Exploitation 2

Jared DeMott
Crucial Security, Inc.
Black Hat 2008

- Introduction
- Traditional Stack Overflow
- Function pointer Overwrite

Exploitation 1

- Traditional Heap Overflow
- Off-by-ones
- Format String Exceptions

Exploitation 2

- Newer Stack Overflows
  - Return to libc
  - SEH overwrite
- Uninitialized variables
  - Stack or heap
- Integer Errors
  - Conversions or overflows
- Summary

Exploitation 3

Some of the Bug classes

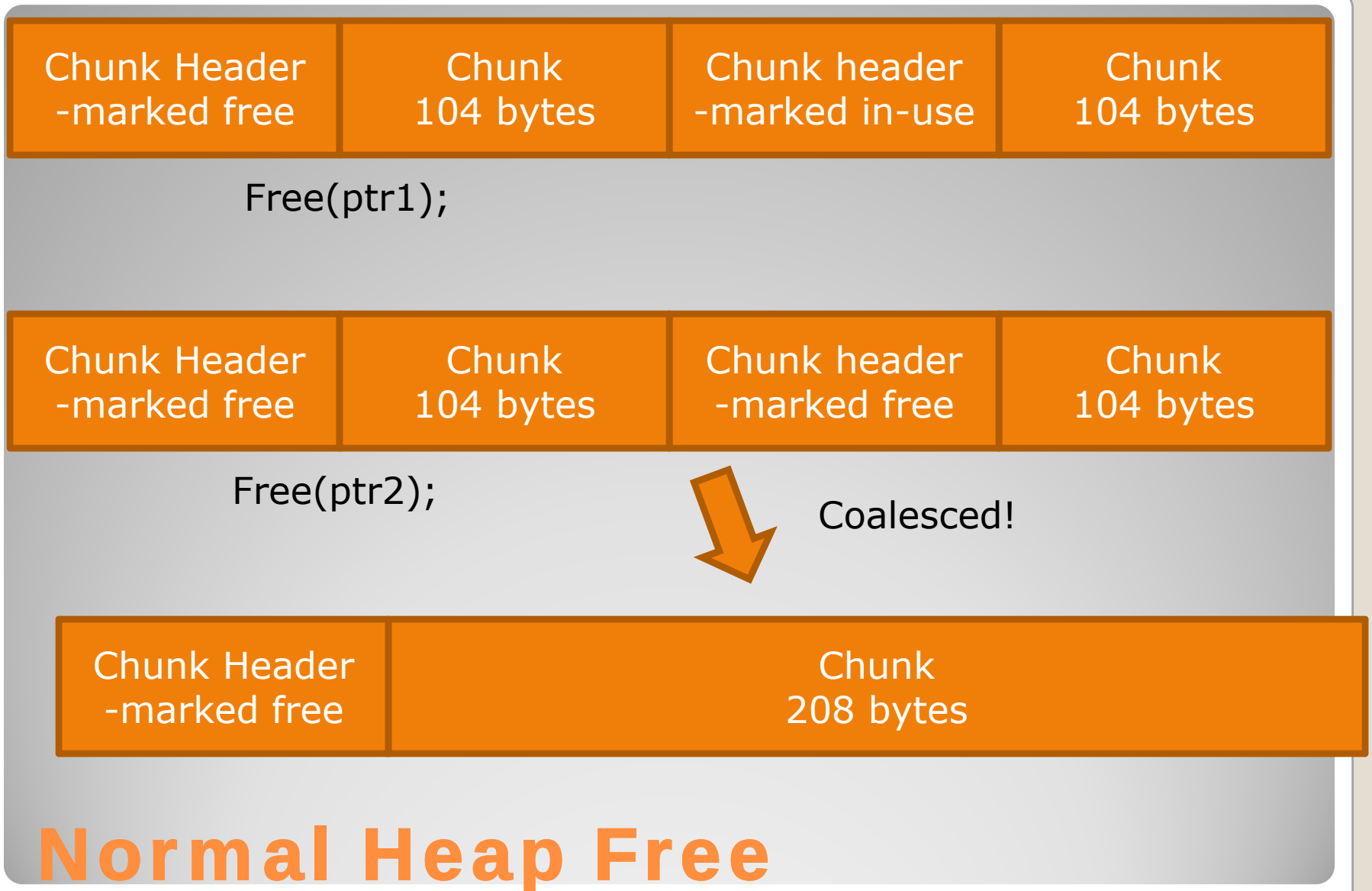
Heap / Stack background

- Overflow buffer on the heap, not the stack
  - Heap is memory returned from *malloc()* style system calls
    - E.g. char * ptr = malloc(100);
  - Heap memory is dynamically allocated and *free()*'ed as the program executes
  - Whereas, Stack memory is a result of local variables
    - E.g. char buf[100];
    - Stack memory is "allocated" upon entrance (call) of function

Heap memory is stored in a doubly-linked list data structure

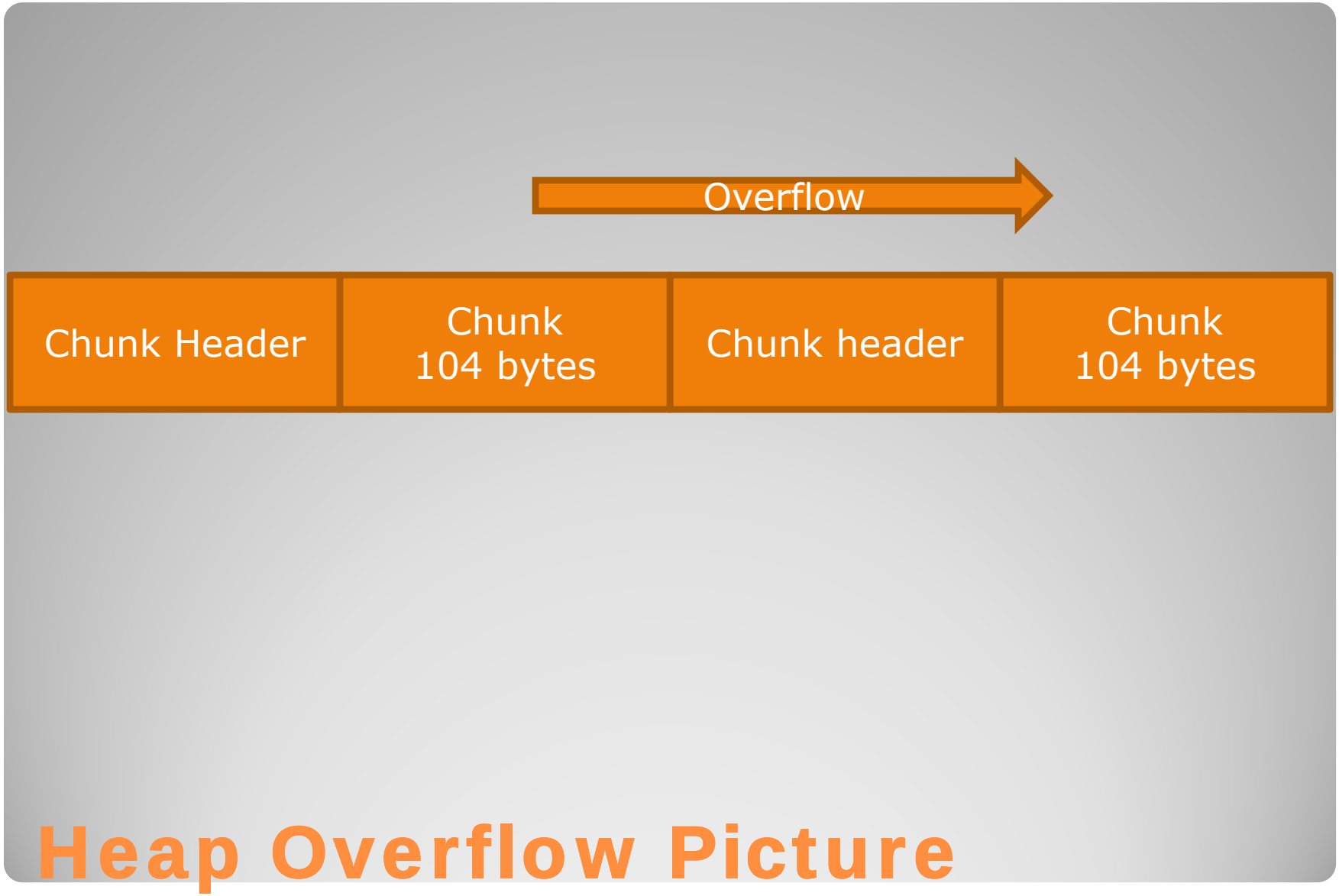
ptr1= malloc(100);
ptr2 = malloc(100);

Normal Heap Allocation


Chunk Header
-marked free
Chunk
104 bytes
Chunk header
-marked in-use
Chunk
104 bytes
Free(ptr1);
Chunk Header
-marked free
Chunk
104 bytes
Chunk header
-marked free
Chunk
104 bytes
Free(ptr2);
Coalesced!
Chunk Header
-marked free
Chunk
208 bytes
Normal Heap Free

Traditional Heap Overflow

- If an application copies data without first checking to see if it fits into the chunk (blocks of data in the heap), the attacker could supply the application with data that is too large, overwriting heap management information (metadata or *control information*) between it and the next chunk
- This allows an attacker to overwrite an arbitrary memory location with four bytes of data
- In traditional environments, this could allow the attacker control over program execution



Heap Overflow Picture

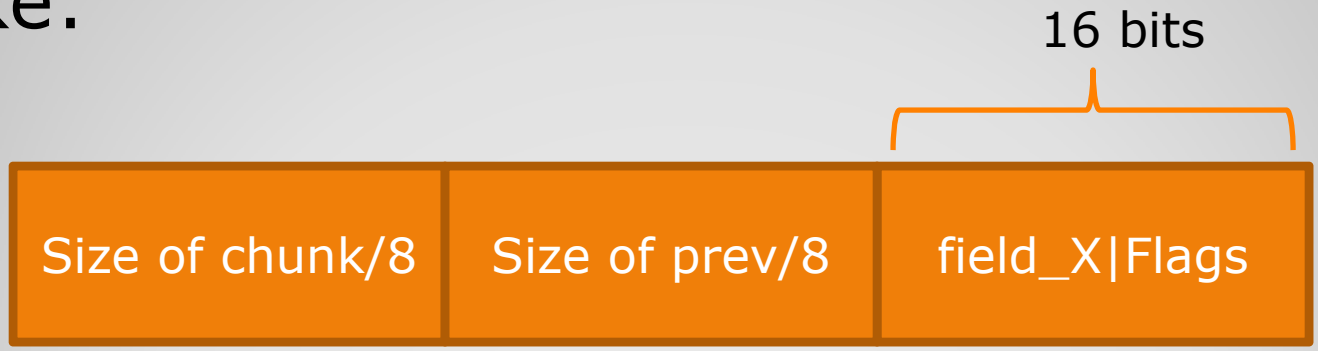
Attacking Heap Algorithms

- Heap Algorithms are somewhat complex
  - Goals are to minimize search time and minimize fragmentation
  - Details vary by platform
- The general gist of an attack:
  1. Chunks are allocated
  2. Control info is "meaningfully" clobbered
  3. Free() must be called at least once
     - This causes "unlinking and coalescation"
       - That is where the aforementioned 4 byte overwrite takes place
  - References for detailed examination:
    - @tlas's 2007 defcon talk for older Unix exploitation
    - Immunity white paper on reliable Win32 Exploitation

- Normally to be vulnerable we need an allocation/overwrite/free pattern something like this:
  - First = AllocChunk(size)
  - Second = AllocChunk(size)
  - Third = AllocChunk(size)
  - Free(First)
  - AllocOverwrite(size)
  - Free(Third)

1. Allocation

- Each control field must be modified to represent a certain field
- The general key here is to create a fake free chunk that will get coalesced (combined) with the adjacent free chunk
- Control headers have a form something like:

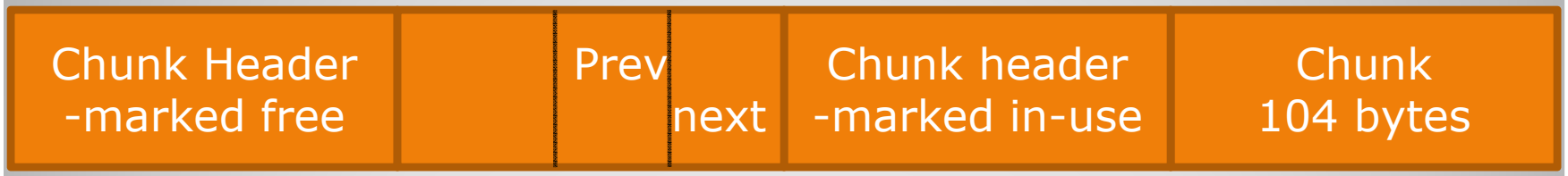


2. Generic Header Overwrite

A more concrete example: Win2k

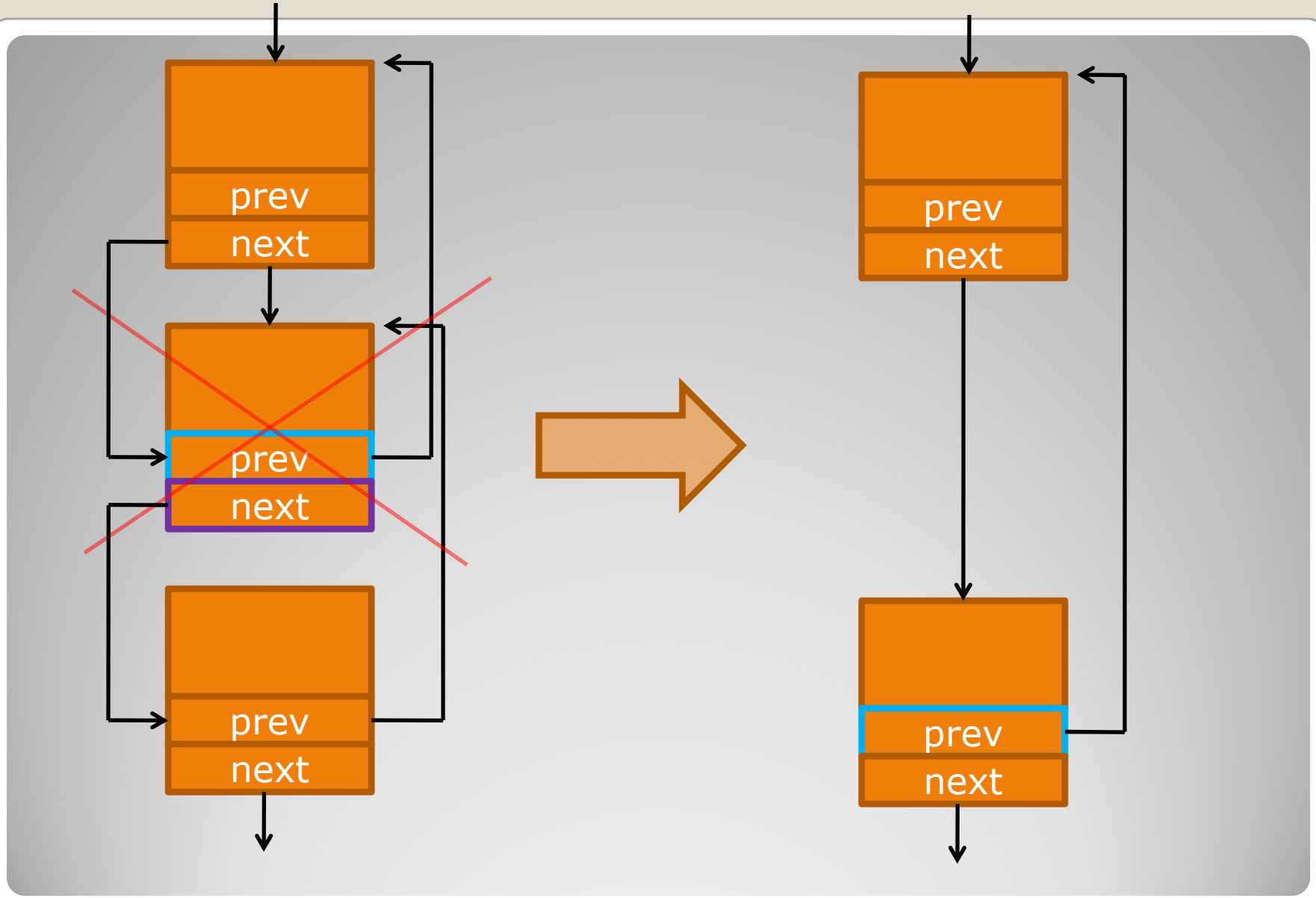
- Reverse engineering ntdll.dll in Windows 2000 revealed the following useful tidbits as it pertains to exploitation:
  - Bit 0 of Flags must be set
  - Bit 3 of Flags must be set
  - Field_X must be smaller than 0x40
  - The first field (own size) must be larger than 0x80
- Figure out a number (such as one with 0x9's in it) that helps us meet the requirements, and allows us to proceed in the unlink and coalesce
- 10011001

- Additional to the control header, once a chunk has been *free'ed()* a previous and a next pointer (in the linked list) get put into the "free" memory area to be used if/when coalesce takes place:



Free(ptr1);

3. How Free() works



Unlinking a doubly-linked list

A free four byte write

- So if we control the contents of a “prev” pointer, and the location at which it will be placed (the “next” ptr), we get a free 4 byte write anywhere to memory (that’s in our virtual address space)
- Viola! Now we can overwrite a GOT pointer or a return address on the stack and we’re well on our way toward exploitation

Heap Exploitation Notes

- Heap overflows are known to be delicate, touchy, very platform dependant, and frankly a bit of a challenge to reliably exploit
  - This is due to the dynamic nature/layout of the heap. It's sometimes difficult to predict if there will be neighbors to properly coalesce with
- Some exploits may have to be run multiple times, or with "staging" packets to setup/configure the heap layout to be in a known exploitable state

Heap protection examples

- Heap integrity checking
  - Recent releases of GNU libc (which incorporate the Doug Lea allocator) can detect heap overflows after the fact
    - Abort program execution if detected
- Since XPSP2 heap protection

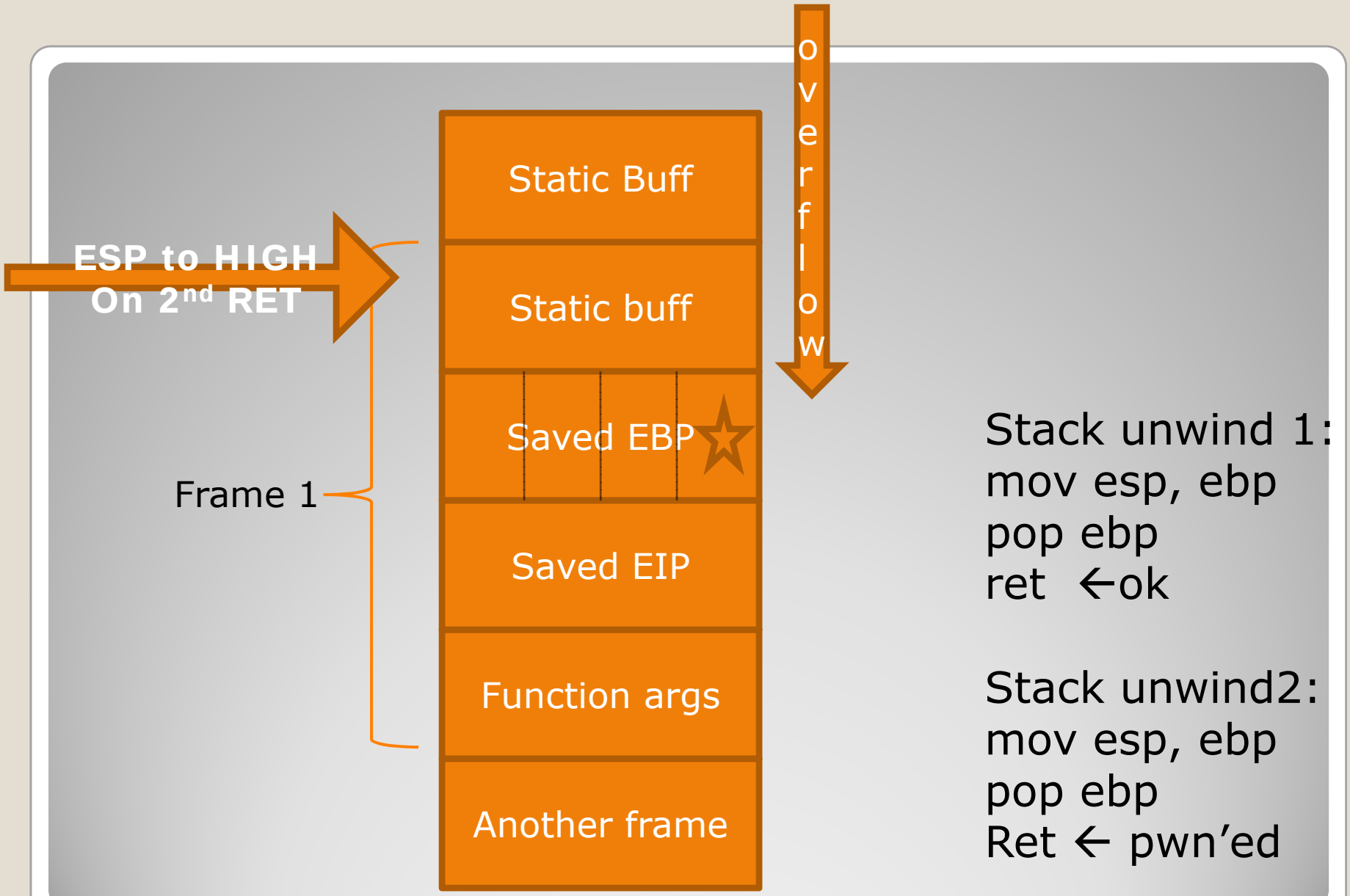
Off-by-one

- Also known as a frame pointer overwrite
- Special case of stack overflow
  - Only clobbers the LSB of EBP
- Was typically exploitable on x86
- The gist is that the stack pointer slides back up into user supplied data during the return address unwind due to the shorted base pointer
  - Code and Diagram on next slides

```
void foo (char *s) {
    char buf[15];
    memset(buf, 0, sizeof(buf));
    strncat(buf, s, sizeof(buf)); //should be: sizeof(buf)-1
    return;
}
```

- *strncat()* is a frequent culprit because the usage of the 'n' style function feels safe
  - But actually *strncat* will add the null byte one beyond the boundary of buf, creating the off-by-one overflow

Code snippet



Picture of off-by-one

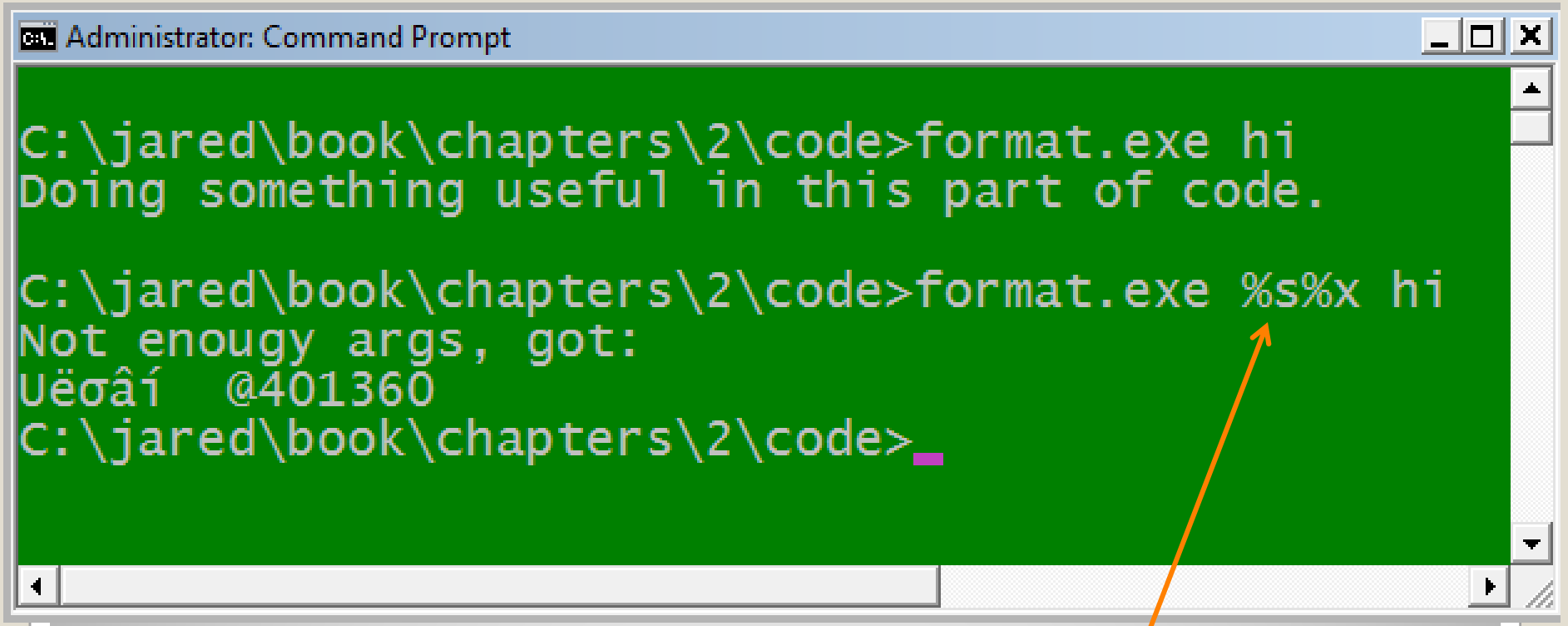
Format String Exceptions

- Result when user input is directly passed to a *printf* style function without a "format string"
  - Could be used for data leak or exploitation
- Example of good call:
  - printf("%s", user_supplied_buffer);
- Example of bad call:
  - printf(user_supplied_buffer);
- No registers/buffers are overflown so immune to stack canaries

```
int main(int argc, char * argv[])
{
  if( argc !=2 )
  {
       printf("Not enougy args, got:\r\n");
       printf(argv[1]);
       exit(-1);
  }

  printf("Doing something useful here.\r\n");
}
```

Format Example Code



As these are changed varying information could be extracted off the stack or even modified if the %n is used. Could be used to overwrite a GOT pointer or return address

Format example output

Format Defense

- Easy for static source code auditing tools to find
  - They have been nearly hunted to extinction already
- Plus other defenses like ASLR/NX would make them difficult to exploit

Summary

- In the past, most important system code was written in C. Win/Unix kernel still written in C/C++
- Few/No protections were in place
- Systems are better protected today
- However, that means what working 0days are still being discovered and used, have moved further underground due to the value of such attacks
  - Very determined hackers can often still find a way even in the midst of solid protections