



# Vulnerabilities, Exploits, and Attacks

Úlfar Erlingsson

Microsoft Research, Silicon Valley

U.C. Santa Cruz, April 2006

Microsoft  
**Research**

- ● ● | Assumptions are vulnerabilities

- How to successfully attack a system
  - 1) Discover what assumptions were made
  - 2) Craft an exploit outside those assumptions
- Two assumptions often exploited:
  - A target buffer is large enough for source data
  - Integer numbers behave like they do in math
- (I.e., buffer overflows & integer overflows)

# ● ● ● | Assumptions on control flow

- We write our code in high-level languages
- **But, actually, at the machine code level**
  1. Can start in the middle of functions
  2. A fragment of a function may be executed
  3. Returns can go to any program instruction
  4. All the data has usually been executable
  5. On the x86, can start executing not only in the middle of functions, but middle of instructions!



## Demo

# Microsoft's GDI+ JPEG bug

(Bug was fixed a little before Windows XP SP2)

- Vulnerability
- Exploit
- Example attack



# What is a buffer overrun?

- The ability to arbitrarily corrupt memory
- Overflows lead to arbitrary code
- Underflows lead to denial of service
- Problem is usually isolated to C and C++

```
int x = 42;  
char zip[6];  
strcpy(zip, userinput);  
printf("x = %i\n", x);
```

2A	00	00	00
00	00	00	00
00	00	00	00

# Anatomy of the stack

Previous function's  
stack frame

Function arguments

Return address

Frame pointer

EH frame

Local variables and  
locally declared  
buffers

Callee save  
registers

Garbage

- x86 stacks grow downward
- A buffer overrun on the stack can always rewrite the:
  - Return address
  - Frame pointer
  - EH frame

# Stack smashing

```
#define BUFLen 4

void vulnerable(void) {
    wchar_t buf[BUFLen];
    int val;

    val = MultiByteToWideChar(
        CP_ACP, 0, "1234567",
        -1, buf, sizeof(buf));
    printf("%d\n", val);
}
```

Attack Code

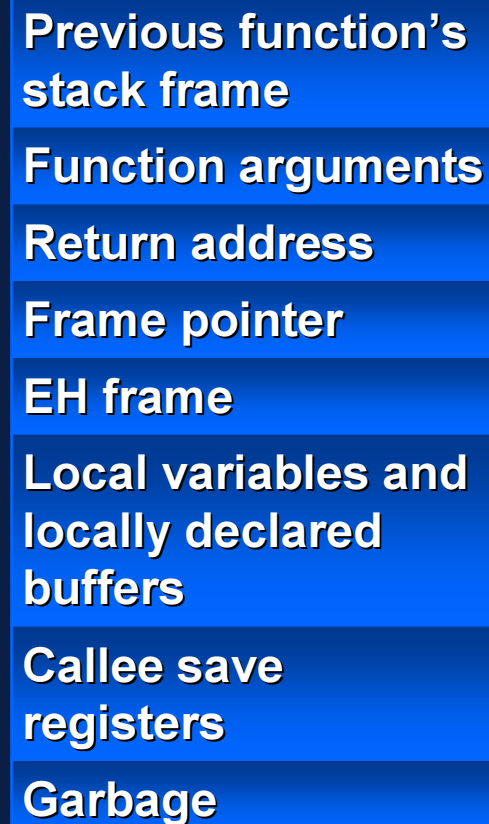
Hijacked EIP

Garbage  
with invalid cookie

Garbage

# Types of exploits

- Stack smashing
- Register hijacking
- Local pointer subterfuge
- V-Table hijacking
- C++ EH clobbering
- SEH clobbering
- Multistage attacks
- Parameter pointer subterfuge





## Unsafe APIs

- o Many historical APIs of the C standard library are bad
  - `strcpy` does not know the array size
  - `strncpy` cannot validate the array size
  - Many more unsafe APIs exist
- o Static analysis tools are helpful
- o ~~Impossible to guarantee a safe API~~

*Challenge*

[Jones Kelly 97]

[Ruwase Lam 04]



# Stack layout in VC++ .NET

## Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
      [___security_cookie (408040h)]
xor    eax,dword ptr [esp+24h]
mov    dword ptr [esp+20h],eax
```

## Function epilog:

```
mov    ecx,dword ptr [esp+20h]
xor    ecx,dword ptr [esp+24h]
add    esp,24h
jmp    ___security_check_cookie (4010B2h)
```

Previous function's  
stack frame

Function arguments

Return address

Frame pointer

Cookie

EH frame

Local variables and  
locally declared  
buffers

Callee save  
registers

Garbage

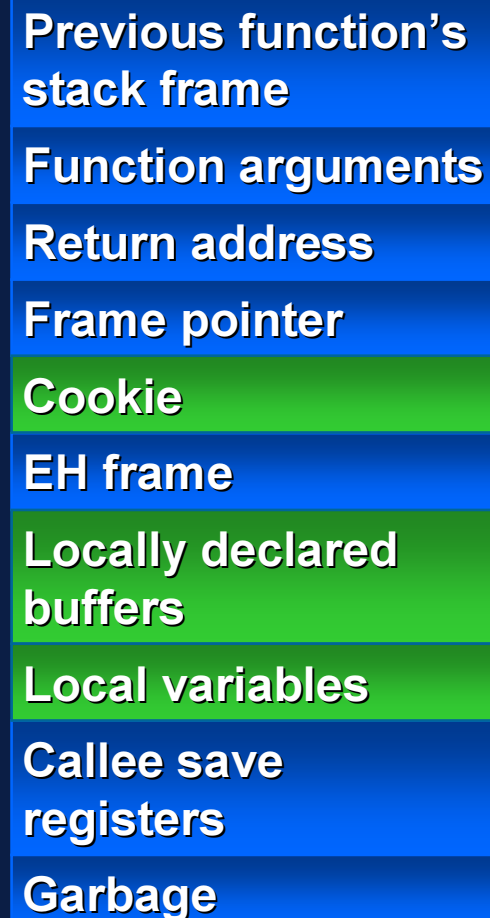
# Stack layout in VC++ 2003

## Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
      [__security_cookie (408040h)]
mov    dword ptr [esp+20h],eax
```

## Function epilog:

```
mov    ecx,dword ptr [esp+20h]
add    esp,24h
jmp    __security_check_cookie (4010B2h)
```





# What is this cookie?

[Cowan et al 98]

- o Generated by the function `__security_init_cookie`
- o Original stored in the variable `__security_cookie`
- o Cookie is random (at least 20 bits)
- o Cookie is per image and generated at load time
- o Cookie is the size of a pointer



## Performance impact

- Expect less than a 2% degradation
- Most application did not notice anything
- With both VC7 and VC7.1 improvements in optimization make up for these security checks
- Each security check is nine instructions

**“The perf hit hasn’t shown up for us. There was no test hit associated with the change. The only cost we’ve had associated with this is getting ourselves to build with /GS.**

**– IIS6 Developer**

# V-Table hijacking

```
class Vulnerable {
public:
    int value;
    Vulnerable() {value=0;}
    virtual ~Vulnerable()
        {value=-1;}
};

void vulnerable(char* str) {
    Vulnerable vuln;
    char buf[20];
    strcpy(buf, str);
}
```

Attack Code

Hijacked V-Table  
& Hijacked V-Table

Garbage

Garbage

# ● ● ● | Pointer subterfuge

```
void vulnerable(  
    char* buf, int cb)  
{  
    char name[8];  
    void (*func)() = foo;  
  
    memcpy(name, buf, cb);  
    (func)();  
}
```

Attack Code

&Attack Code

Garbage

Garbage

# EH clobbering

```
int vulnerable(char* str) {
    char buf[8];
    char* pch = str;
    strcpy(buf, str);
    return *pch == '\0';
}

int main(
    int argc, char* argv[]) {
    __try {
        vulnerable(argv[1]);
    } __except(2) { return 1; }
    return 0;
}
```

Attack Code

Hijacked EH frame

Garbage

0xBFFFFFFF

Garbage

Garbage



# Exploits possible despite /GS

- Parameter pointer subterfuge
- Two stage attacks
- Local objects with buffers
- Heap attacks
- ...
- ...
- ...



## Class of x86 injection attacks

- Attacker controls victim behavior by getting x86 machine code of their choice to execute in victim's environment
- Subverts execution trace
  - Different x86 machine instructions execute



- Not the only attack: e.g., scripts





## NX: x86 Code Lockdown

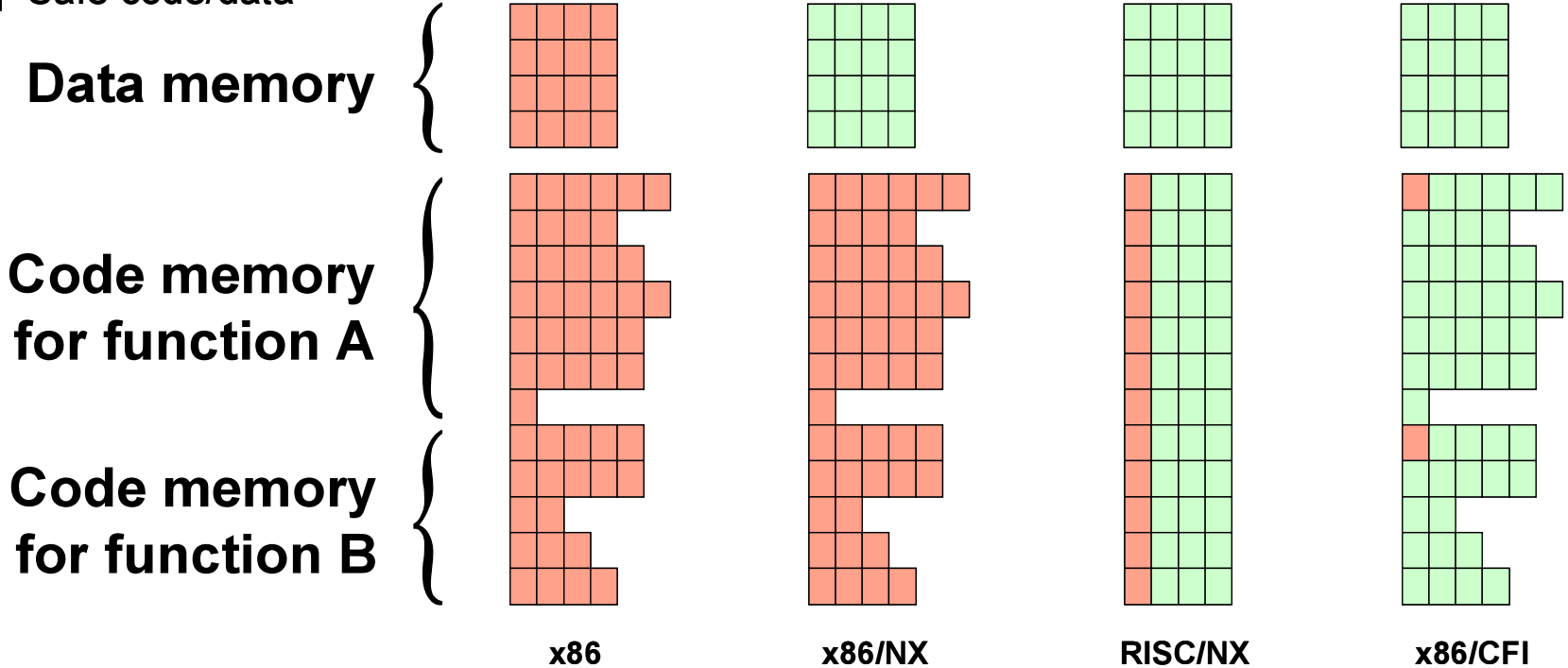
- Distinguish between code and data
  - Harvard architecture?
- Prevent data from being executed as x86 machine code
- Slight modification to hardware RM validity checks



# What bytes will CPU interpret?

-  Possible control flow destination
-  Safe code/data

## Possible Execution of Memory





# Implementing NX

- Piece of cake with SFI (albeit slow)
- Hardware support on many CPUs
  - IA-64 (more realistically on amd64 aka x64)
- Breaks lots of software:
  - Most Win32 GUI apps, CLR (and JITs)
- Can synthesize on IA-32 chips
  - Mark all data pages non-touchable
  - On trap, temporarily mark read/write, touch with MOV (which loads D-TLB), revert the page back to being untouchable



# Circumventing NX (aka jump-to-libc)

- Don't introduce new code (at first)

- Script existing code!

- E.g.

```
Victim's arg 2  
Victim's arg 1  
Victim's return addr
```



```
4th Function Args  
4th Function Args  
garbage (4th func is end)  
3rd Function Args  
3rd Function Args  
4th Function Address  
2nd Function Args  
2nd Function Args  
3rd Function Address  
1st Function Args  
1st Function Args  
2nd Function Address  
garbage  
garbage  
1st Function Address
```

- VirtualAlloc exec page,  
then InterlockedExch,  
then memcpy, then  
jump to alloc'd page



# Address-space Randomization

[PaX]

- x86 code injection attack must target the last “good” machine instruction
  - What happens just before exploit starts ?

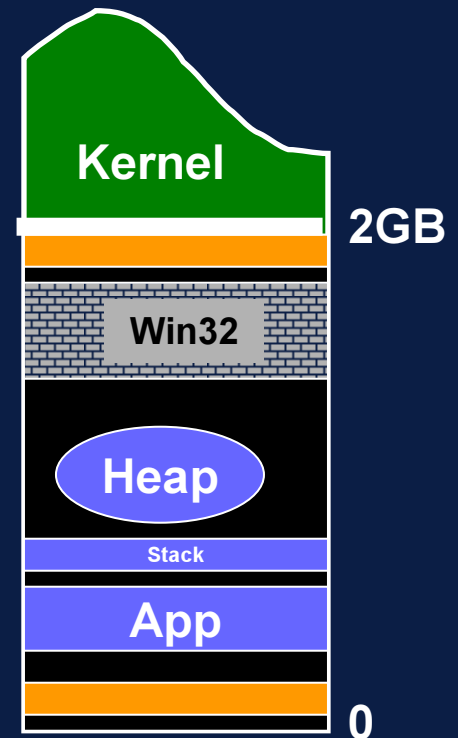


- It's control flow to an absolute address
  - Call [EAX], Jmp [EBX], Ret (implicit [ESP])
- Attacker must know where to go !!!



# What absolute addresses ?

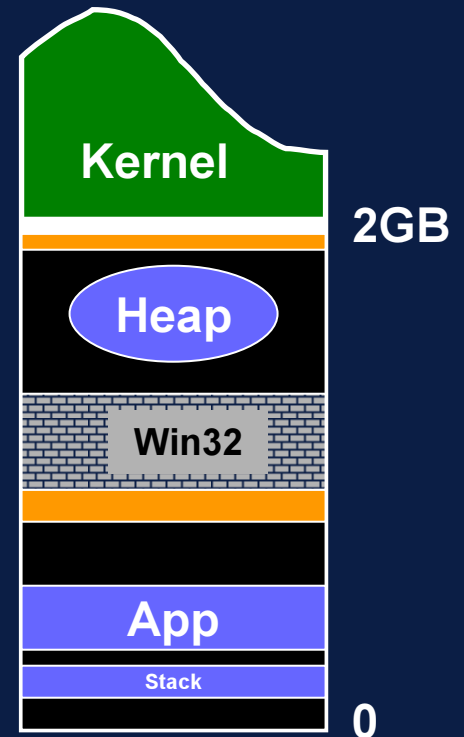
- Attacker examines victim's address space on his/her machine:
  - For a version of Windows each address space is mostly the same (Win32 & friends)
  - Also, apps always lay out executables, stack, heap in the same way
- Attacker crafts exploit given above; waits for a vulnerability





# Randomization / Rebasing

- Windows allows most things to be relocated
- Can do it
  - Dynamically @ load
  - Statically @ install
- Problems:
  - Most EXE files cannot be moved at all... etc.
- Example of Edit Automata





# Circumventing ASLR

- Learn the memory layout specifics of your target for attack [Durden 02]
  - Possible using format string attack etc.
  - May leak accidentally, e.g., via “nonce” in a protocol or Windows error reporting
  - Epidemic can automatically craft code
- Use brute force [Boneh et al. 04]
  - Only 16 bits of shuffle on 32-bit machines
  - Easy to exhaust keyspace (automatically)

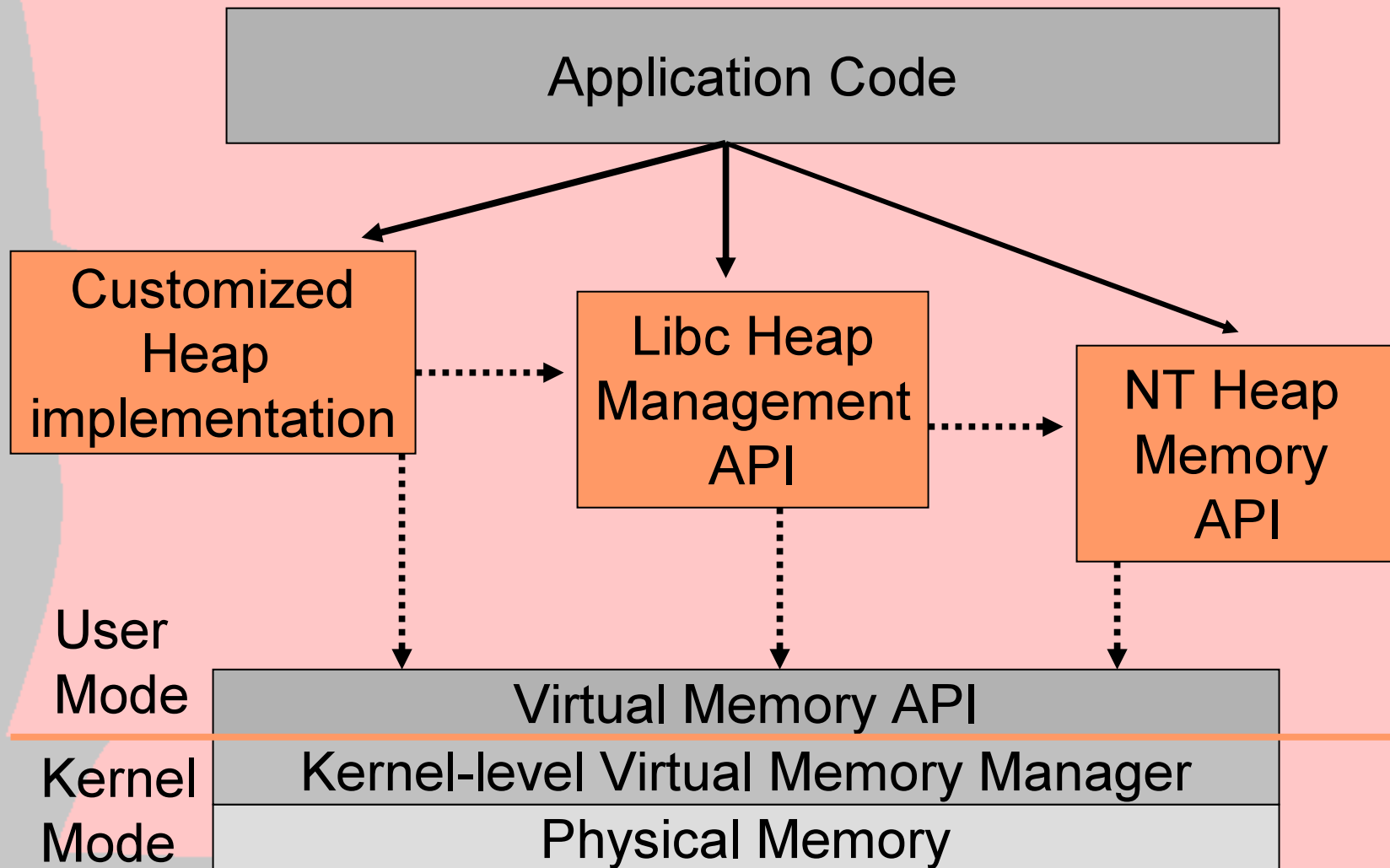
# Sample Vulnerability Discovery

[SolarDesigner01] [Slides, Halvar Flake 02]

- Surprisingly easy to find vulnerability
- Vulnerability types often translate
  - Heap manager problem in glibc.so
  - Heap problem in Windows 2000

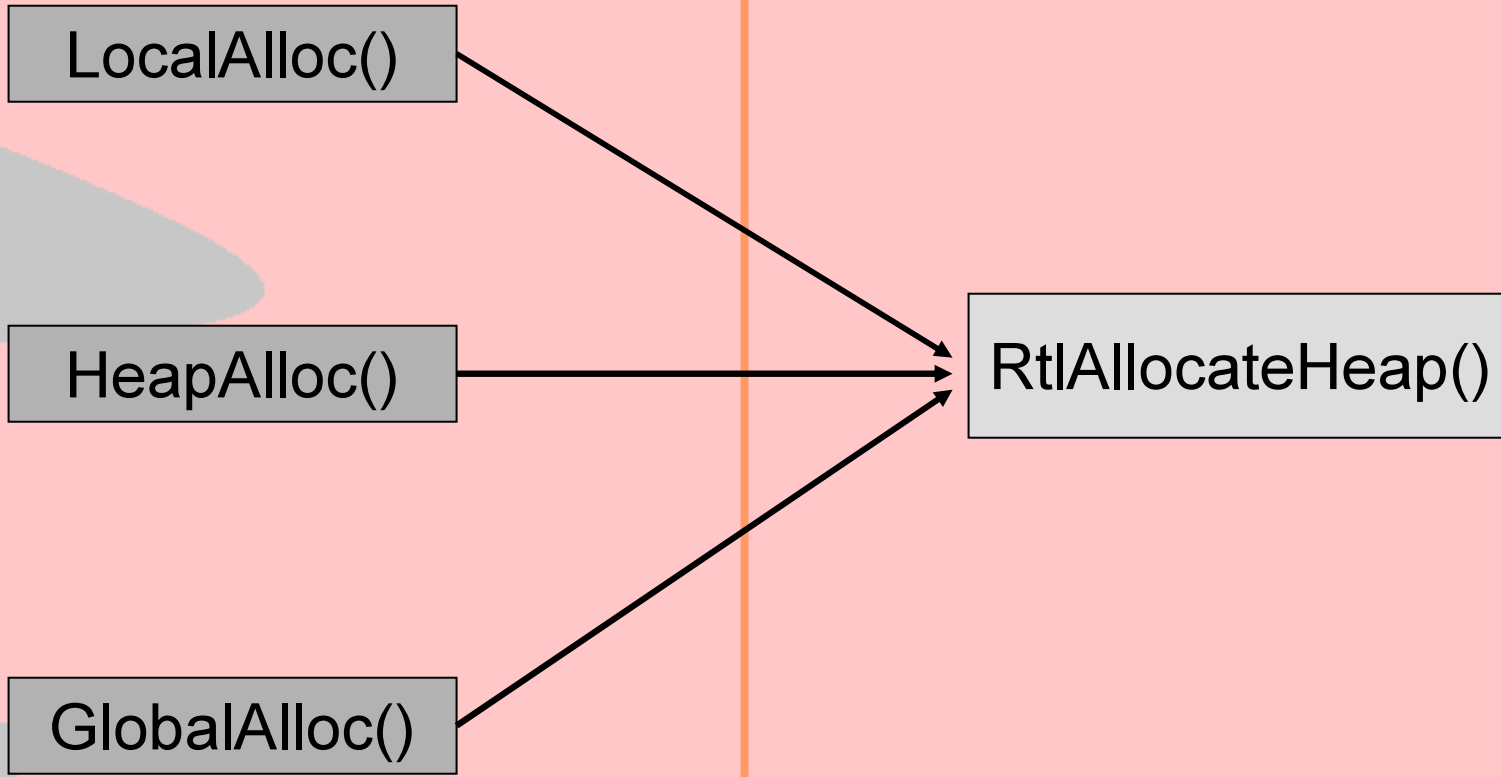
# Heap Structure Exploit Generalities

## Win32 heap management model



# Heap Structure Exploits

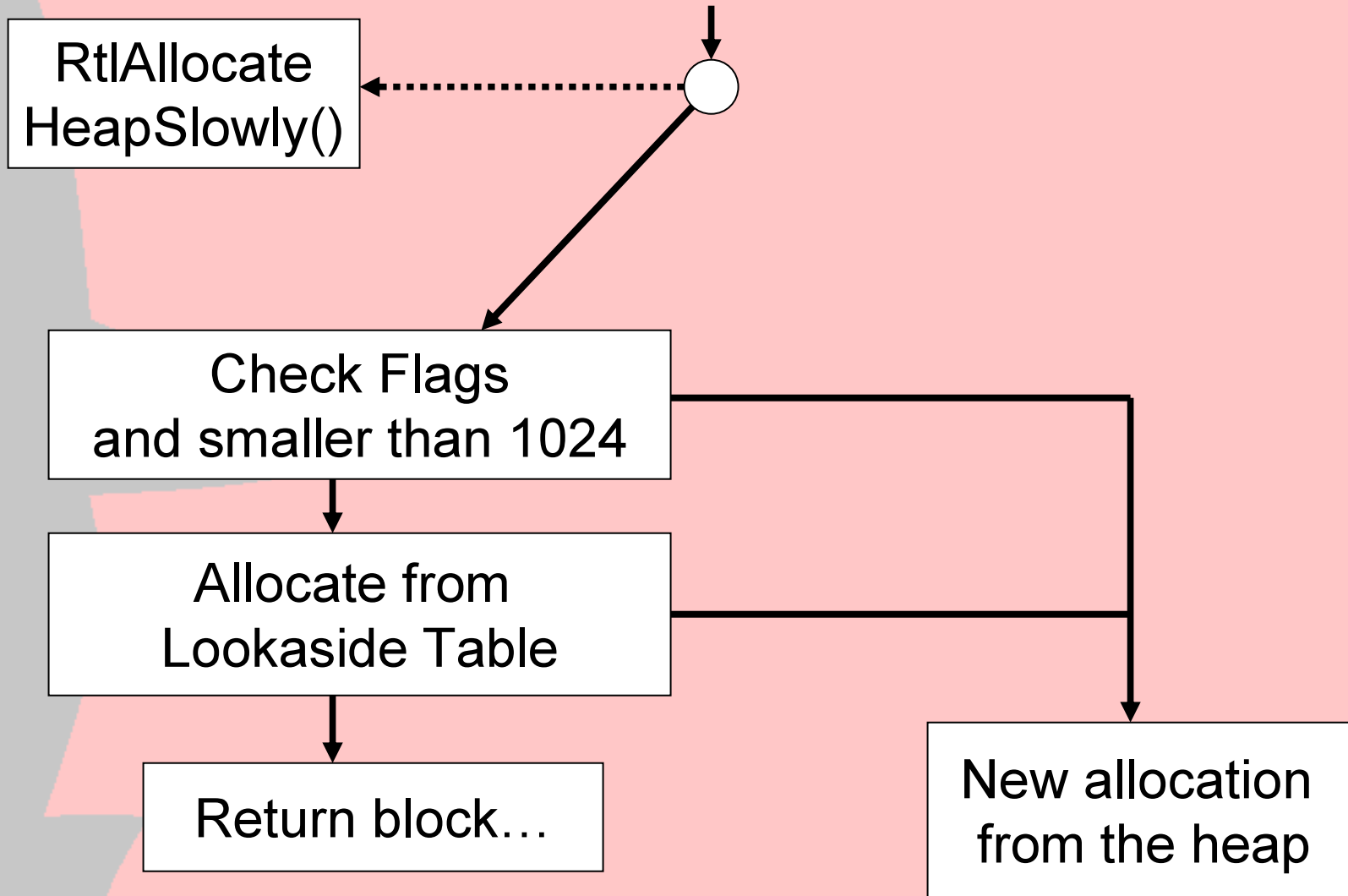
## Win2k Heap Manager (I)



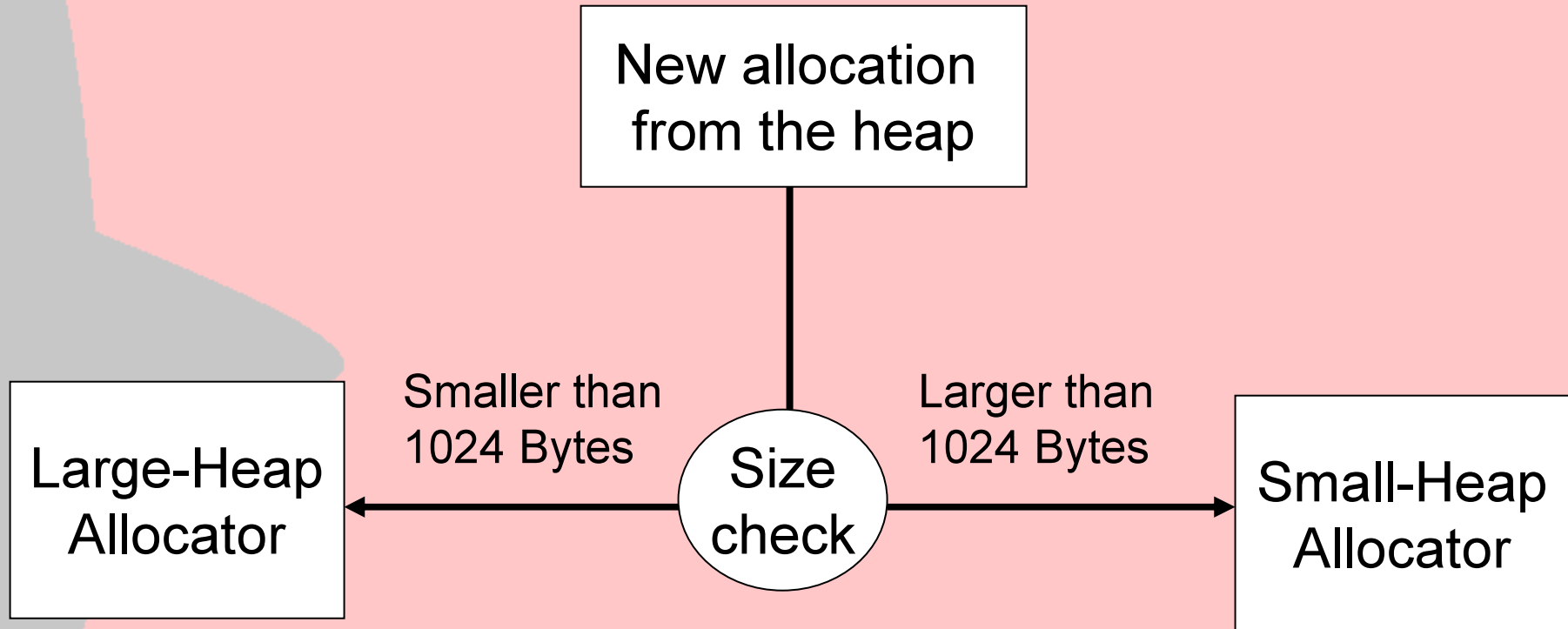
Kernel32.DLL

NTDLL.DLL

# RtlAllocateHeap (I)



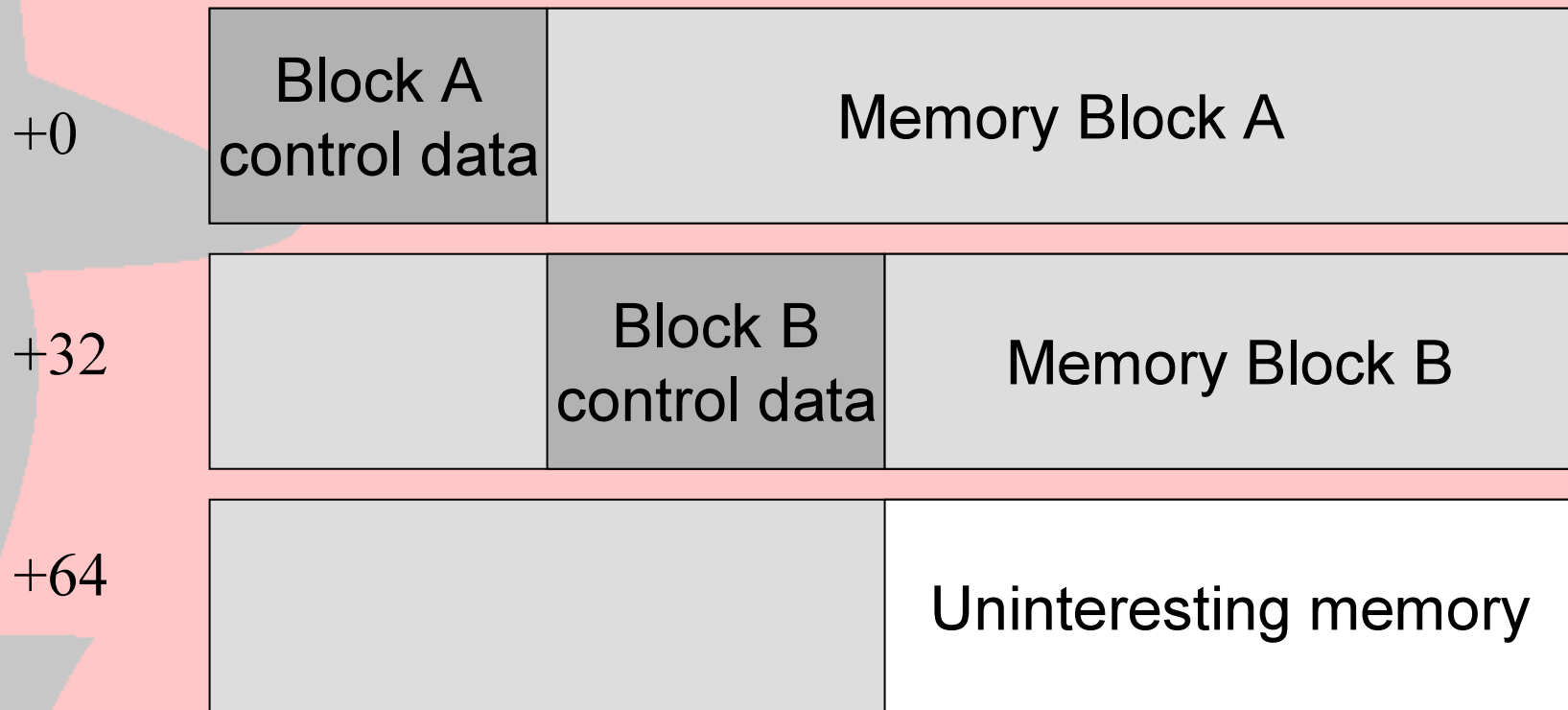
# RtlAllocateHeap (II)



# Heap Structure Exploits

## Win2k Heap Manager (II)

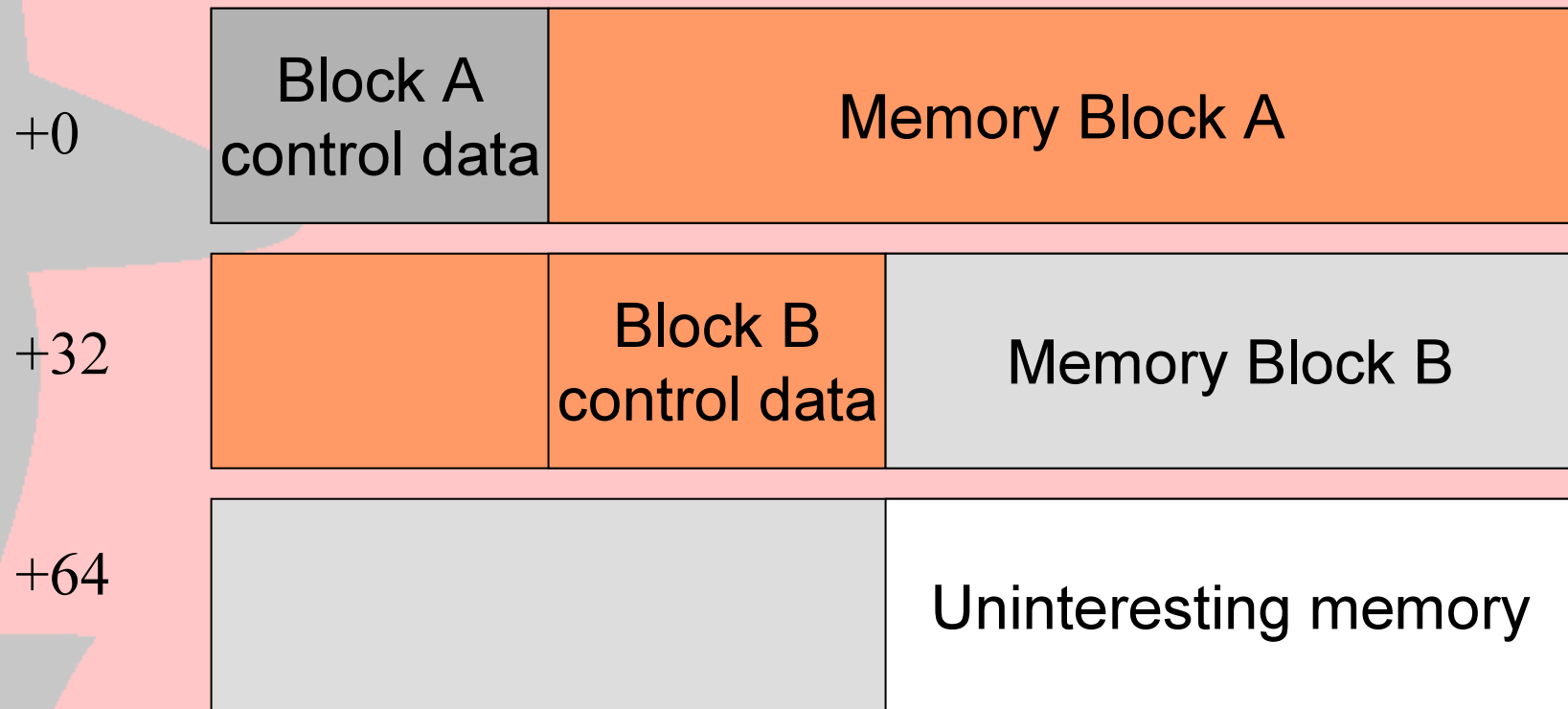
After two allocations of 32 bytes each our heap memory should look like this:



# Heap Structure Exploits

## Win2k Heap Manager (III)

Now we assume that we can overflow the first buffer so that we overwrite the *Block B control data*.



# Heap Structure Exploits

## Win2k Heap Manager (IV)

When Block B is being freed, an attacker has supplied the entire control block for it. Here is the rough layout:



If we analyze the disassembly of `_RtlHeapFree()` in `NTDLL`, we can see that our supplied block needs to have a few properties in order to allow us to do anything evil.

# Heap Structure Exploits

## Win2k Heap Manager (V)

Properties our block must have:

- Bit 0 of Flags must be set
- Bit 3 of Flags must be set
- Field\_4 must be smaller than 0x40
- The first field (own size) must be larger than 0x80

The block 'XXXX99XX' meets all requirements.

We reach the following code now:

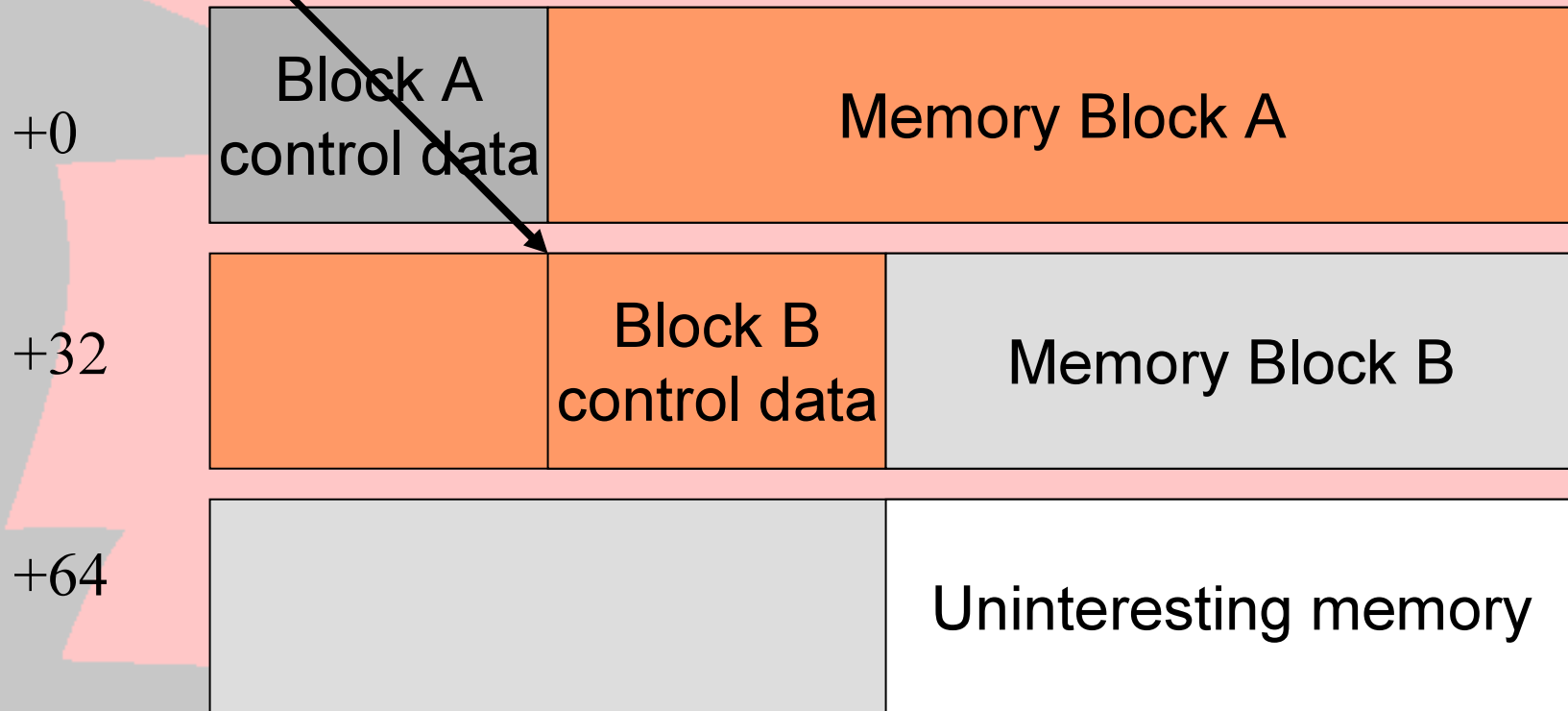
# Heap Structure Exploits

## Win2k Heap Manager (VI)

```
add esi, -24
```

ESI points here

...now here ...

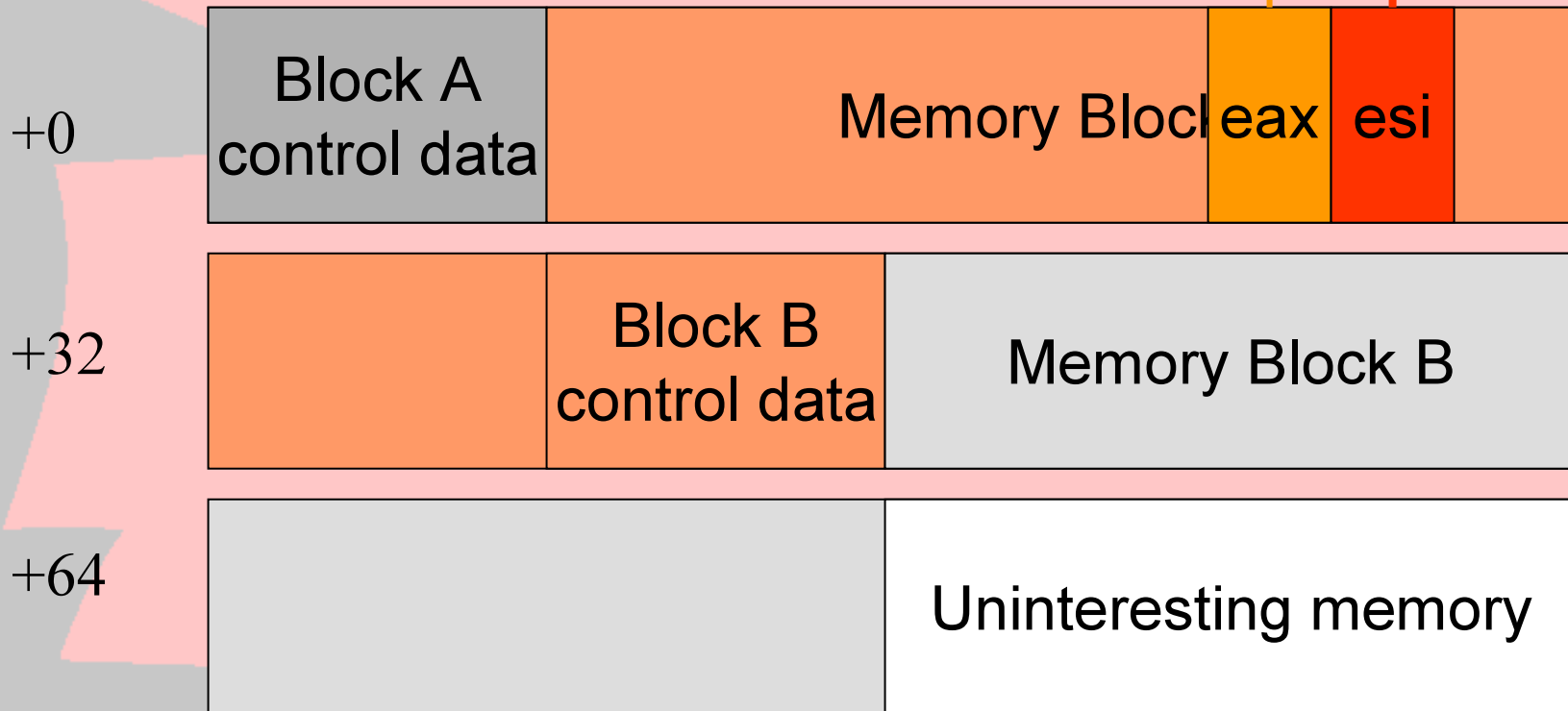


# Heap Structure Exploit

## Win2k Heap Manager (VII)

```
mov    eax,[esi]
```

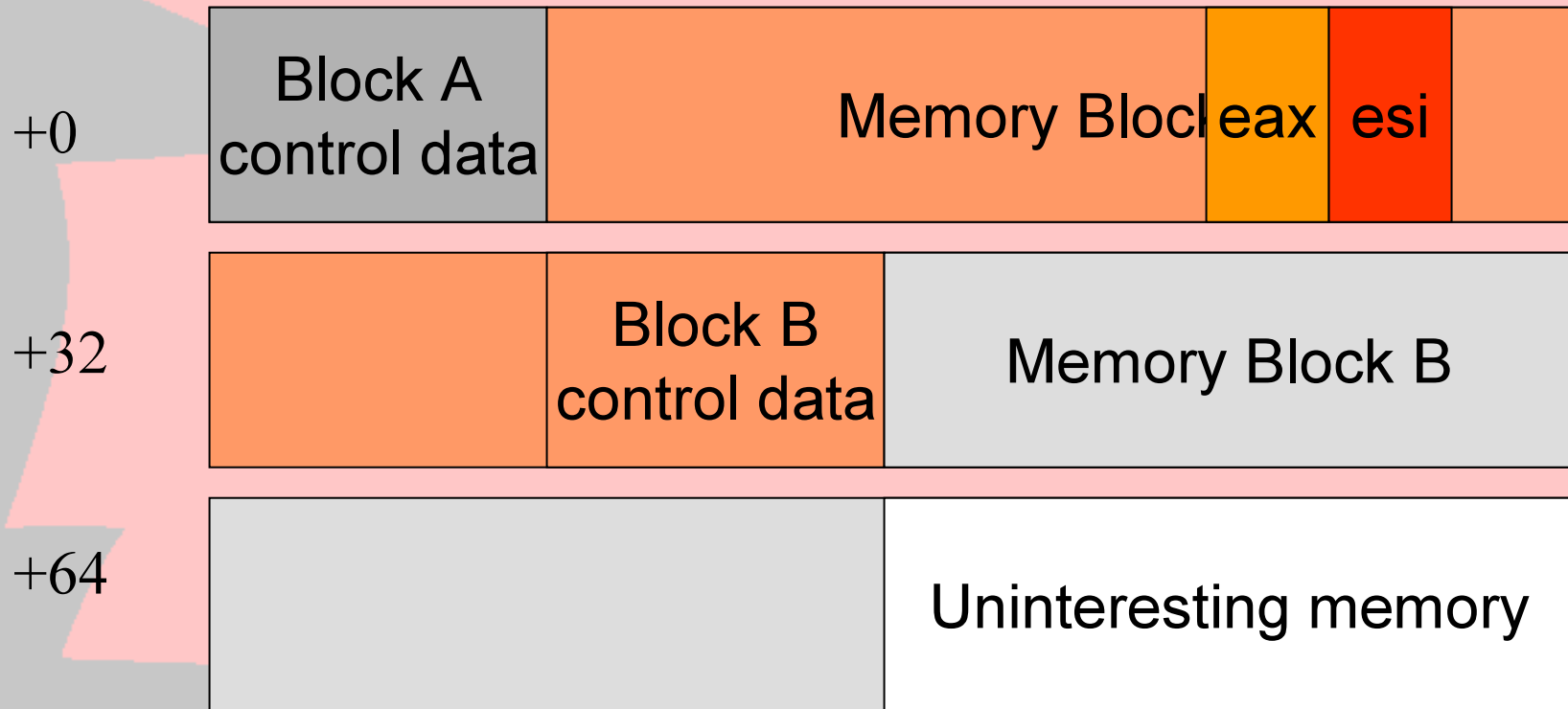
```
mov    esi,[esi+4]
```



# Heap Structure Exploits

## Win2k Heap Manager (VIII)

`mov [esi], eax ; Arbitrary memory overwrite`



# Heap Structure Exploits

## Win2k Heap Manager (IX)

- If we can overwrite a complete control block (or at least 6 bytes of it) and have control over the data 24 bytes before that, we can easily write any value to any memory location.
- It should be noted that other ways of exploiting exist for smaller/different overruns – use your Disassembler and your imagination.



## Demo

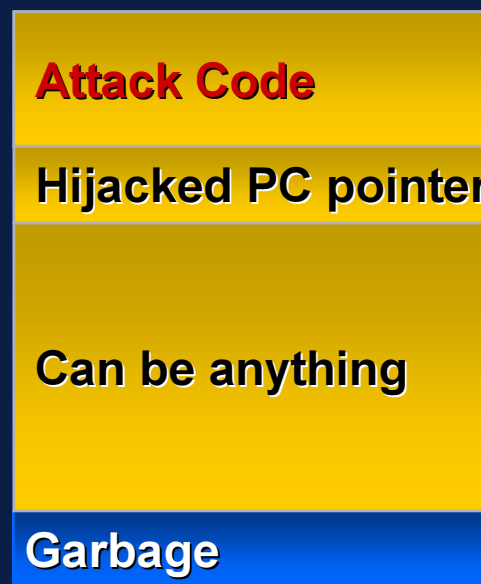
Reverse-engineering machine  
code is surprisingly easy !

# Machine code attacks

- About 60% subvert expected machine-code control flow



- E.g., overflow buffer to overwrite return address on the stack
- Other vulnerabilities can also be exploited to hijack execution



## Mitigations

**NX** prevents data memory execution

**/GS** checks return pointer hasn't been overwritten

# World's most dangerous code?

- Code is unsafe despite NX and /GS
  - Vulnerable code may slip past security reviews
  - Can't eliminate bugs in 3rd-party & legacy code

```
int median(int* data, int len, void* cmp)
{
    int tmp[MAX_LEN];
    assert(len <= MAX_LEN);
    memcpy(tmp, data, len*sizeof(int));
    qsort(tmp, len, sizeof(int), cmp);
    return tmp[len/2];
}
```

- Want to prevent tomorrow's exploits ...