Attacking the Vista Heap

Ben Hawkes

LATERAL SECURITY

v4

The Heap

- The “heap” describes:
  - areas of memory (as in RAM) used by an application dynamically
  - implementation of structures and algorithms for managing memory

Windows API: HeapAlloc, HeapFree

The Heap

Heap Vulns

- Application uses heap memory incorrectly
- Results in corruption
- Heap memory can be placed in to an inconsistent state

Heap Exploit

- An “exploit” places the heap in to a state designed to give the attacker arbitrary code execution

- An HTTP request responds with a command shell instead of a response… because we exploited a remote heap overflow in IIS 6

Intro-clusion 1

- Heap vulnerabilities are harder to find
- Programmers don’t suck quite as much as they used to
- **CLAIM:** Proportionally, vulnerability research in this area is decreasing

Intro-clusion 2

- Heap exploits are harder to write than ever
- Application specific attacks are the future
- **CLAIM:** Complex heap implementation attacks should still be considered

Intro-clusion 3

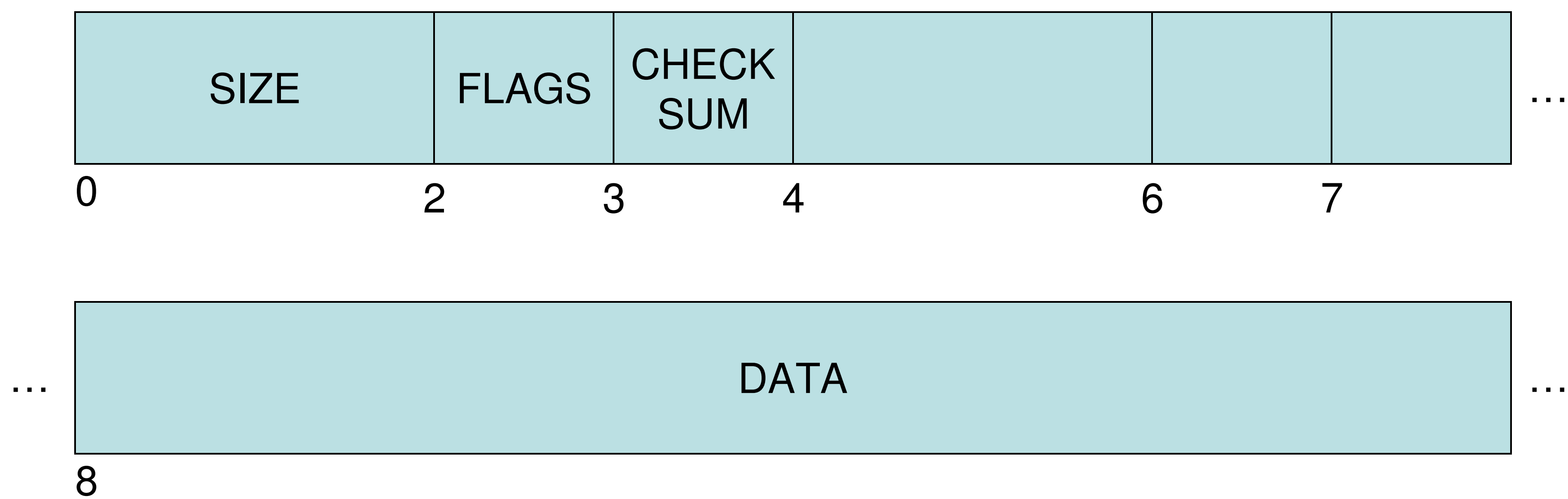
- Now is a good time to start learning and looking for these bugs

- History repeats itself.

- **CLAIM:** The decline of memory corruption research will coincide with the increase of memory corruption bugs

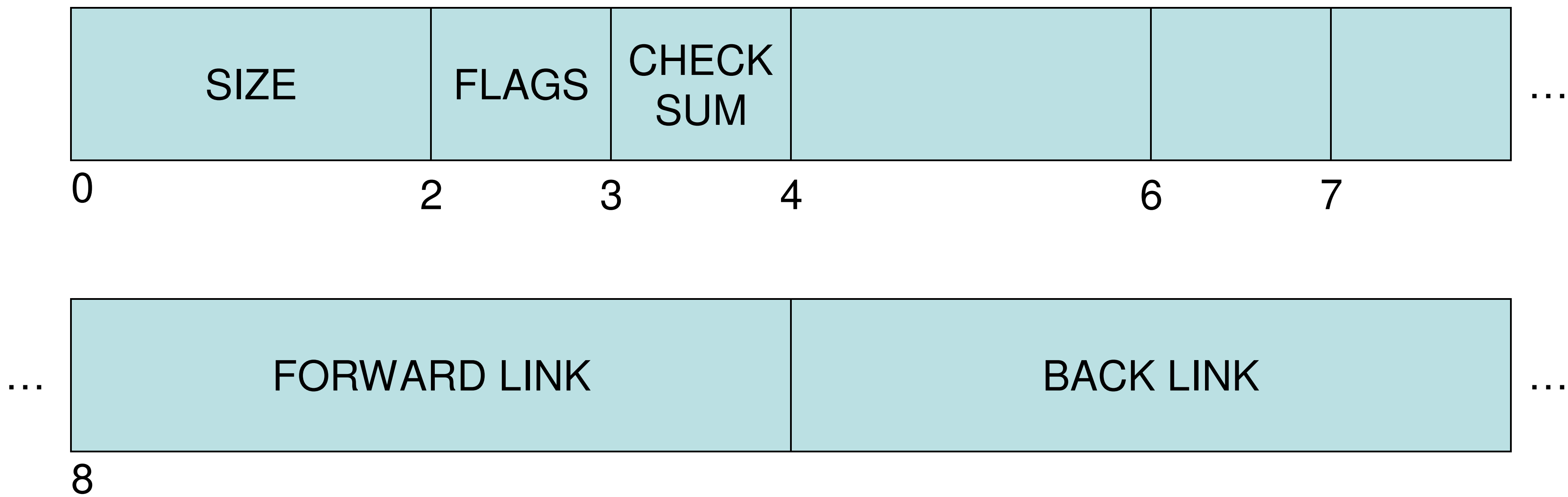
Heap Chunk

- HeapAlloc returns a chunk of memory for use by the application
- It looks like this:



Heap Chunk

- Or, if the chunk gets freed by HeapFree it looks like this:



Unlink

- Solar Designer haxed netscape in 2000
- Introduced the “unlink” technique for writing heap exploits

- Popularized in Phrack 57
  - Once upon a free()
  - Vudo malloc tricks

Unlink

- Countless exploits using this technique

- But only two with rad names:
  - OpenSSL KEY_ARG a.k.a Slapper
  - RPC DCOM a.k.a Blaster

Unlink

- HISTORY: control the fwd/bck links of a chunk, trigger removal of chunk from free list:

```
BK = P->BCK
FD = P->FWD
FD->BCK = BK
BK->FWD = FD
```

Any takers?

Unlink

- Arbitrary overwrite with an arbitrary value
  - pwned!

- Except it was trivially fixed
- So how do you write a heap exploit now?

Current Heap Exploitation

- So evil haxors now use application specific techniques:
  - Overflow the target application’s data stored on the heap
  - Ensure important structures are allocated after the overflow
  - Profit

Attacking the Application

- But… a vulnerability can only corrupt a subset of all heap data
- So… you can't always corrupt an "important" structure

Attacking the Application

- At the time, unlink was a complex technique
- It exploited underlying heap structures

- History repeats itself
- We can target underlying heap structures in Vista too

Overflow Summary

- A heap overflow can potentially overwrite:
  - Internal heap structures
    - Chunk headers
    - Bucket structures
    - Main heap structure
  - Application data
    - Application buffers, flags, integers etc.
    - Function pointers
    - Heap pointers

Attacking the Vista Heap

- The techniques I published in Vegas:
  - Overwrite the main heap structure
  - Free and then allocate the main heap structure, overwrite with application
  - Off-by-one into apps which do not opt in to “termination on heap corruption” option
  - Overwrite low fragmentation heap’s bucket structure
  - Partial overwrite of LFH heap chunk

Exploit Techniques

- Build up an arsenal of techniques
- Then choose the best technique for the vulnerability

- Let the vulnerability choose the technique, all options should be considered

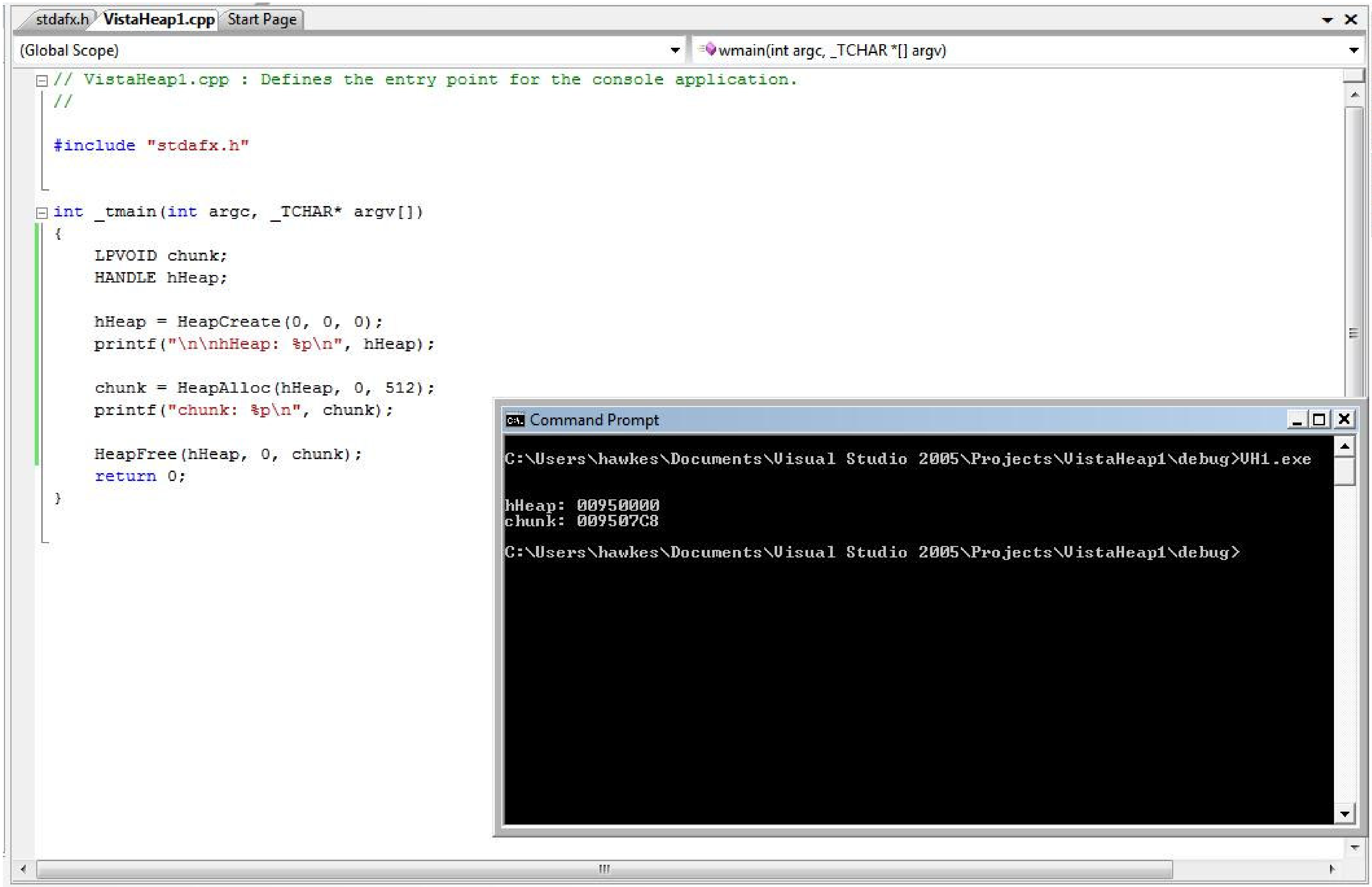
hHeap HANDLE payload

Heap HANDLE

- Application requests access to a heap by calling HeapCreate
- This initializes all heap structures
- Returns a pointer to a heap HANDLE
- Which can then be used by the allocator

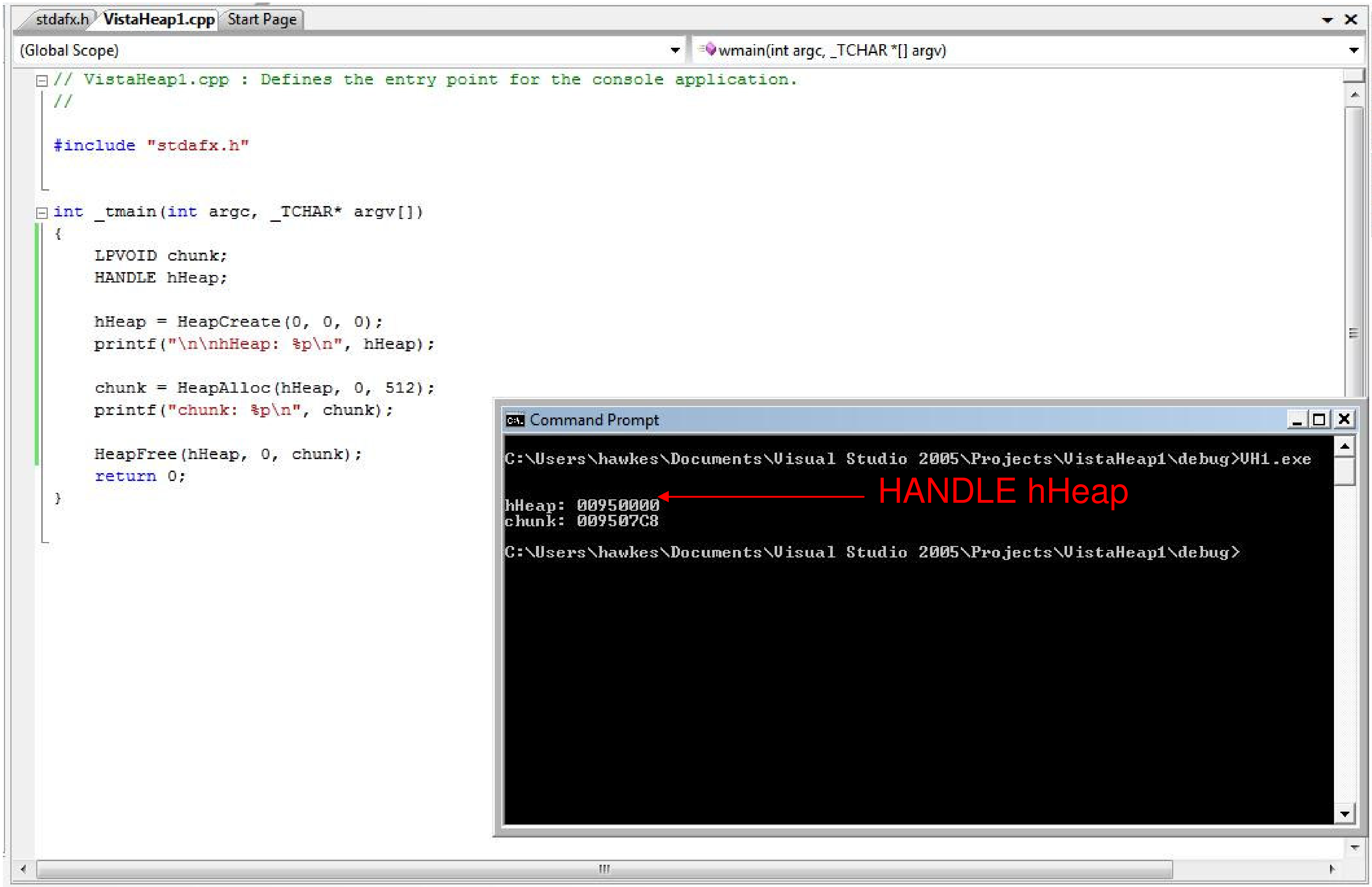
```
HANDLE hHeap = HeapCreate(0,0,0);
LPVOID mem = HeapAlloc(hHeap, 0, 512);
```

Heap API



stdafx.h
VistaHeap1.cpp
Start Page
(Global Scope)
wmain(int argc, _TCHAR *[] argv)
// VistaHeap1.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
LPVOID chunk;
HANDLE hHeap;
hHeap = HeapCreate(0, 0, 0);
printf("\n\nhHeap: %p\n", hHeap);
chunk = HeapAlloc(hHeap, 0, 512);
printf("chunk: %p\n", chunk);
HeapFree(hHeap, 0, chunk);
return 0;
}
Command Prompt
C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>VH1.exe
hHeap: 00950000
chunk: 009507C8
C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>

Heap API


stdafx.h
VistaHeap1.cpp
Start Page
(Global Scope)
wmain(int argc, _TCHAR *[] argv)
// VistaHeap1.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
LPVOID chunk;
HANDLE hHeap;
hHeap = HeapCreate(0, 0, 0);
printf("\n\nhHeap: %p\n", hHeap);
chunk = HeapAlloc(hHeap, 0, 512);
printf("chunk: %p\n", chunk);
HeapFree(hHeap, 0, chunk);
return 0;
}
Command Prompt
C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>VH1.exe
hHeap: 00950000
chunk: 009507C8
C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>
HANDLE hHeap

Heap API

stdafx.h | VistaHeap1.cpp | Start Page

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

```cpp
// VistaHeap1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    LPVOID chunk;
    HANDLE hHeap;

    hHeap = HeapCreate(0, 0, 0);
    printf("\n\nhHeap: %p\n", hHeap);

    chunk = HeapAlloc(hHeap, 0, 512);
    printf("chunk: %p\n", chunk);

    HeapFree(hHeap, 0, chunk);
    return 0;
}
```

Command Prompt

```
C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>VH1.exe

hHeap: 00950000
chunk: 009507C8

C:\Users\hawkes\Documents\Visual Studio 2005\Projects\VistaHeap1\debug>
```

HANDLE hHeap

LPVOID chunk

Heap HANDLE II

- Heap HANDLE is just a structure at the beginning of the heap's memory area
- Heap HANDLE is ridiculously important
- Central management structure for each individual heap
  - Free lists
  - Heap canary
  - Flags and tunable options
  - Etc…

Heap HANDLE II

- Unfortunately there are no guard pages in the Vista heap implementation
- Relies on randomization to introduce holes in the address space

- Heap spray + Heap Overflow

  = Heap HANDLE overflow

Heap HANDLE II

- Introducing bad ass technical Heap HANDLE payload:

  [‘H’x68][0x82828283][‘H’x8][0x41414141]
  [‘H’x4][encodeHook][‘H’x92][0x7F6F5FC8]
  [0x7F6F0148][‘H’x16][commitHook]

- Total of 212 bytes

Heap HANDLE II

- Set a Heap HANDLE to this payload, trigger an allocation on the heap.
- This will give arbitrary EIP:

  EIP = encodeHook XOR commitHook

- See the appendix in this slide deck for more detail
- Payload needs more real life testing (works in au/nz, but .us, cn, de etc? quite probably not in this form)

hHeap overflows VII

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.

Arbitrary Free

Arbitrary Free

- By overflowing heap pointers we can control the way the heap "works"
  - Which chunks will be freed
  - And thus where new chunks will be allocated

- Can perform exploits against either the application or the heap implementation
- CLAIM: Flexibility leads to reliability

Arbitrary Free I

- Assume you can overflow into a pointer returned from HeapAlloc called X
  - i.e. X = HeapAlloc(hHeap, 0, 4096);

- Application will HeapFree X at some point
- So…

Arbitrary Free II - Generic

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 III

Arbitrary Free (generic) 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

Vista Arbitrary Free I

- Generic arbitrary free attacks application
- Vista heap implementation is part of the application…
- So lets attack it!

Vista Arbitrary Free II

- Is there some way to reliably make the overflowed heap pointer X point to the “important structure” heap chunk (Y) required in generic arbitrary free?
- hHeap HANDLE is an important structure..

Vista Arbitrary Free III

- Disturbingly, hHeap HANDLE is also a valid heap chunk
- Has its own HEAP_ENTRY at offset 0
  - Encoded with valid canary
  - Containing a correct checksum
  - Set up by HeapCreate
- Known location relative to all heap pointers in the first segment

Vista Arbitrary Free IV

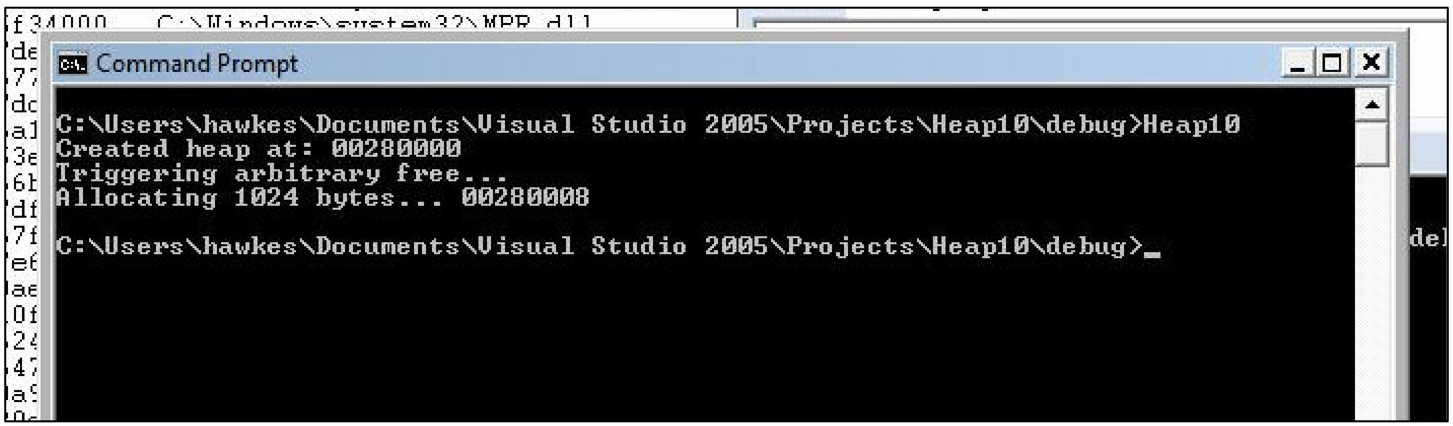
- Partial overwrite of heap pointer e.g.

$$X = \texttt{0x00B8A228}$$
$$\downarrow$$
$$\texttt{0x00B8XXXX}$$
$$\downarrow$$
$$\texttt{0x00B80008} = Y$$

- Then trigger HeapFree on X

Vista Arbitrary Free V

- Trigger allocations of size <= 1400
- Eventually HeapAlloc will return… the hHeap HANDLE (Z)



- Use application to write payload described earlier

Vista Arbitrary Free VI

Vista Arbitrary Free requirements:

- X points to a chunk in first 64kb of some heap (usually)
- Sufficient control of Z usage

NOTE: all other segments start with valid heap entry too… hmm

Securing the Heap

Vista Heap Changes

- List integrity checks
- Encoded heap entry headers
- Checksum in headers
- Randomized heap base
- Fail on corruption
- Low Fragmentation Heap

Securing the Heap I - Specific

- Add guard pages, remove functions pointers from hHeap HANDLE
- Remove internal use of RtlpAllocateHeap, replace with guarded mappings
- 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 some locality)

Food for Thought

- Fundamentally this type of bug will be a problem for a long long time
- Because our computers fundamentally handle memory corruption badly

Rant On

- The application sees a large block of available virtual memory
- It is the application's job to decide how this will be segregated
- This is fundamentally **wrong**

- Should users decide how to set their file permissions? DAC vs MAC

Rant Off

- We need an architecture that allows efficient segregation of memory at a byte level (as opposed to page level)
- Make the system handle data segregation

- But this is not going to happen any time soon (if ever)

Rant Off

- What about C#, Java etc?
- The underlying architecture for their virtual machines is still the same monolithic beast…

- But it is an improvement in terms of attack surface

Summary

- Heap vulnerabilities are hard to exploit
- Sometimes even impossible
- But we can usually win if we are determined

- This seems like arcane knowledge
- But these bugs are here for the long term, so its worth learning (for money + fame…)

Greetz

+ caddis and the rux crew, booyah!

+ von d, ratu and crew

+ the circle of lost hackers

+ duke, mercy, nemo, dme, cyfa, scott, moby, zilvio, antic0de, pipes, si, delphic, metl, hntr, sham, core, kaixin, …

Appendix

1 – page 54 – hHeap overflow

2 – page 76 - Adjusted Double free

3 – page 82 – Heap Termination

4 – page 90 – Information Leak

5 – page 94 – Low Frag Heap

Appendix 1
hHeap overflows

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[1]

- Spray amount Z, with Z > ~16MB
- Y + (Z/2) => your data w/ probability ~= 1

1. with variability approaching 2MB (more when early)

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: large contiguous writable block of committed memory

hHeap overflows I

- Overflow in contiguous space can overwrite potentially everything on a heap
  - Application data from different heaps
  - Segment, chunk and bucket headers
  - hHeap HANDLEs

hHeap overflows I

- Overflow in contiguous space can overwrite potentially everything on a heap
  - Application data from different heaps
  - Segment, chunk and bucket headers
  - hHeap HANDLEs

hHeap overflows III

- **Goal 1:** get overflow chunk positioned before some hHeap HANDLE
- **Goal 2:** Craft payload to overwrite commitHook…
- Encoded function pointer located in hHeap HANDLE, called when heap extended
- **Result:** arbitrary code execution on next HeapAlloc

hHeap overflows IV

- 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

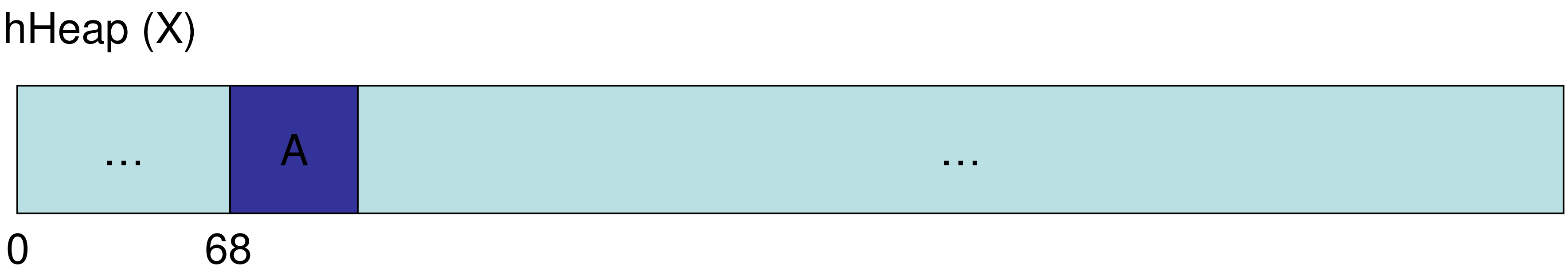
hHeap overflows V

- 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 VI

- 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

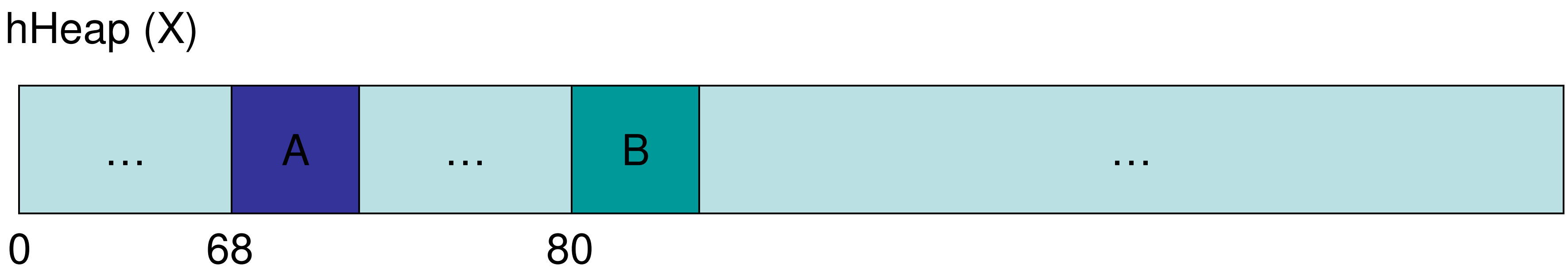


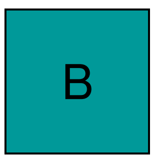
- A **heapOptions**, set the two bits in 0x10000001 (others don't matter):

  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



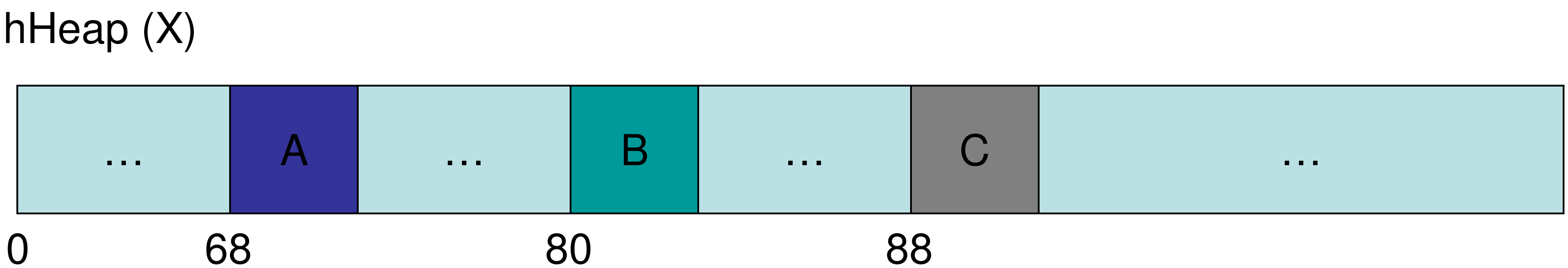
- 

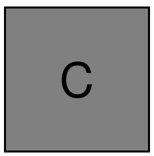
heapCanary, set to pass checksum inegrity test on freeEntry element[1] (more later)

Set to 0x41414141

1. 678E5

hHeap payload

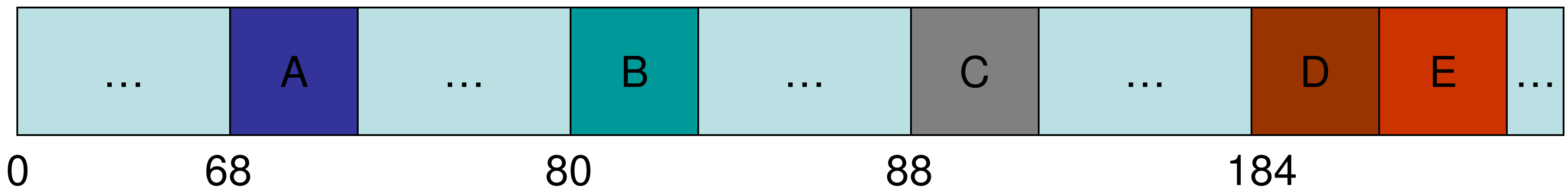


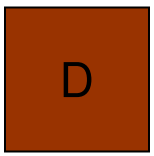
- 

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

hHeap payload

hHeap (X)



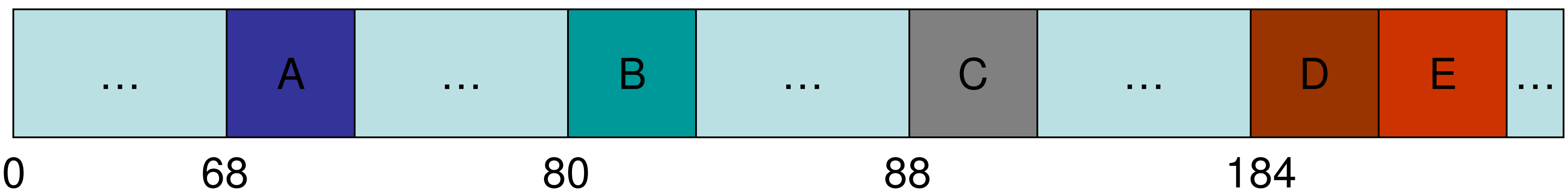
- 

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

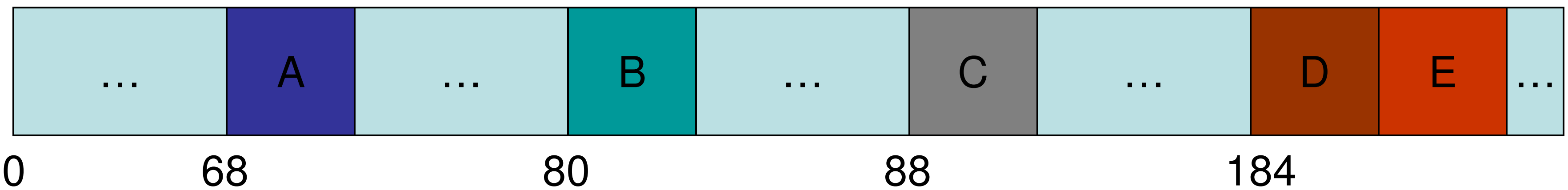
hHeap (X)

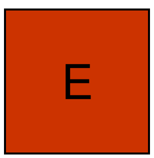


- D **freeEntry**, one good candidate is
  0x7F6F5FC8
- Mysterious static read-only mapping
- Y-8 value points to sprayed or overflowed heap area… set equal to heapCanary
- Or just set up another heap spray

hHeap payload

hHeap (X)

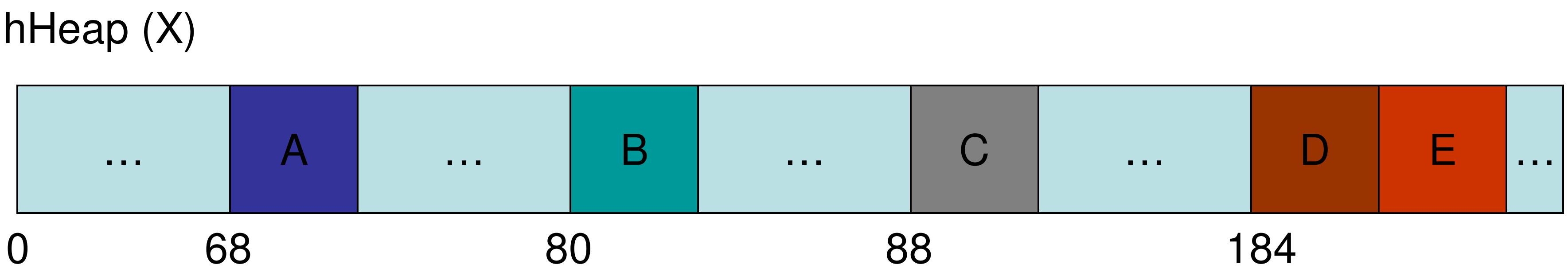


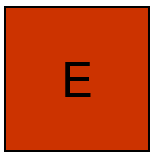
- 

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



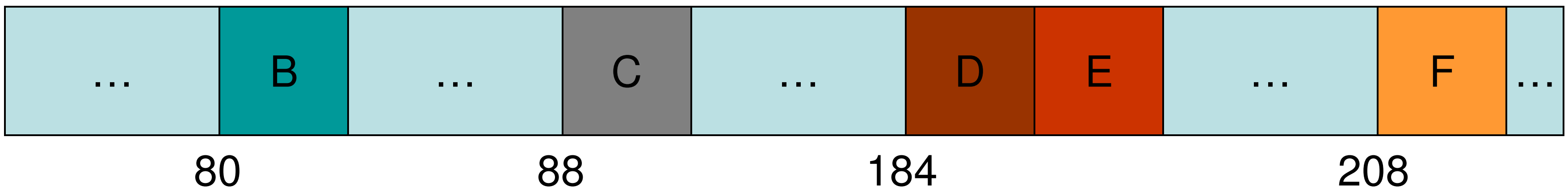
- 

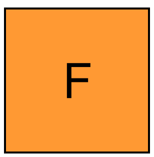
ucrEntry, one good candidate is **0x7F6F0148**

- Again, alternative is just to use some crafted heap spray address

hHeap payload

hHeap (X)



- 

commitHook, function pointer used by RtlpFindAndCommitPages, XOR with encodeHook to set arbitrary EIP

Appendix 2
Adjusted double free

Adjusted Double Free I

- 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 II

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 III

- 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 IV

- 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 V

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:

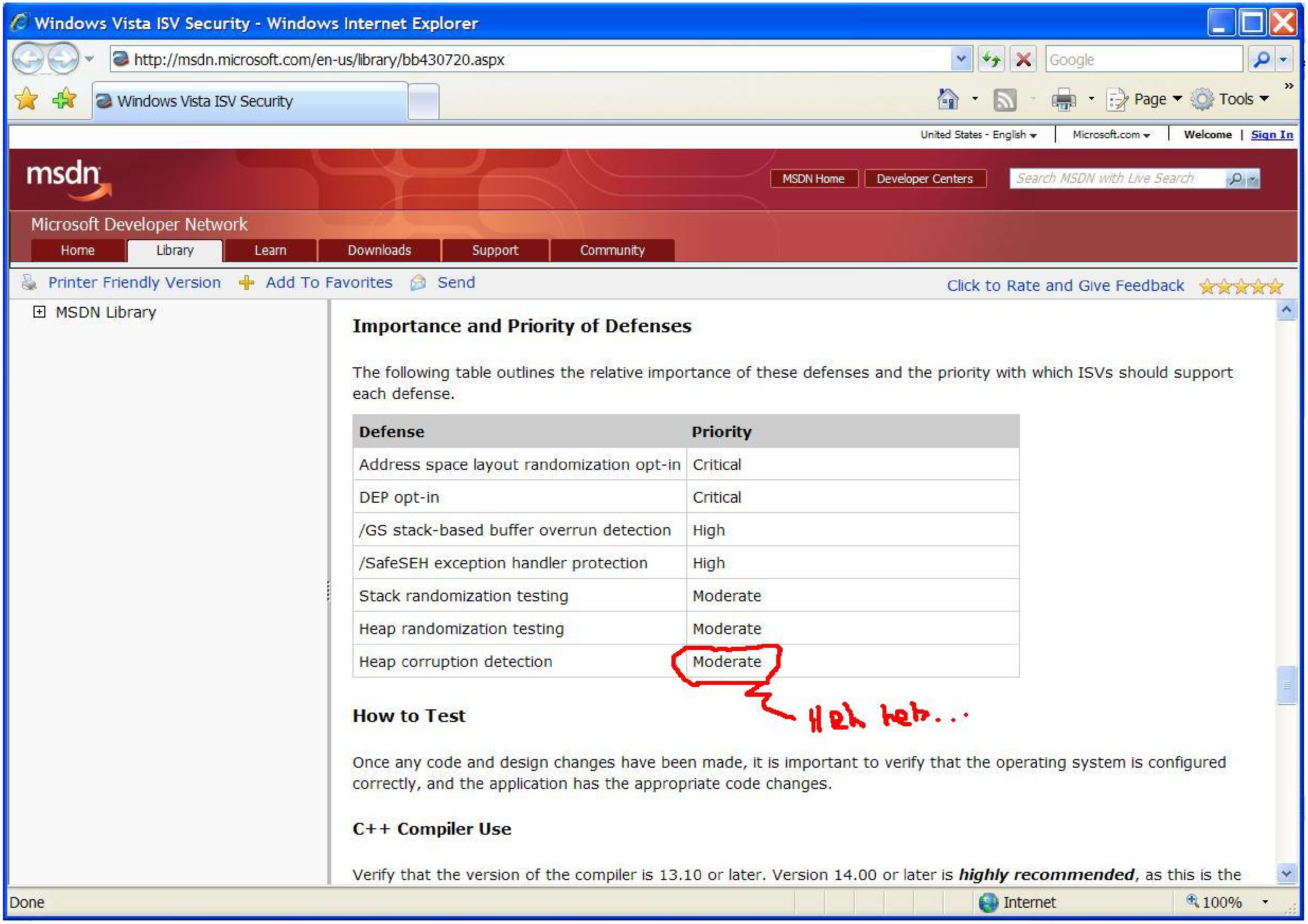
- ASLR doesn’t matter

Appendix 3
Heap Termination

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



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

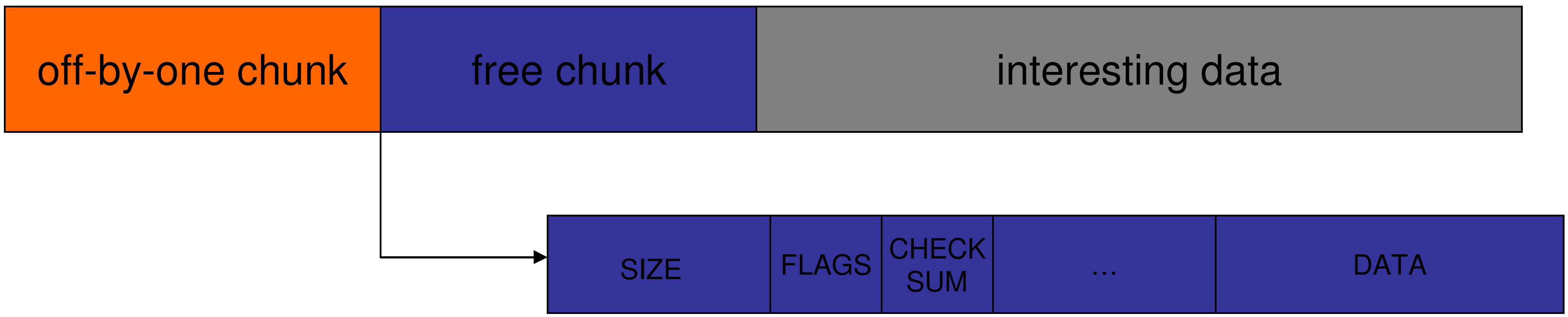
Heap termination III

- 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…

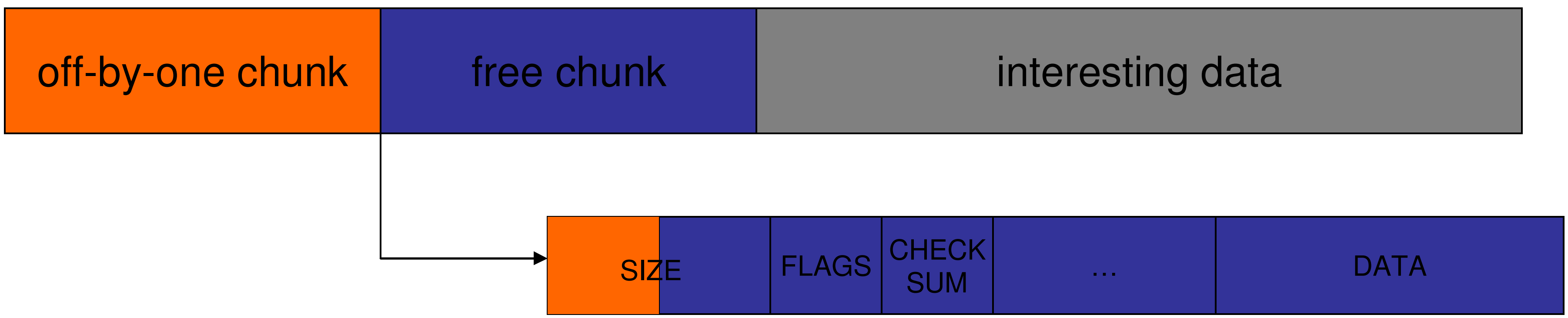
Off-by-one I

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



Off-by-one II

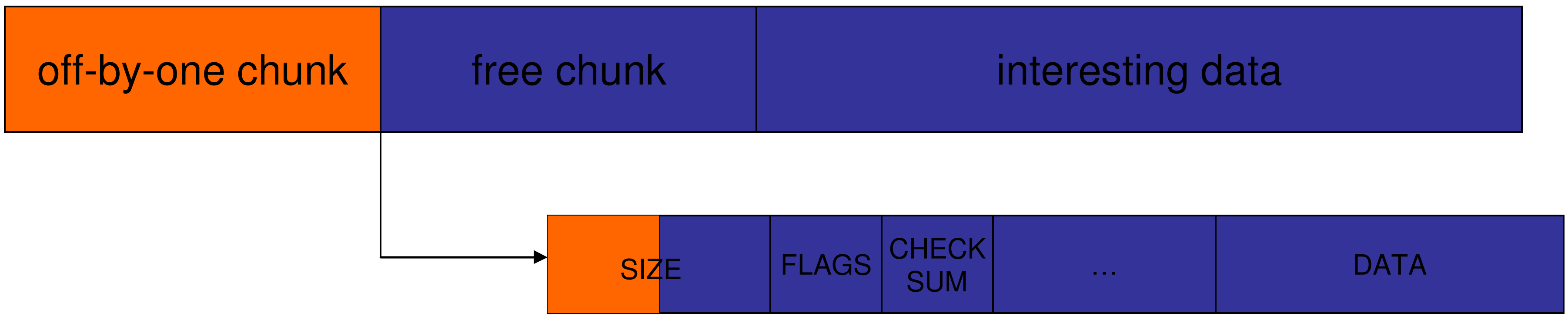
- Modify free chunk's size value to something larger



- 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

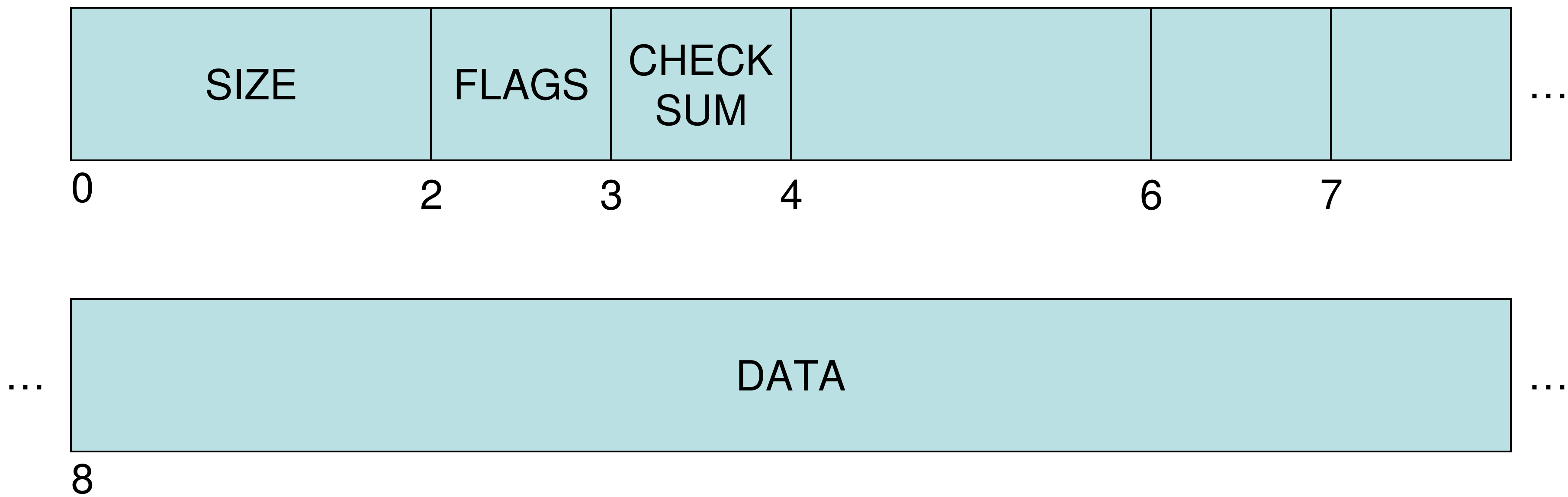
Off-by-one overflow requirements:

- 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

Appendix 4
Information Leak

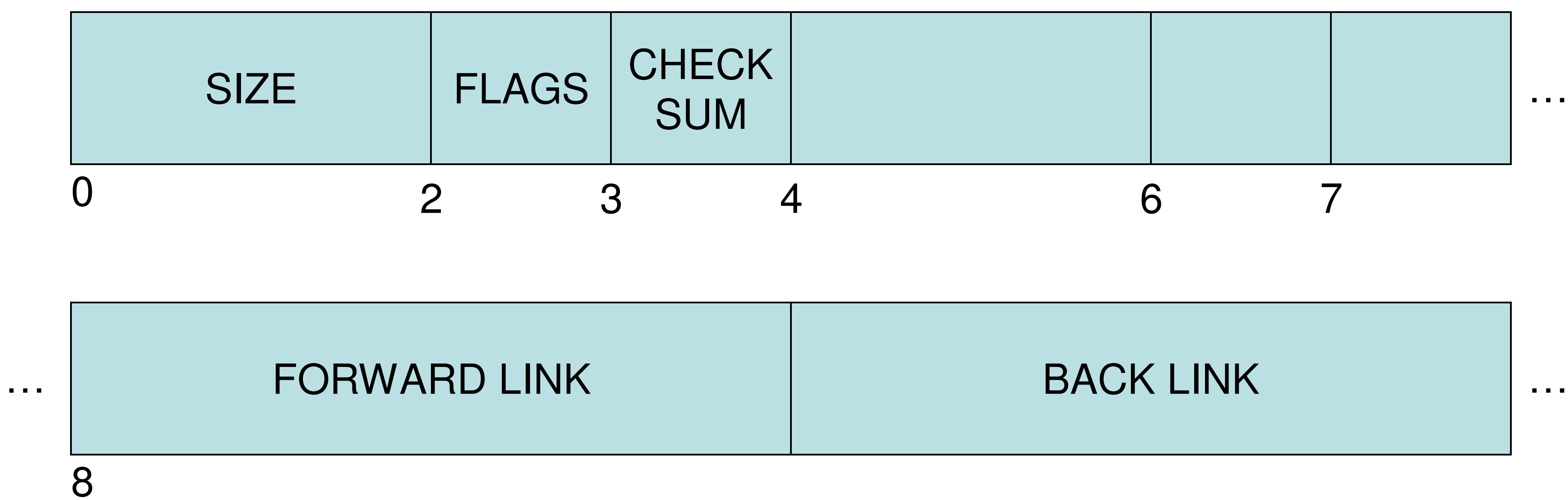
Vista Chunks

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



Vista Chunks

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



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

Appendix 5
Low Fragmentation Heap

LFH bucket overflow I

- LFH bucket allocated internally using RtlAllocateHeap when LFH created

RtlpAllocateHeap

RtlpPerformHeapMaintenance

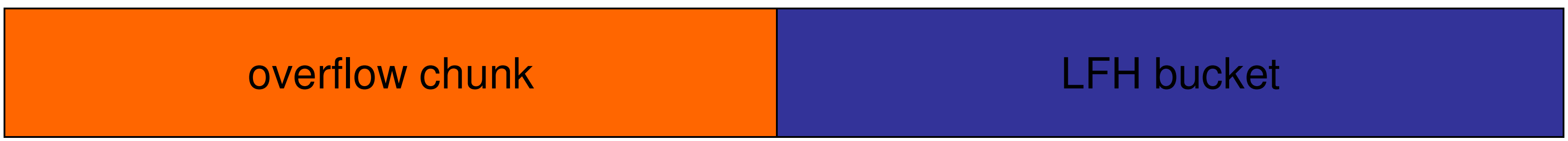
RtlpActivateLowFragmentationHeap

RtlpExtendListLookup

RtlAllocateHeap (sz 0x3D14)

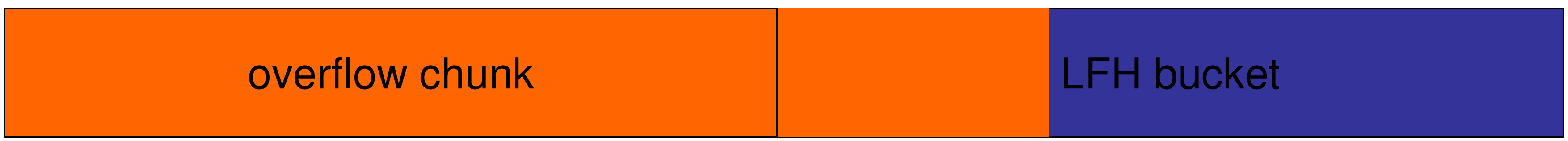
LFH bucket overflow II

- LFH bucket created relatively deterministically, i.e. easy to find
- Force overflow chunk to be allocated before LFH bucket



LFH bucket overflow II

- LFH bucket created relatively deterministically, i.e. easy to find
- Force overflow chunk to be allocated before LFH bucket



- Overflow first 24 bytes (or more)
- Trigger alloc request of size R >= 1024

LFH bucket overflow III

- RtlAllocateHeap also used internally by LFH allocator[1]
- Uses LFH bucket structure to decide location of hHeap…

- GOAL: trigger internal LFH allocation with arbitrary hHeap
- Can then use previous payload

1. RtlpAllocateUserBlock from RtlpLowFragHeapAllocFromContext

LFH bucket overflow IV

- Set ent_14, ent_20 of LFHBucket to control X

$$X = ent\_20 + ((R + 8)/8 - ent\_14)*4$$

- Set X->ent_4 to Y

LFH bucket overflow V

- Y is used as LFH context
- Point Y to an “empty” context:

Y offset	value
0	Zero
218 … 25C	Zero
-A0..-A4	Zero
-B0..-B4	Zero
-DC	hHeap

LFH bucket overflow V

- Y is used as LFH context
- Point Y to an “empty” context:

(Y-100) offset	value
100	Zero
318 … 35C	Zero
60..64	Zero
50..54	Zero
24	hHeap

LFH bucket overflow VI

LFH bucket overflow requirements:

- Position overflow chunk before some LFH bucket
- Find an appropriate X value
- Craft or find an appropriate fake LFH context (Y)
- Form a correct hHeap payload at the location decided by Y
- Reliably trigger R-allocation after overflow

LFH header overflow I

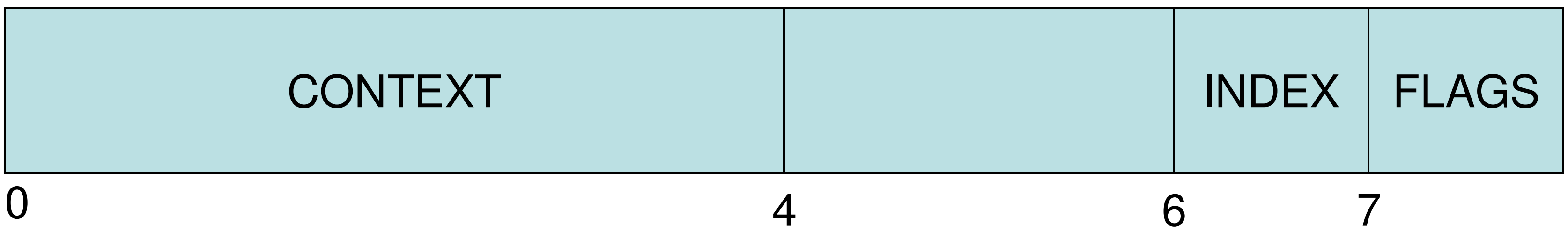
- Given an overflow that can write NULL bytes, what do we gain?

LFH header overflow I

- Given an overflow that can write NULL bytes, what do we gain?
- Small overflow envelope technique on LFH chunks **even with a terminating heap**

LFH header overflow II

LFH_HEAP_ENTRY:



- RtlpLowFragHeapFree uses INDEX to determine adjusted location of chunk **before checksum test**
- Only when FLAGS == 5 and CONTEXT == 0x0000002

LFH header overflow III

LFH_HEAP_ENTRY:

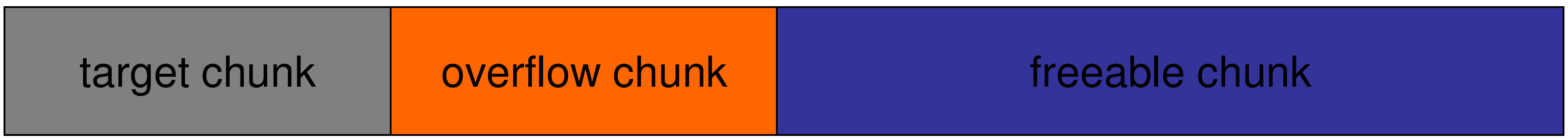
0x02	0x00	0x00	0x00	0x41	0x41	X	0x05
P+0				P+4		P+6	P+7

$$P' = P - (X * 8)$$

- P' must point to valid LFH_HEAP_ENTRY
- One byte gives range of 2040 bytes

LFH header overflow IV

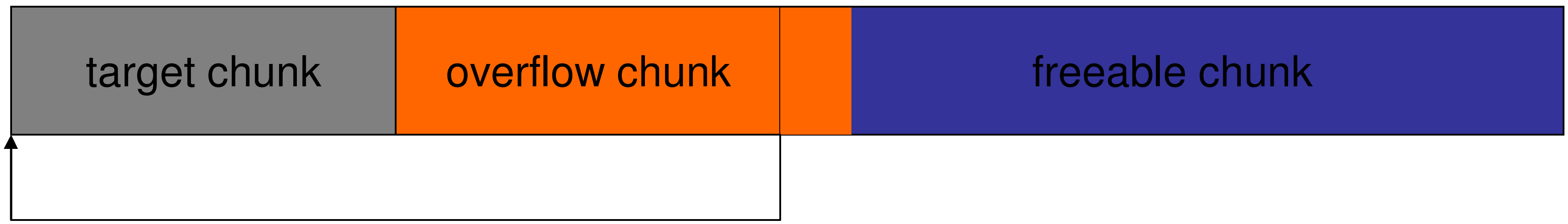
LFH Chunk layout:



- Target chunk and overflow chunk combined must be less than 2040 bytes

LFH header overflow V

LFH Chunk layout:



- Overflow some “freeable” LFH chunk
- Free the overflowed chunk
- Actually frees target chunk…
- So reallocate target chunk and overwrite

LFH header overflow VI

LFH header overflow requirements:

- Ability to write NULL bytes in overflow
- Small target and overflow chunk on LFH
- Some allocation pattern that gives required layout
- Ability to leverage reallocated target chunk