_pNextNode;
    DWORD   m_pPreviousNode;
    PVOID   m_pfnVectoredHandler;
}
```

Exploit: Using Vectored Exception Handling

Vectored handlers are called *before* any frame based handlers! Technique involves overwriting the pointer to the first `_VECTORED_EXCEPTION_NODE` @ `0x77FC3210` with a pointer to a fake VE node.

Exploit: Using Vectored Exception Handling

```
77F7F49E      mov     esi,dword ptr ds:[77FC3210h]
77F7F4A4      jmp     77F7F4B4
77F7F4A6      lea    eax,[ebp-8]
77F7F4A9      push   eax
77F7F4AA      call   dword ptr [esi+8]
77F7F4AD      cmp    eax,0FFh
77F7F4B0      je     77F7F4CC
77F7F4B2      mov    esi,dword ptr [esi]
77F7F4B4      cmp    esi,edi
77F7F4B6      jne    77F7F4A6
```

The code behind calling the vectored exception handler.

NGSSoftware

Intelligent solutions
for an evolving world.

Exploit: Using Vectored Exception Handling

Need to find a pointer on the stack to our buffer. Assume it can be found at `0x0012FF50`. This becomes our `m_pfnVectoredHandler` making the address of our pseudo `_VECTORED_EXCEPTION_NODE` `0x0012FF48`.

Exploit: Using Vectored Exception Handling

Remember on the free we get an arbitrary
DWORD overwrite:

```
77F6256F mov dword ptr [ecx],eax  
77F62571 mov dword ptr [eax+4],ecx
```

We set EAX to **0x77FC320C** and ECX to
0x0012FF48.

NGSSoftware

Intelligent solutions
for an evolving world.

Exploit: Using Vectored Exception Handling

0x77FC320C is moved into 0x0012FF48 then 0x0012FF48 is moved into 0x77FC3210 – thus our pointer is set. When an exception occurs 0x0012FF48 (our pseudo VEN) is moved into ESI and DWORD PTR[ESI+8] is called. ESI+8 is a pointer to our buffer.

Exploit: Using Vectored Exception Handling

Notes: If the location of the stack (and thus the pointer to the buffer) moves this method can be unreliable.

Example: See code listing E – heap-vector.c and F – exploit-vector.c

Exploit: RtlEnterCriticalSection pointer in the PEB

Each process contains a structure known as the PROCESS ENVIRONMENT BLOCK or PEB. The PEB can be referenced from the Thread Information/Environment Block TIB/TEB. **FS:[0]** points to the TEB.

```
mov eax, dword ptr fs:[0x30]  
mov eax, dword ptr fs:[eax+0x18]
```

Exploit: RtlEnterCriticalSection pointer in the PEB

As well as containing other process specific data the PEB contains some pointers to RtlEnterCriticalSection and RtlLeaveCriticalSection. These pointers are referenced from RtlAcquirePebLock and RtlReleasePebLock. RtlAcquirePebLock is called from ExitProcess for example.

Exploit: RtlEnterCriticalSection pointer in the PEB

The location of the PEB is stable across Windows NT 4 / 2000 / XP and thus the pointer to RtlEnterCriticalSection can be found at 0x7FFDF020. Whilst the PEB can be found at the same address in Windows 2003 the function pointers are no longer present so this method won't work with 2003.

Exploit: RtlEnterCriticalSection pointer in the PEB

The method simply involves overwriting the pointer to RtlEnterCriticalSection in the PEB with the address of an instruction that will return to the buffer.

Example: See code listing G – heap-peb.c and H – exploit-peb.c

Exploit: TEB Exception Handler Pointer

Each Thread Environment Block contains a pointer to the first frame based exception handler. The first thread's TEB has a base address of 0x7FFDE000 and each new thread's TEB is assigned an address growing towards 0x00000000. If a thread exits and a new thread is created then it will get the address of the previous thread's TEB.

Exploit: TEB Exception Handler Pointer

This can lead to a “messy” TEB table and can make this method uncertain.

However, if the address of the vulnerable thread’s TEB is stable then this method can be used quite effectively.

Exploit: TEB Exception Handler Pointer

The method involves overwriting the pointer to the first exception handler in the TEB with an address that points to an instruction that will get path of execution back to the buffer.

Exploit: Getting Creative!

There are other ways to exploit heap based buffer overflows to execute arbitrary code to defeat mechanisms such as marking the heap as non-executable.

Exploit: Getting Creative!

Assume we have a process with the heap marked as non-executable. This can be defeated with pointer subversion.

An example of this can be found in the fault reporting functionality of the `UnhandledExceptionFilter()` function.

Exploit: Getting Creative!

The fault reporting code calls `GetSystemDirectoryW()` to which "faultrep.dll" is concatenated. This library is then loaded and the `ReportFault()` function is called.

Exploit: Getting Creative!

`GetSystemDirectoryW()` references a pointer in the .data section of kernel32.dll that points to where the wide character string of the Windows system directory can be found. This pointer can be found at `0x77ED73BC`. On overflow we can set this pointer to our own system directory.

Exploit: Getting Creative!

Thus when `GetSystemDirectoryW()` is called the “system” directory is a directory owned by the attacker – this can even be a UNC path. The attacker would create their own `faultrep.dll` which exports a `ReportFault()` function and so when the `UnhandledExceptionFilter()` function is called arbitrary code can be executed.

Exploit: Getting Creative!

Whilst code paths are finite I'd argue that the possibilities of what can be done is limited more by the imagination.

NGSSoftware

Intelligent solutions
for an evolving world.

Conclusion

Hopefully this presentation has demonstrated the dangers of heap based buffer overflows and that developers not treat them as benign.

Any questions?

NGSSoftware

Intelligent solutions
for an evolving world.

Thanks for coming!

Thanks for coming and enjoy the rest of the conference!

NGSSoftware

Intelligent solutions
for an evolving world.