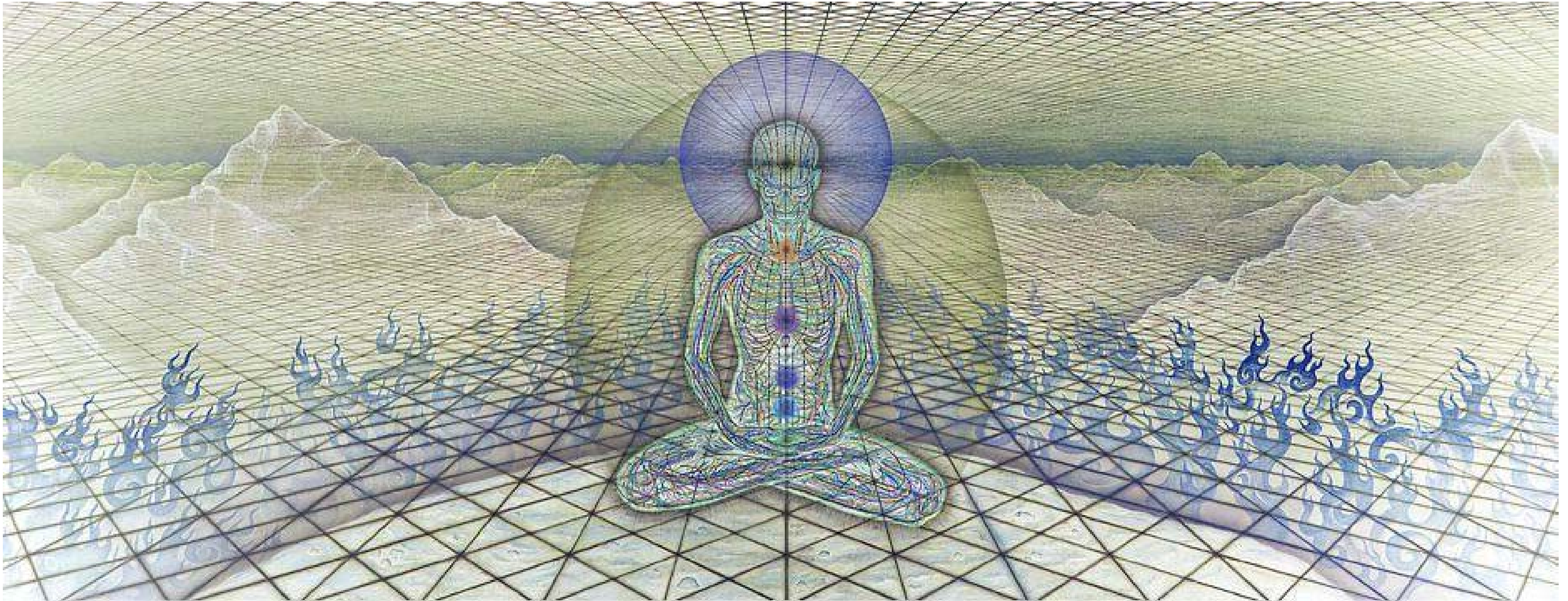
Hawkes

Attacking the Vista Heap



Theologue (inverted)
Copyright Alex Grey, 1984

Intro-clusion

- Heap exploits are harder than ever
- Application specific attacks are the future

- But complex heap implementation attacks should still be considered

The Heap

- “Heap” == dynamic memory allocation
- Runtime memory management
- Using API
  - HeapCreate
  - HeapAlloc
  - HeapFree

Heap API

HANDLE hHeap HeapCreate(…);

LPVOID HeapAlloc(HANDLE hHeap, …,
SIZE_T dwSize)

BOOL HeapFree(HANDLE *hHeap*, …,
LPVOID *lpMem*)

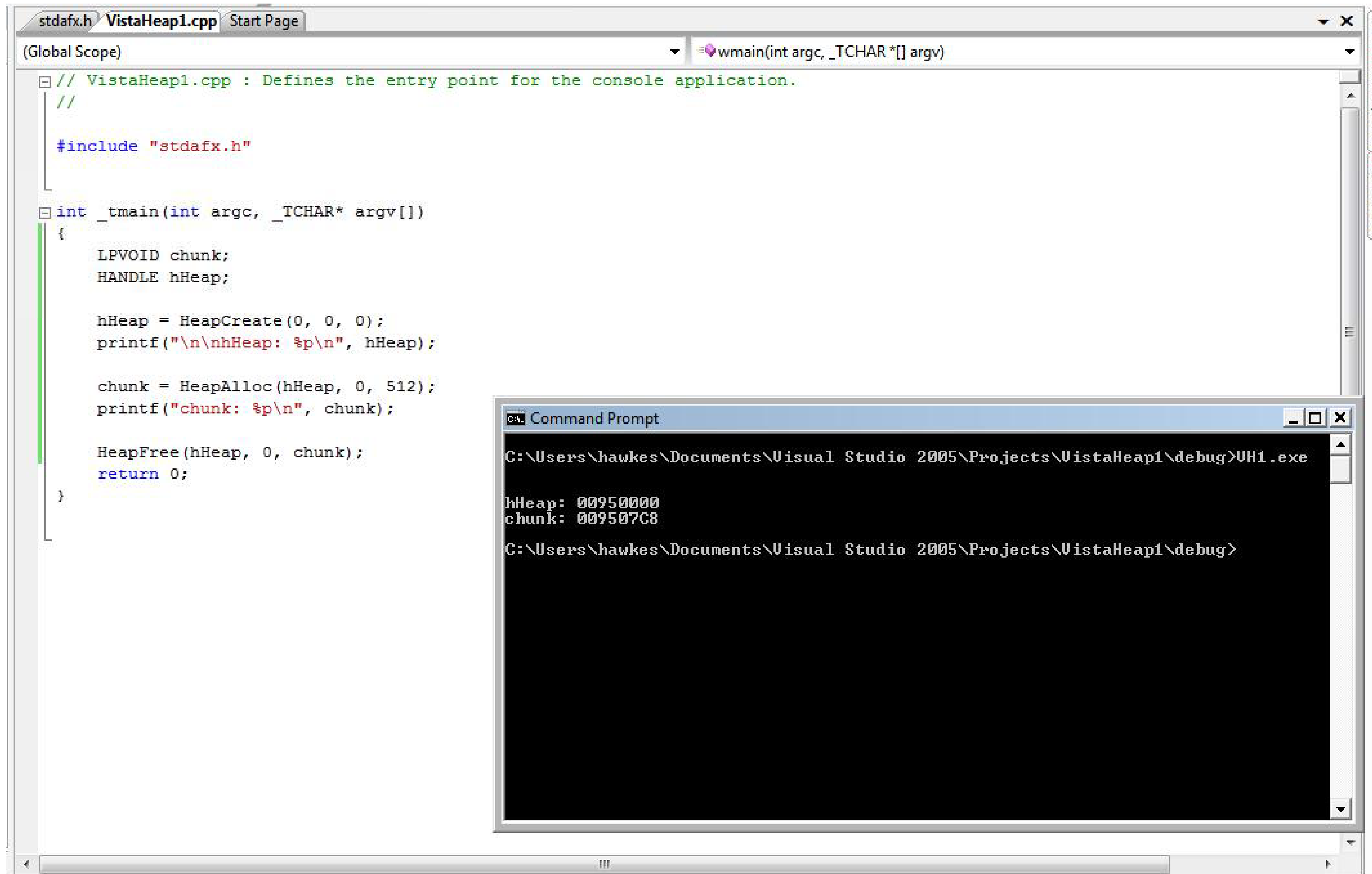
Heap API

HANDLE hHeap HeapCreate(…);

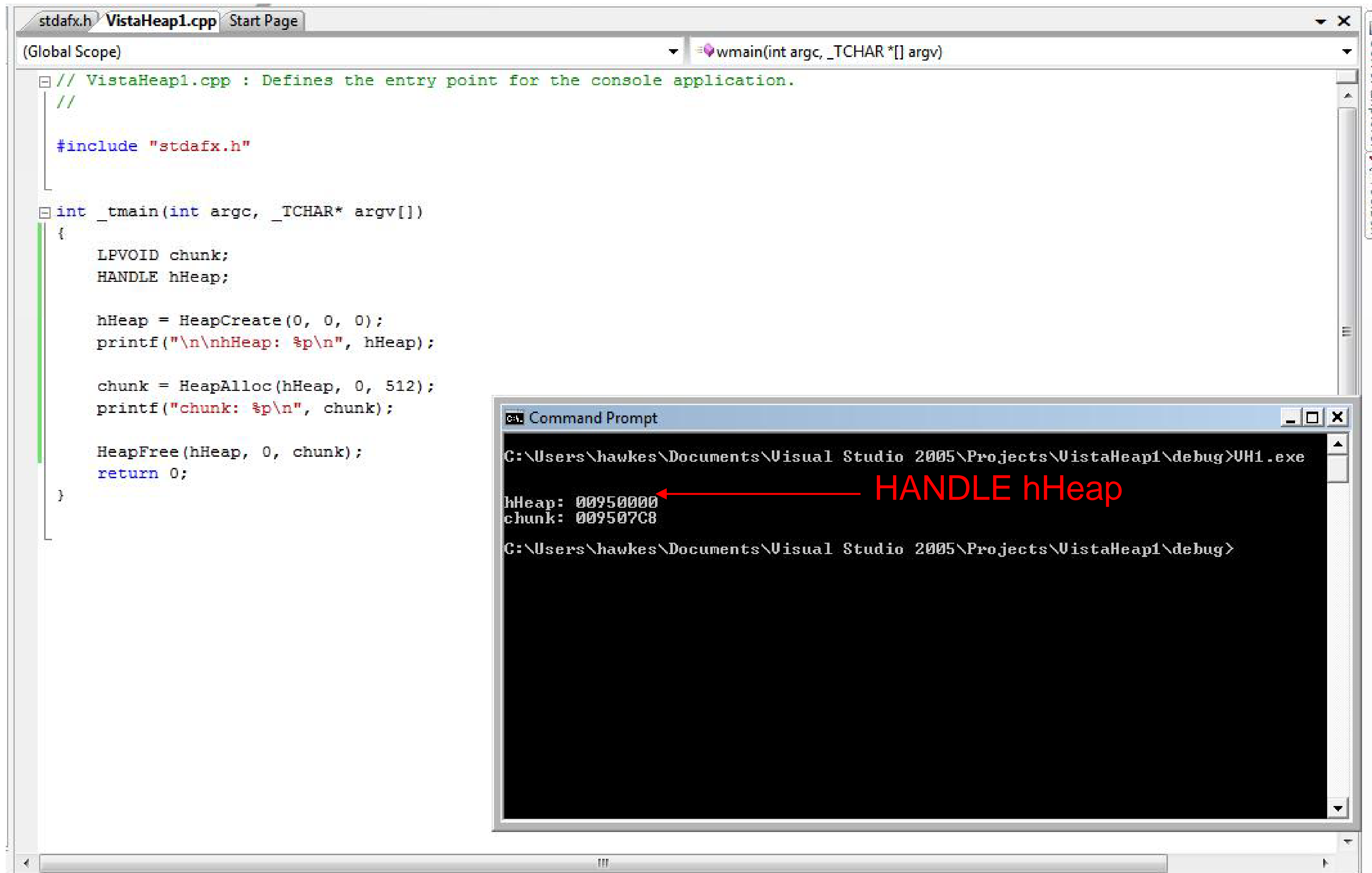
LPVOID HeapAlloc(HANDLE hHeap, …,
SIZE_T dwSize)

BOOL HeapFree(HANDLE hHeap, …,
LPVOID lpMem)

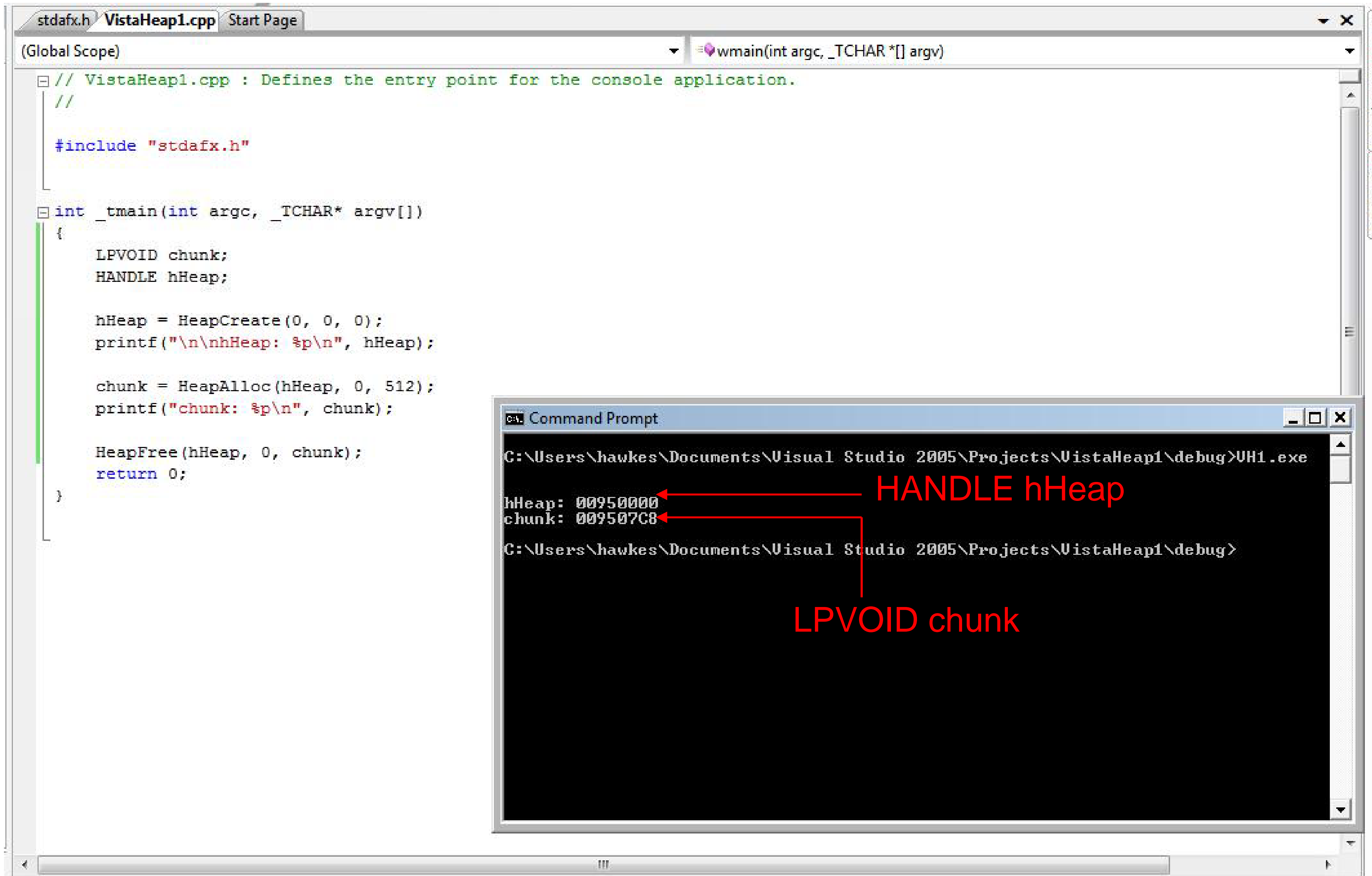
Heap API



Heap API

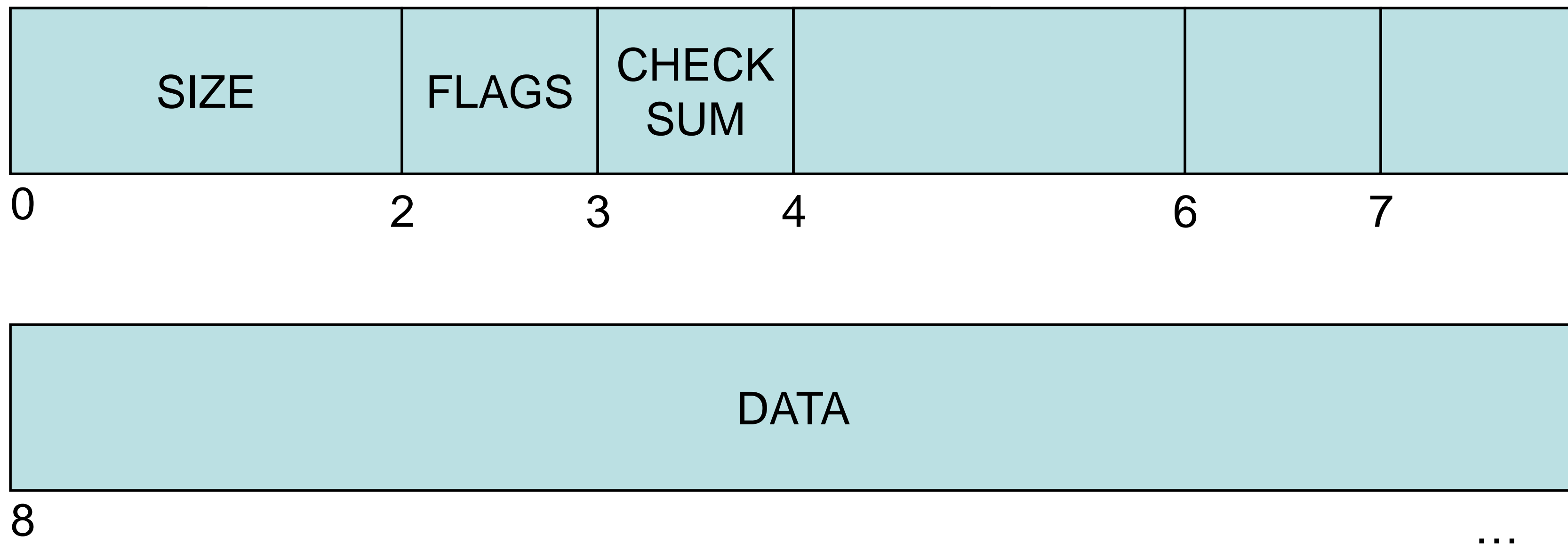


Heap API



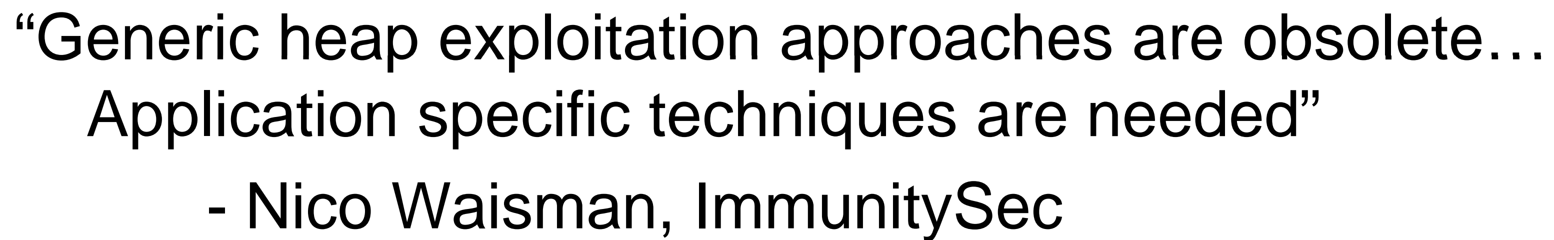
Vista Internals

- Every chunk has a header
- 8 bytes, called HEAP_ENTRY



Vista Changes

- Encoded heap entry headers
- Checksum in headers
- Randomized heap base
- List integrity checks
- Opt-in for failing on corruption


“Generic heap exploitation approaches are obsolete…
Application specific techniques are needed”
- Nico Waisman, ImmunitySec

Applications

- Overflow the target applications heap data
- Target important structures
- Ensure that they are allocated after overflow chunk

Arbitrary Free

- Overflow into a HeapAlloc pointer X
- Application will HeapFree X at some point
- So…

Arbitrary Free

1. Attacker sets X to point to chunk Y, where Y is an important chunk for the application
2. Attacker triggers HeapFree on X
3. Chunk Y is freed, application still using it
4. Attacker triggers allocation of size(Y)
5. Allocator returns Y (say into variable Z)
6. Attacker makes application use Z to overwrite Y

Arbitrary Free

Arbitrary Free requirements:

- Control the X pointer
- Know the address of the Y chunk (partial overwrite, info leak, heap spray)
- Contain any deallocation corruption to Y
- Sufficient control of Z usage
- Ability to leverage control of Y

Adjusted Double Free

- Application specific double free attacks
- As opposed to UNLINK double free
- Order of free/allocation pattern changes
- Traditionally: free free alloc write alloc
- Adjusted: free alloc free alloc write
  (Which is not always possible)

Adjusted Double Free

free alloc free alloc write

1. Free chunk X
2. Before second free, allocate X for application, into Y
3. Free chunk X… which now releases Y
4. Allocate X for application, into Z

Adjusted Double Free

- At this point: Application has Y and Z, both with equal address X
- But used for different purposes, so…
- Make either Y or Z hold some important structure
- And ensure the other is attacker controlled
- Writing into this chunk changes important structure

Adjusted Double Free

- Devil is in the application specific details

- Local vs global double free, only a subset is ever exploitable
- Important structure usually must be initialized before being overwritten

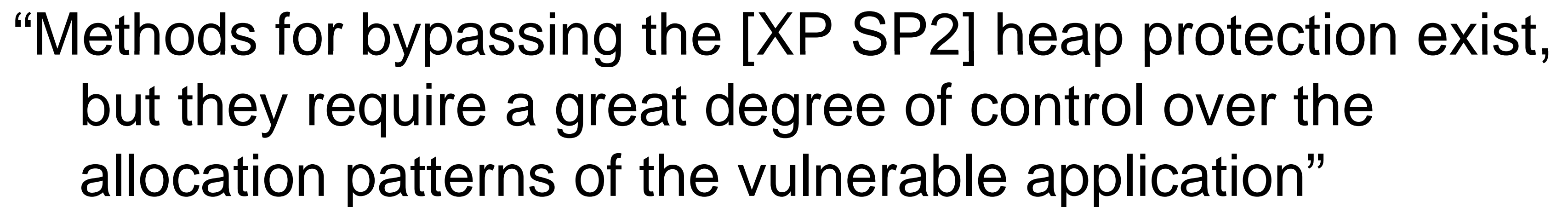
Adjusted Double Free

Adjusted Double Free requirements:

- Double free with interleaved allocation
- While also giving a meaningful allocation
- Sufficient control of one chunks usage
- Ability to leverage control of the other

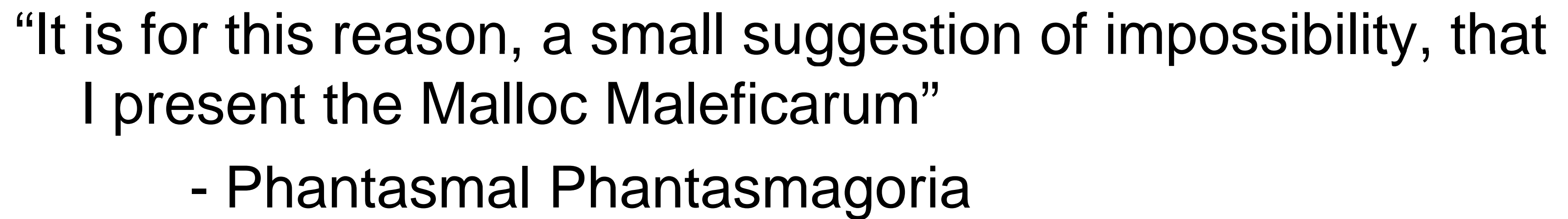
Bonus:

- ASLR doesn’t matter



- Alexander Sotirov, Determina

"I can make strawberry pudding with so many prerequisites"
- Sinan Eren, ImmunitySec


“It is for this reason, a small suggestion of impossibility, that I present the Malloc Maleficarum”
- Phantasmal Phantasmagoria

ASLR

HeapCreate:

```
1   randPad = (RtlpHeapGenerateRandomValue64() & 0x1F) << 16;
    totalSize = dwMaximumSize + randPad;
    …
2   NtAllocateVirtualMemory(INVALID_HANDLE_VALUE, &allocAddr, 0,
                &totalSize, MEM_RESERVE, rwProt);
    …
3   RtlpSecMemFreeVirtualMemory(INVALID_HANDLE_VALUE, &allocAddr,
                &randPad, MEM_RELEASE);
    …
4   hHeap = (HANDLE) allocAddr + randPad;
```

Segment Allocation

RtlpExtendHeap:

```
1   NtAllocateVirtualMemory(INVALID_HANDLE_VALUE, &allocAddr, 0,
                &hHeap->segmentReserve, MEM_RESERVE, rwProt);
    …
2   NtAllocateVirtualMemory(INVALID_HANDLE_VALUE, &allocAddr, 0,
                &segmentCommit, MEM_COMMIT, rwProt);
    …
3   return allocAddr;
```

Large Chunk Allocation

RtlpAllocateHeap (large chunk):

```
1     dwSize += BASE_STRUCT_SIZE;
      …
2     NtAllocateVirtualMemory(INVALID_HANDLE_VALUE, &baseAddr, 0,
                  &dwSize, MEM_COMMIT, rwProt);
      …
      hHeap->largeTotal += dwSize;
      …
3     chunk = (LPVOID) baseAddr + BASE_STRUCT_SIZE + HEAP_ENTRY_SIZE;
      …
      return chunk;
```

Heap Spray I

- Heap base randomized, segments and large chunks not
- Linearly allocated in first available region
- But still affected by random heap base

- Heap spray used to position data statically
  - Spray small chunks within a single heap
  - Or allocate large chunk(s)

Heap Spray II – the stats

- Say NtAllocateVirtualMemory gives consecutive allocations X
- Every heap base can lie anywhere from X to X + 0x1F0000 (~2MB range)
- Segment reserve size ~ 16MB
- Large chunk >= 512KB

Heap Spray III – the theory

- For target application, find average Y of last reserved page across all heaps
- Y = function of the amount of committed and reserved heap pages, with variability approaching 2MB (more when early)
- Spray amount Z (segments or large chunk), with Z > ~16MB
- Y + (Z/2) => your data w/ probability ~= 1

(who cares really)

Guarding hHeap

- Notice lack of guard pages
- Consider a heap spray filling the entire 32-bit address space (<2GB)
- Segments will readjust size to fill smaller holes
- Left with: reserved holes in early heap space, followed by large contiguous writable block of committed memory

hHeap overflows I

- Overflow in contiguous space can overwrite… potentially everything
  - Application data from different heaps
  - Segment and chunk headers
  - hHeap HANDLEs

hHeap overflows I

- Overflow in contiguous space can overwrite… potentially everything
  - Application data from different heaps
  - Segment and chunk headers
  - hHeap HANDLEs

hHeap overflows II

- hHeap HANDLE is ridiculously important
- Central management structure for each individual heap
  - Free lists
  - Heap canary
  - Flags and tunable options
  - Etc…
- Returned from HeapCreate

hHeap overflows III

- Assume can overflow hHeap at location X
- Crafted payload of 212 bytes relative to X
- Results in arbitrary code execution on next HeapAlloc from this heap

- Not WRITE4 primitive, direct control of EIP

hHeap overflows IV

- Goal: get overflow chunk positioned before some hHeap HANDLE
- Align and repeat payload on each page, padding where necessary

hHeap overflows V

- Pattern 1:
  - Spray some fixed amount X
  - Trigger creation of new heap in application
  - Spray remaining address space
  - Overflow from initial heap spray area X (may need to free some of X first, to make room for overflow chunk)
  - Trigger allocation on new heap

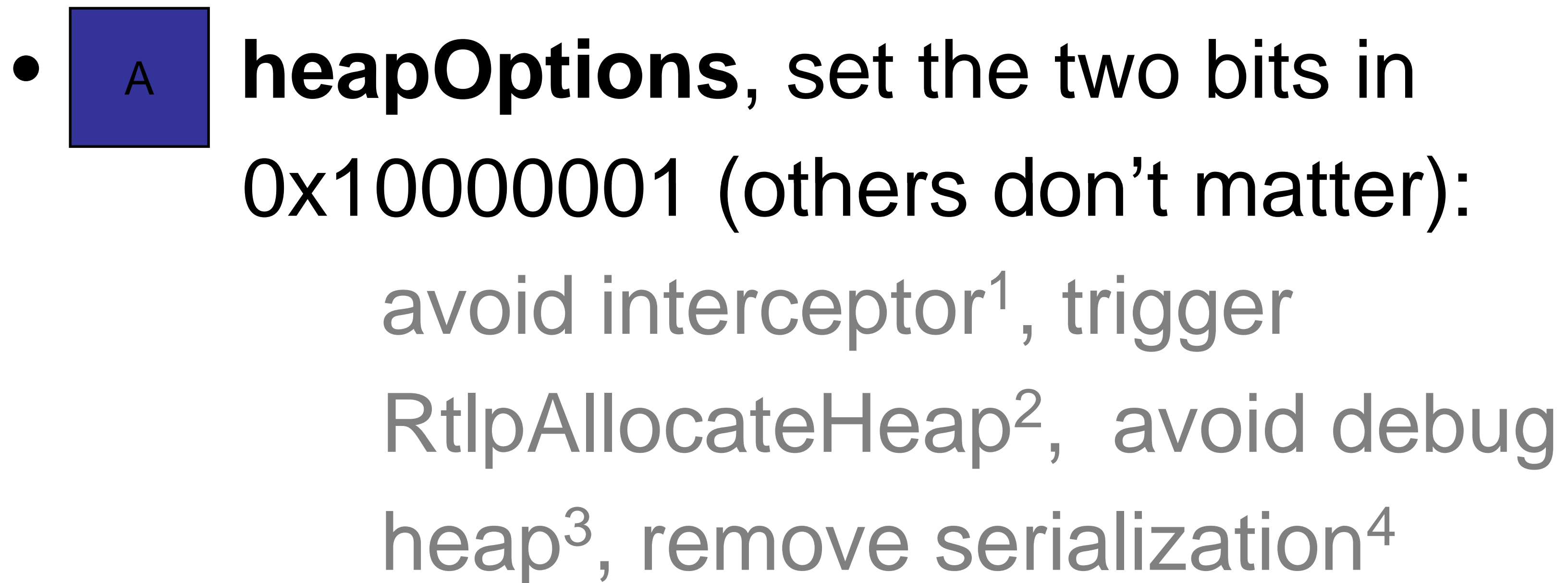
hHeap overflows VI

- Pattern 2:
  - Trigger creation of new heaps continuously until failure
  - Overflow into one or many of the new heaps
  - Trigger allocation on all newly created heaps

hHeap overflows VII

- Pattern 3:
  - Spray the entire range
  - 3rd to last segment allocated is directly before hHeap of heap being sprayed
  - Last 3 segments are size 0x10000, so take chunk from ~150kb back from failure
  - Free it, and use as overflow chunk
  - Trigger allocation

hHeap payload



- **heapOptions**, set the two bits in 0x10000001 (others don't matter):
  avoid interceptor[1], trigger RtlpAllocateHeap[2], avoid debug heap[3], remove serialization[4]

Offsets relative from .text segment base of ntdll.dll 6.0.6001.18000 (i.e. Vista SP1):
1. 6F3E7 2. 648DC 3. 8CC70 4. 677E5

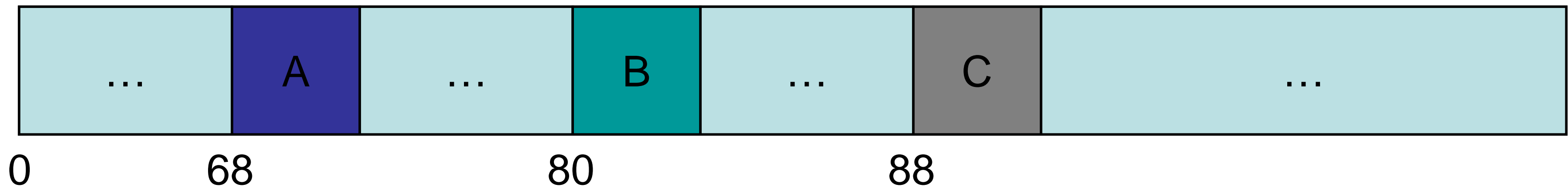
hHeap payload

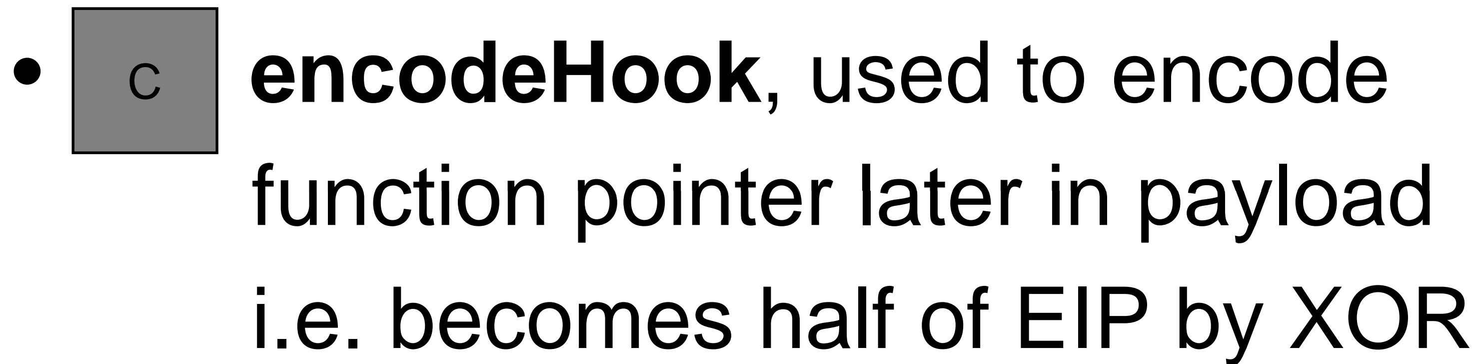
- B **heapCanary**, set to pass checksum inegrity test on freeEntry element[1] (more later)

  Set to 0x41414141

1. 678E5

hHeap payload

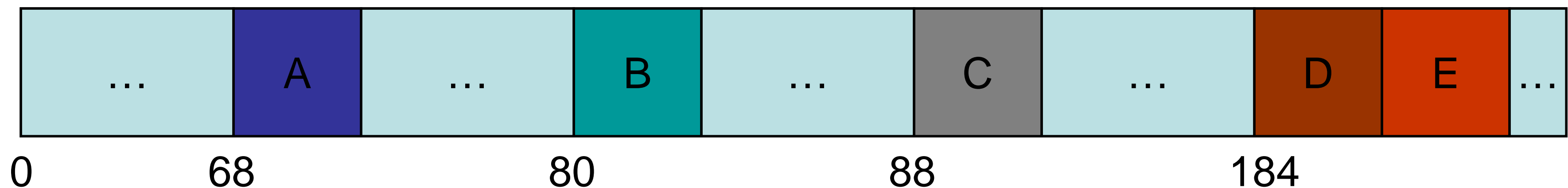


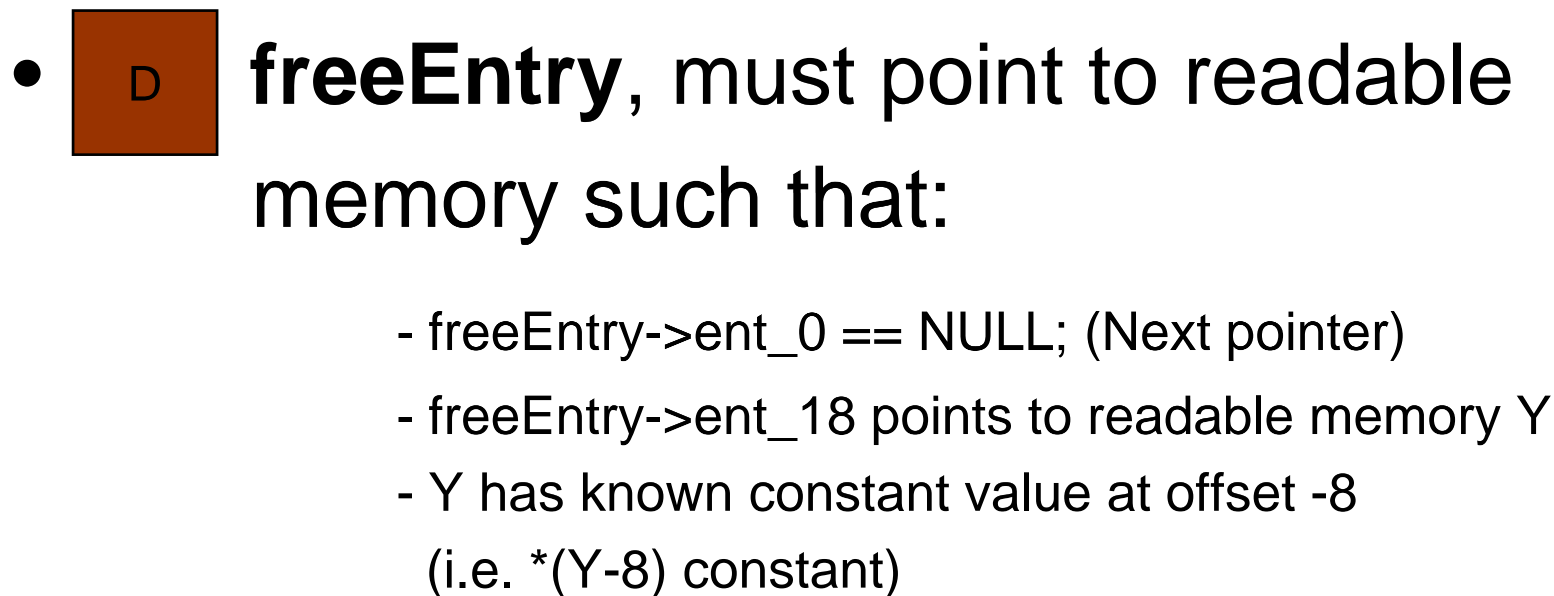


- **encodeHook**, used to encode function pointer later in payload i.e. becomes half of EIP by XOR

hHeap payload

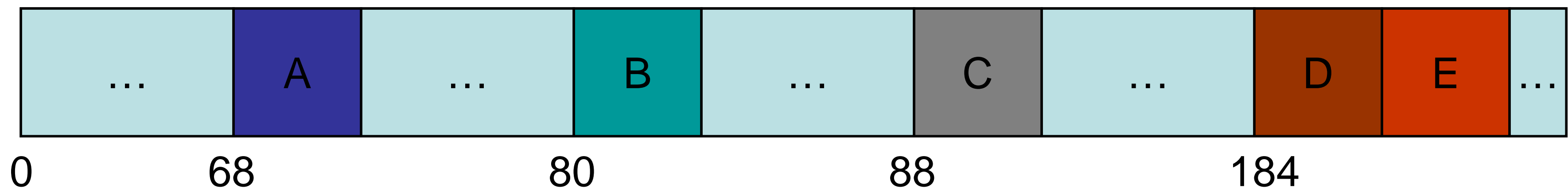
hHeap (X)





- **freeEntry**, must point to readable memory such that:
  - freeEntry->ent_0 == NULL; (Next pointer)
  - freeEntry->ent_18 points to readable memory Y
  - Y has known constant value at offset -8 (i.e. *(Y-8) constant)

hHeap payload



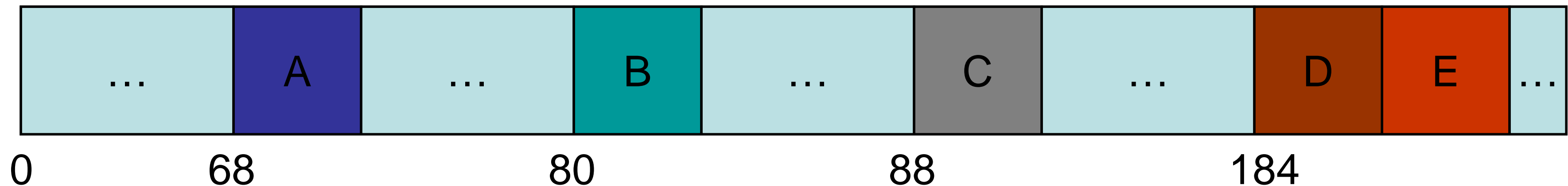
- D **freeEntry**, one good candidate is

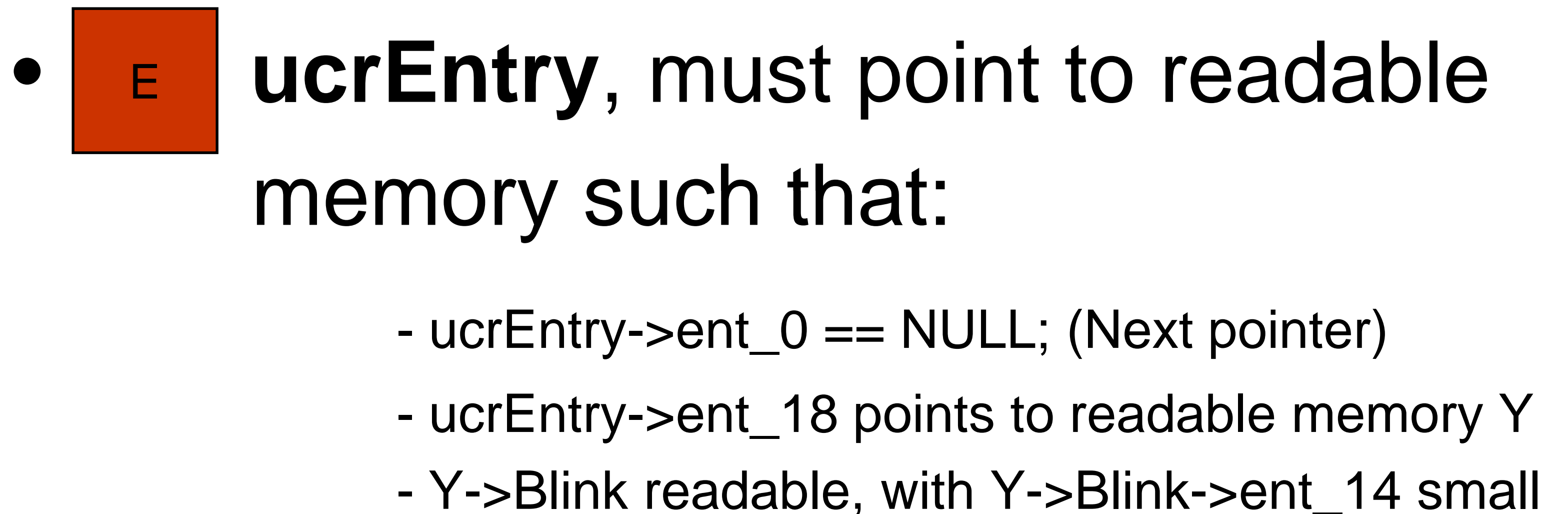
  0x7F6F5FC8

  Mysteriously static read-only mapping

  Y-8 value of sprayed/overflowed heap

hHeap payload

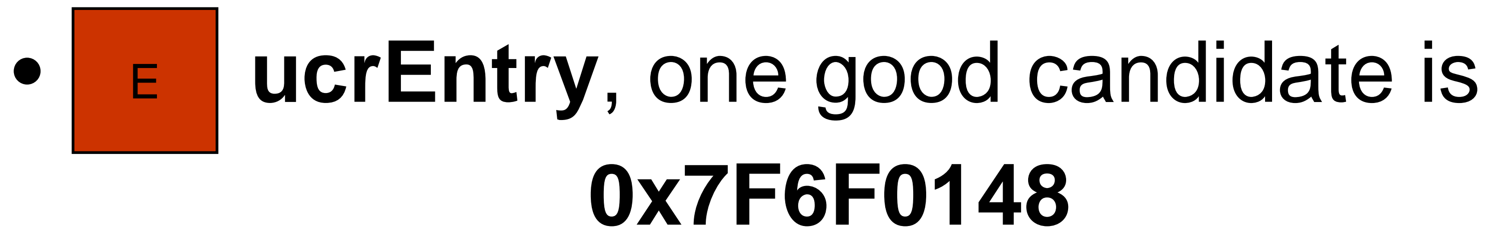




- **ucrEntry**, must point to readable memory such that:

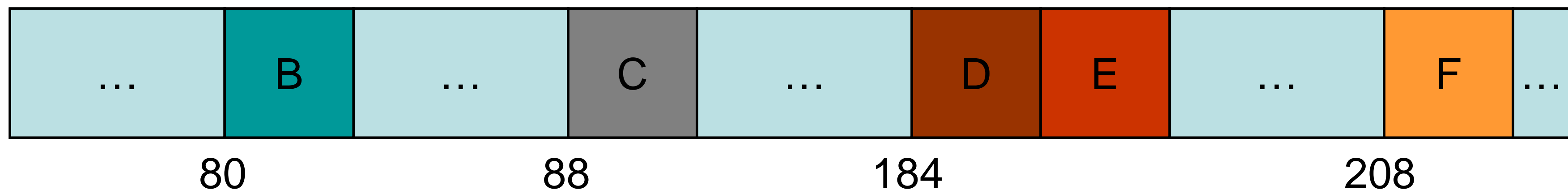
  - ucrEntry->ent_0 == NULL; (Next pointer)
  - ucrEntry->ent_18 points to readable memory Y
  - Y->Blink readable, with Y->Blink->ent_14 small

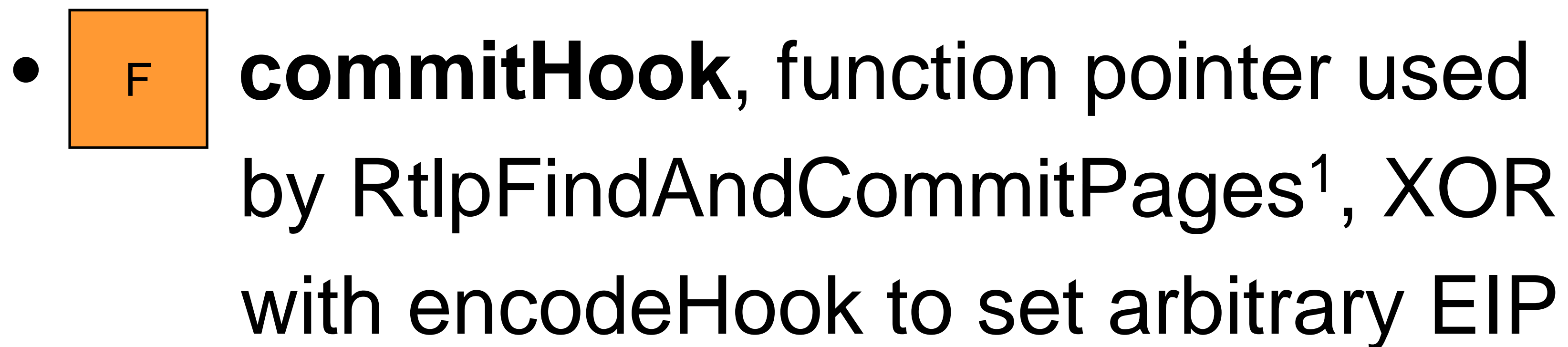
hHeap payload



- **ucrEntry**, one good candidate is **0x7F6F0148**

hHeap payload





- **commitHook**, function pointer used by RtlpFindAndCommitPages[1], XOR with encodeHook to set arbitrary EIP

hHeap payload II

stdafx.h | Heap3.cpp | Start Page

(Global Scope) | wmain(int argc, _TCHAR *[] argv)

```
for (i = 0; i < sz; i += 4) {        // includes some padding before the payload too
    *(set + i)      = 0x41;
    *(set + i + 1) = 0x41;
    *(set + i + 2) = 0x41;
    *(set + i + 3) = 0x41;
}

sz -= 0xD4;                          // set + sz is start of hHeap payload

*(set + sz + 0x44)       = 0x83;     // heapOptions, with MSB and LSB set
*(set + sz + 0x44 + 1)  = 0x82;
*(set + sz + 0x44 + 2)  = 0x82;
*(set + sz + 0x44 + 3)  = 0x82;

*(set + sz + 0x58)      = 0x3B;      // encodeHook
*(set + sz + 0x58 + 1) = 0x3B;
*(set + sz + 0x58 + 2) = 0x3B;
*(set + sz + 0x58 + 3) = 0x3B;

*(set + sz + 0xB8)      = 0xC8;      // freeEntry, 0x7F6F5FC8
*(set + sz + 0xB8 + 1) = 0x5F;
*(set + sz + 0xB8 + 2) = 0x6F;
*(set + sz + 0xB8 + 3) = 0x7F;

*(set + sz + 0xBC)      = 0x48;      // ucrEntry, 0x7F6F0148
*(set + sz + 0xBC + 1) = 0x01;
*(set + sz + 0xBC + 2) = 0x6F;
*(set + sz + 0xBC + 3) = 0x7F;

*(set + sz + 0xD0)      = 0x7A;      // commitHook, such that encodeHook ^ commitHook == 0x41414141
*(set + sz + 0xD0 + 1) = 0x7A;
*(set + sz + 0xD0 + 2) = 0x7A;
*(set + sz + 0xD0 + 3) = 0x7A;
```

hHeap overflows IIX

hHeap overflow requirements:

- Control the application to get contiguous layout with overflow before heap
- Suffer through a large heap spray (time!)
- Know (roughly) the position of the overflow chunk for alignment of payload
- Large enough overflow. Small overflows may need to be repeated to hit heap.

hHeap overflows IX

hHeap overflow requirements cont:

- Enough control of overflow character set to craft payload
- Must be 32-bit target

moving on…

Heap termination I

```
BOOL SetHeapOptions() {
   HMODULE hLib = LoadLibrary(L"kernel32.dll");
   if (hLib == NULL) return FALSE;

   typedef BOOL (WINAPI *HSI)
          (HANDLE, HEAP_INFORMATION_CLASS ,PVOID, SIZE_T);
   HSI pHsi = (HSI)GetProcAddress(hLib,"HeapSetInformation");
   if (!pHsi) {
      FreeLibrary(hLib);
      return FALSE;
   }

#ifndef HeapEnableTerminationOnCorruption
#   define HeapEnableTerminationOnCorruption (HEAP_INFORMATION_CLASS)1
#endif

   BOOL fRet = (pHsi)(NULL,HeapEnableTerminationOnCorruption,NULL,0)
            ? TRUE
            : FALSE;
   if (hLib) FreeLibrary(hLib);

   return fRet;
}
```

Heap termination II

Windows Vista ISV Security - Windows Internet Explorer

http://msdn.microsoft.com/en-us/library/bb430720.aspx

Importance and Priority of Defenses

The following table outlines the relative importance of these defenses and the priority with which ISVs should support each defense.

Defense	Priority
Address space layout randomization opt-in	Critical
DEP opt-in	Critical
/GS stack-based buffer overrun detection	High
/SafeSEH exception handler protection	High
Stack randomization testing	Moderate
Heap randomization testing	Moderate
Heap corruption detection	Moderate

Heh heh...

How to Test

Once any code and design changes have been made, it is important to verify that the operating system is configured correctly, and the application has the appropriate code changes.

C++ Compiler Use

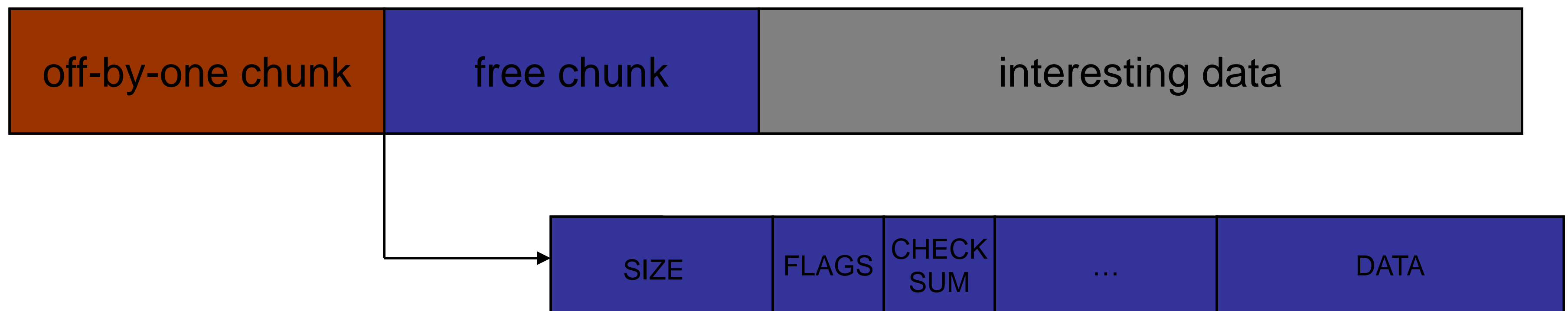
Verify that the version of the compiler is 13.10 or later. Version 14.00 or later is ***highly recommended***, as this is the

Heap termination III

- Must opt-in to heap termination on corruption with HeapSetInformation
- Windows executables basically always do
  - ntdll!RtlpDisableBreakOnFailureCookie == 0
- So just quickly, for all the 3rd party stuff that doesn't…

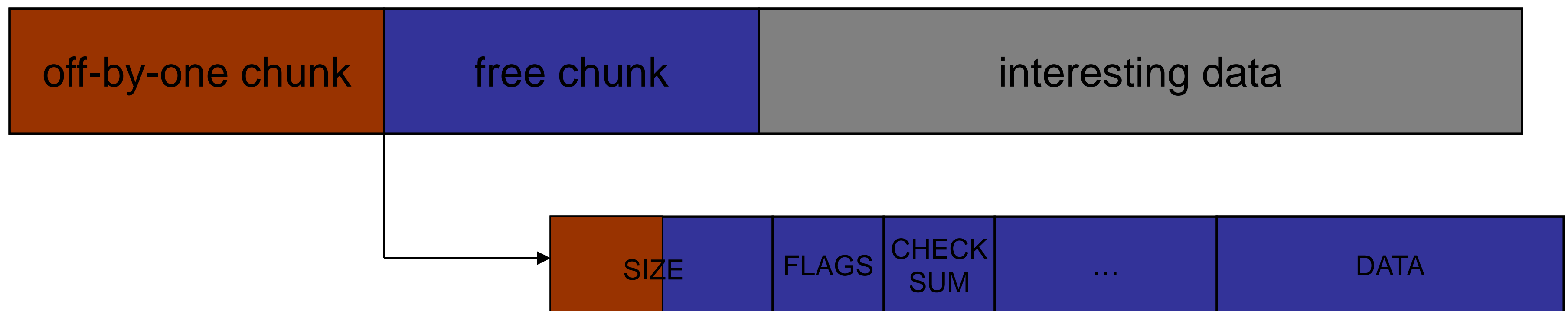
Off-by-one I

- Say you have off-by-one or small overflow on some heap. Not exploitable?



Off-by-one II

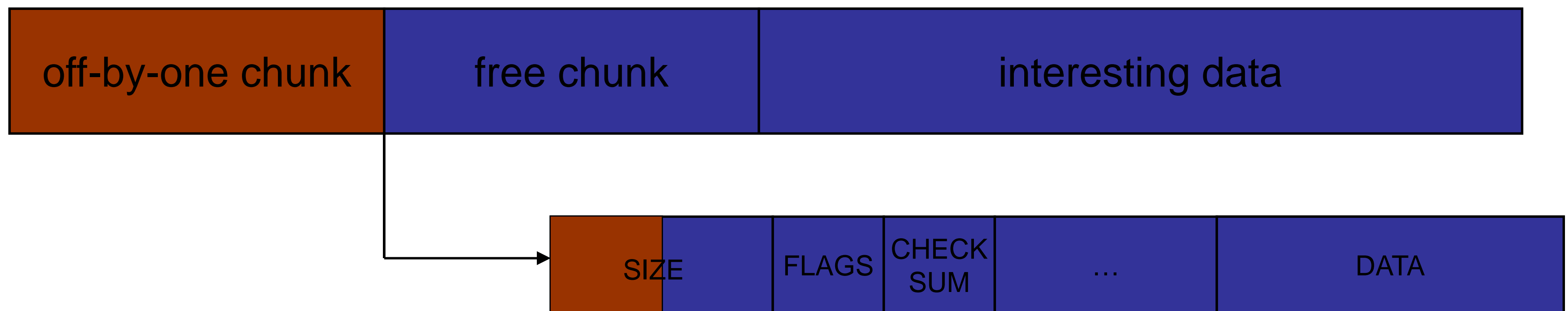
- Modify free chunk's size value to something larger



- Envelope interesting data in free chunk
- Must be precise with new size value

Off-by-one III

- Trigger allocations of the new size, HeapAlloc will eventually return free chunk



- Checksum will fail, but heap continues…
- Application still using interesting data, but can be overwritten using new allocation

Off-by-one IV

Off-by-one overflow requirements:

- Not opted-in for termination on heap corruption
- Position off-by-one chunk next to an appropriate envelope chunk
- Know exact sizes of free and interesting chunks
- Sufficient control of returned chunk to control interesting data

Checksum collisions

- For small overflows, the same technique can be applied on a large scale even when heap termination is enabled
- Overflow 3 bytes of adjacent header with constant value
- 3-Byte XOR against random value collides with probability ~= 0.004
- Approximately 250 attacks per success
- Good enough for worms, repeatable vulns

Canary leak

- Leak of a chunk header of known size and state gives leak of heap wide canary value

  $$C1 = L1 \wedge K1$$

  $$C2 = L2 \wedge K2$$

  $$C3 = L3 \wedge K3$$

  $$C4 = L4 \wedge K1 \wedge K2 \wedge K3$$

- Can then use overflow to change size, allocated/free, flags, FWD/BCK links etc

UCR overflow I

- UCR structure used to link and describe segments
- Stored at beginning and end of each heap segment

- So end UCR structure always accessible by overflow in a heap segment

UCR overflow II

- UCR overflow techniques

LFH bucket overflow I

- LFH bucket allocated internally using RtlAllocateHeap when LFH created

RtlpAllocateHeap

  RtlpPerformHeapMaintenance

    RtlpActivateLowFragmentationHeap

      RtlpExtendListLookup

        RtlAllocateHeap (sz 0x3D14)

LFH bucket overflow II

- LFH bucket overflow technique

Future Techniques

- Attack on HeapFree
- Attack on LFH

Securing the Heap I - Specific

- Add guard pages, remove functions pointers from hHeap HANDLE
- Remove internal use of RtlpAllocateHeap, replace with guarded mappings
- Similarly remove UCR from end of segments
- Ensure checksum is always validated before any use of chunk headers

Securing the Heap II - Generic

- Add randomization to segments and large chunks
- Increase the amount of address entropy
- Increase the size of the checksum
- Encode all of the chunk
- Reduce use of list operations

Securing the Heap III - Theory

- Remove all meta-data structures from anywhere contiguous to any data
- Still have canaries between chunks, but not encoding anything (just for integrity)
- Smaller segments, more guard pages
- Introduce true non-determinism to allocator patterns (i.e. internally randomize where a chunk can go, while still ensuring locality)

Questions