Third Generation Exploitation

Smashing the Heap under Win2k

Blackhat Briefings Windows 2002

Halvar Flake
Reverse Engineer
Blackhat Consulting

Third Generation Exploits

Overview (I)

- **Introduction**
  - First Generation Exploits
  - Second Generation Exploits
  - Third Generation Exploits
- **Heap Structure Exploitation**
  - Generalities
  - Win2k Heap Manager
  - Borland C++ libc
  - Demonstration
  - The future of Exploitation

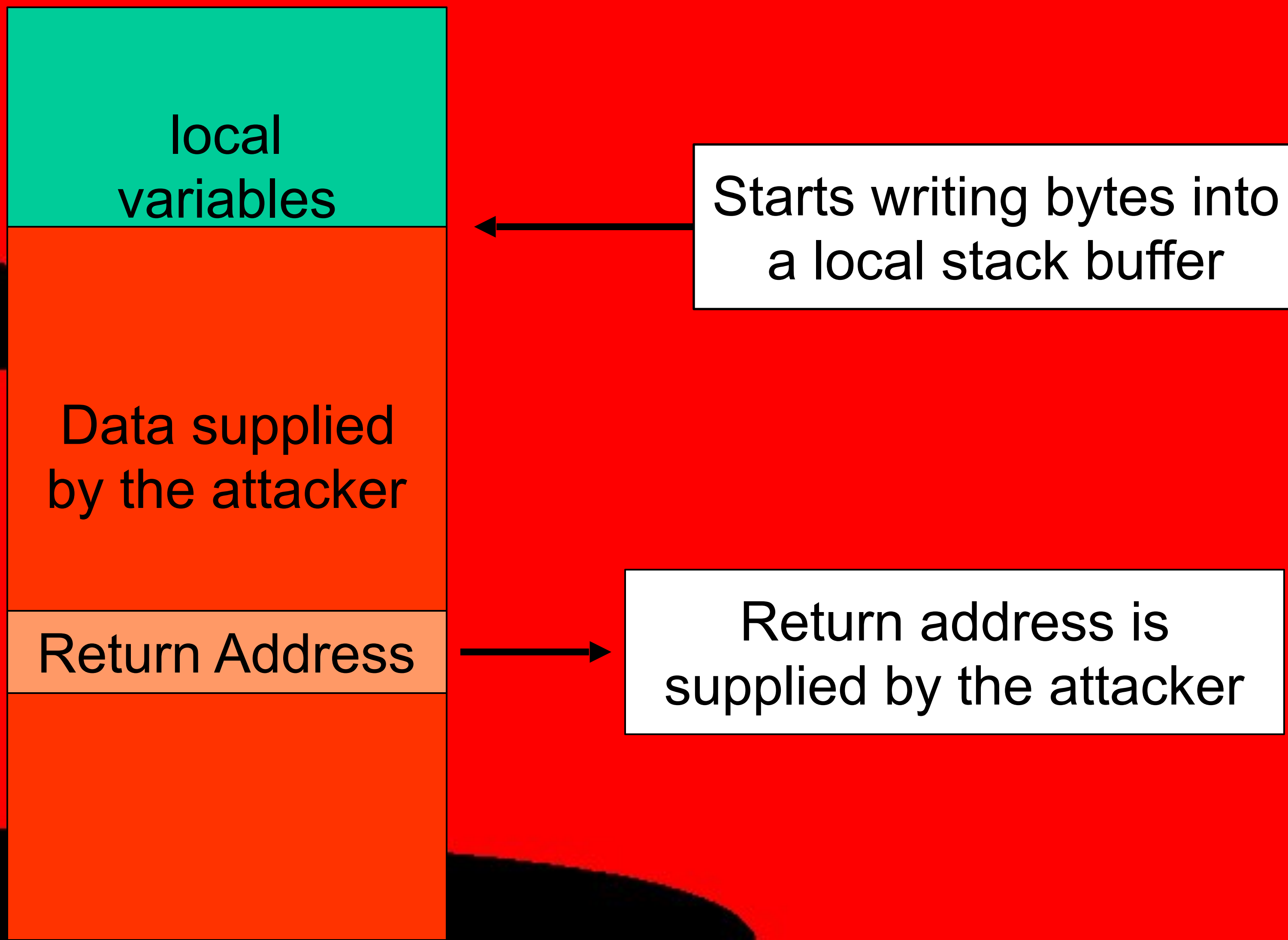
Third Generation Exploits

Overview (II)

- **Format String Bugs**
  - History
  - Automated Detection
  - Exploitation
- **Exploitation reliability**
  - Problem definition
  - **U**nhandled **E**xception **F**ilter **A**ttack
  - **T**hread **E**nvironment **S**tructure **O**verwrite
  - Free time for questions, answers and discussions

Introduction

First Generation Exploits (I)



Introduction

First Generation Exploits

- Simple stack smashes
- Documented *ad nauseam*
- EIP completely taken
  Hardware-specific feature (e.g. RET instruction)
- *strcpy()*, *gets()*, *sprintf()* …
- Trivial to exploit
- Can be detected via stress-testing
- Bug Species almost extinct

Introduction

Second generation exploits

- Cast screw-ups, off-by-one's
- *strncat()*, *strncpy()*, manual pointer handling, …
- Fairly well documented
- EIP not overwritten, EBP manipulated
  - Compiler functionality (e.g. standard function prologue/epilogue for C compilers)
- Can be quite hard to detect, but can be detected via stress testing
- Takes control of execution after a small detour
- Due to the hard-to-find nature, a few of these are still around

Introduction

Off-by-one-exploitation (I)

saved_EBP‘s lowest byte is set to 0x00

Function epilogue: **mov esp, ebp**

Introduction

Off-by-one-exploitation (II)

Function epilogue: **pop ebp**

Introduction

Off-by-one-exploitation (III)

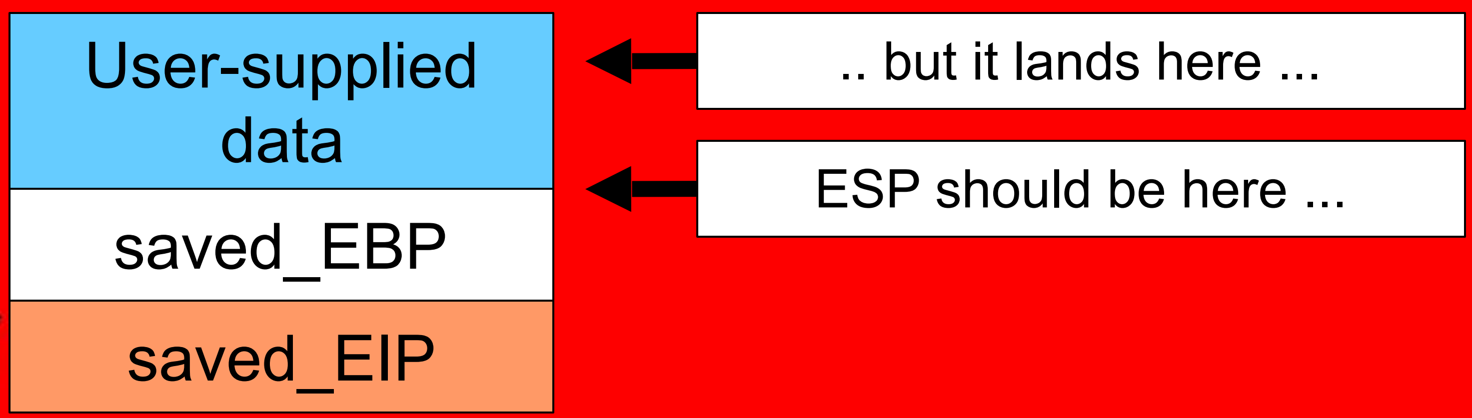
The value in EBP (the *frame pointer*) is now our modified value !

Function epilogue: **ret**

Introduction

Off-by-one-exploitation (IV)

Next function epilogue:**mov esp, ebp**
ESP slides upwards (as its lowest order byte was overwritten) into the user-supplied data. We can now supply a new return address to gain control



Introduction

Third Generation: Format Strings

- New bug class surfaced in Summer 2000
- **printf()* - family functions
- Trivial to spot
- Fairly well-documented and widely exploited
- [illegible]s reading from & writing to arbitrary addresses
- No CPU registers overwritten
  - Specific libc-functionality which is documented in the ANSI/ISO C specification
- Simple to exploit, powerful, easy to find → hunted to extinction within a very short time

Introduction

Third Generation:
Heap Structure Exploits

- Publically documented by Solar Designer
- Takes advantage of libc-specific implementations for malloc()/free()
  - More abstract than Generation I/II, less standardized than format string bugs
- Allows writing of arbitrary data to arbitrary addresses
- Documented in Phrack 57 / Undocumented for NT
- Hard/impossible to detect via stress testing
- Similarly hard to spot as Generation II

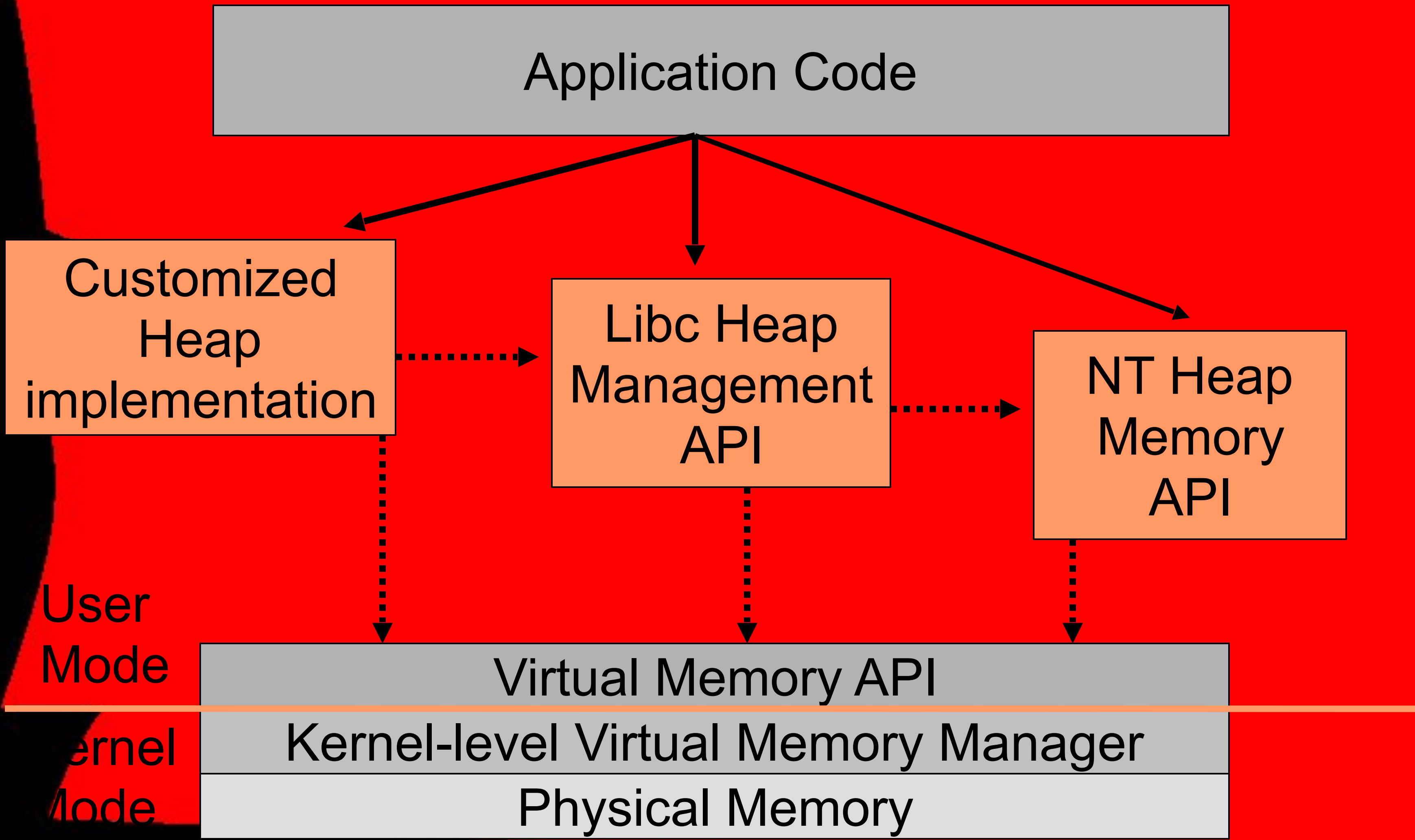
Heap Structure Exploit Generalities

Generalities on Heap Management

- Every libc/compiler has different algorithms, philosophies & internal structures for heap management (Vranhalia lists at least 8 different Kernel Memory allocators under *NIX)
- Customized optimization of heap management gives [illegible] performance leaps for applications, thus many large-scale applications have their own heap management algorithms
- Operating systems (such as WinNT2kXP) may provide their own heap management algorithms which the application might use

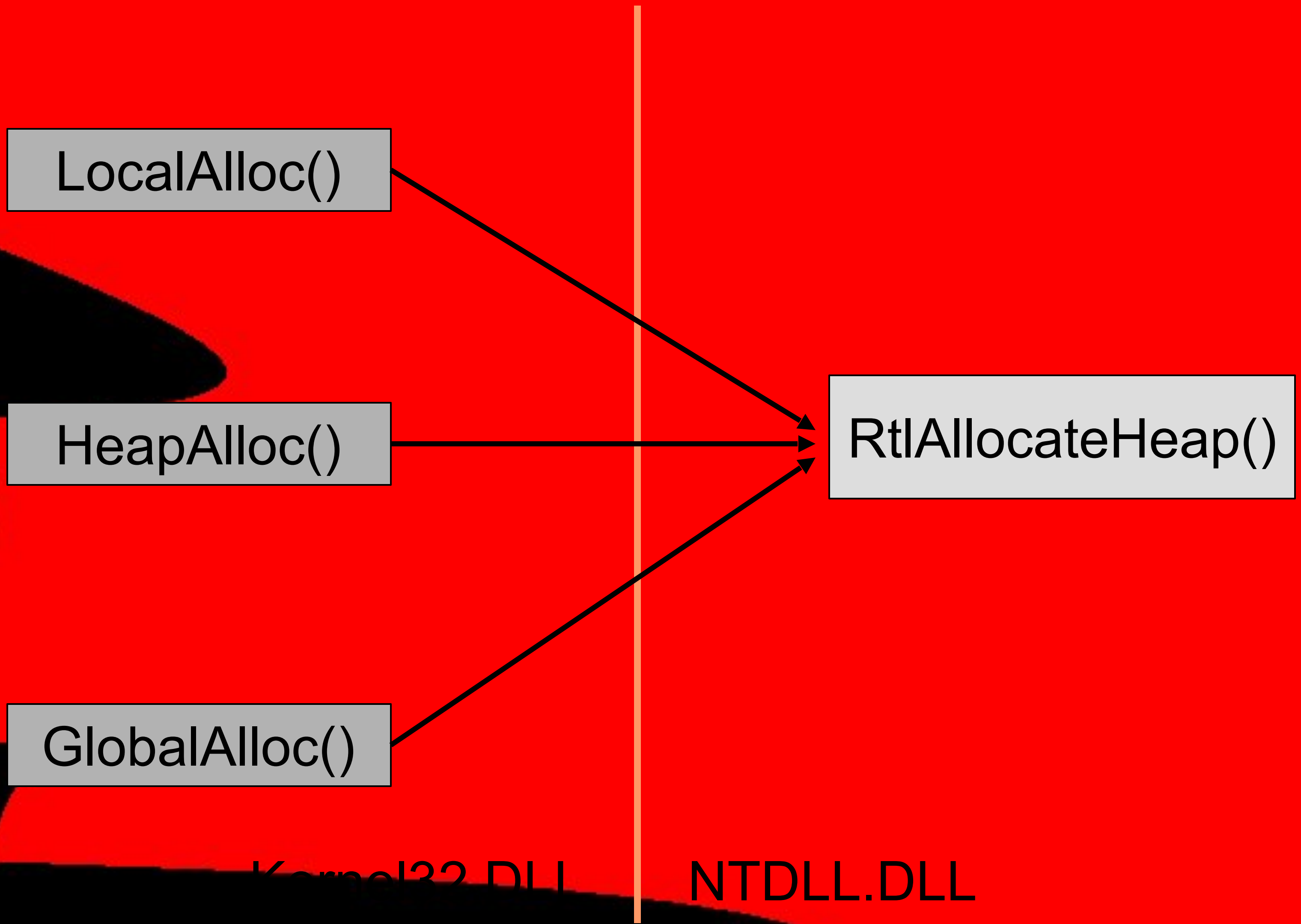
Heap Structure Exploit Generalities

Win32 heap management model

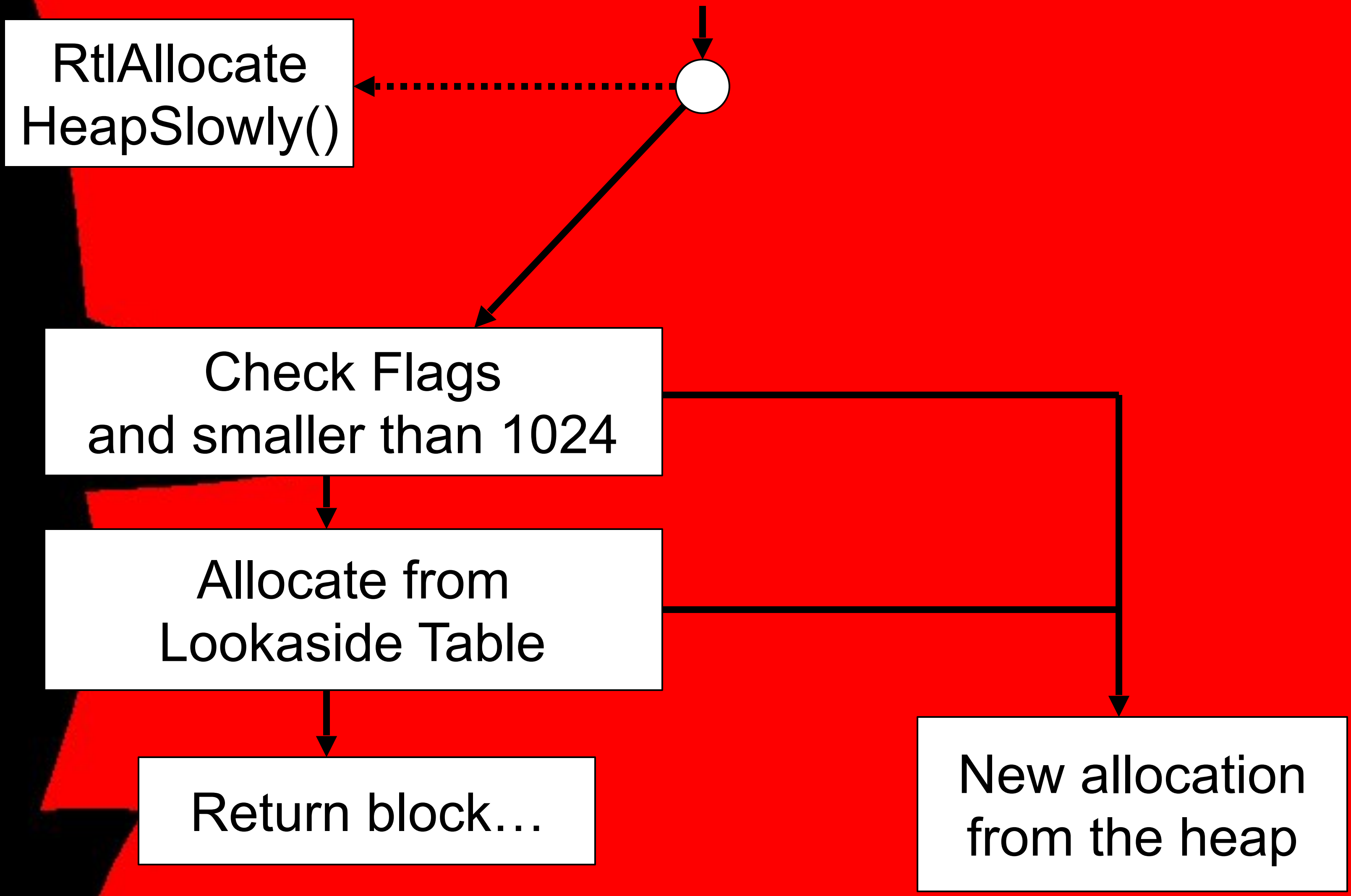


Heap Structure Exploits

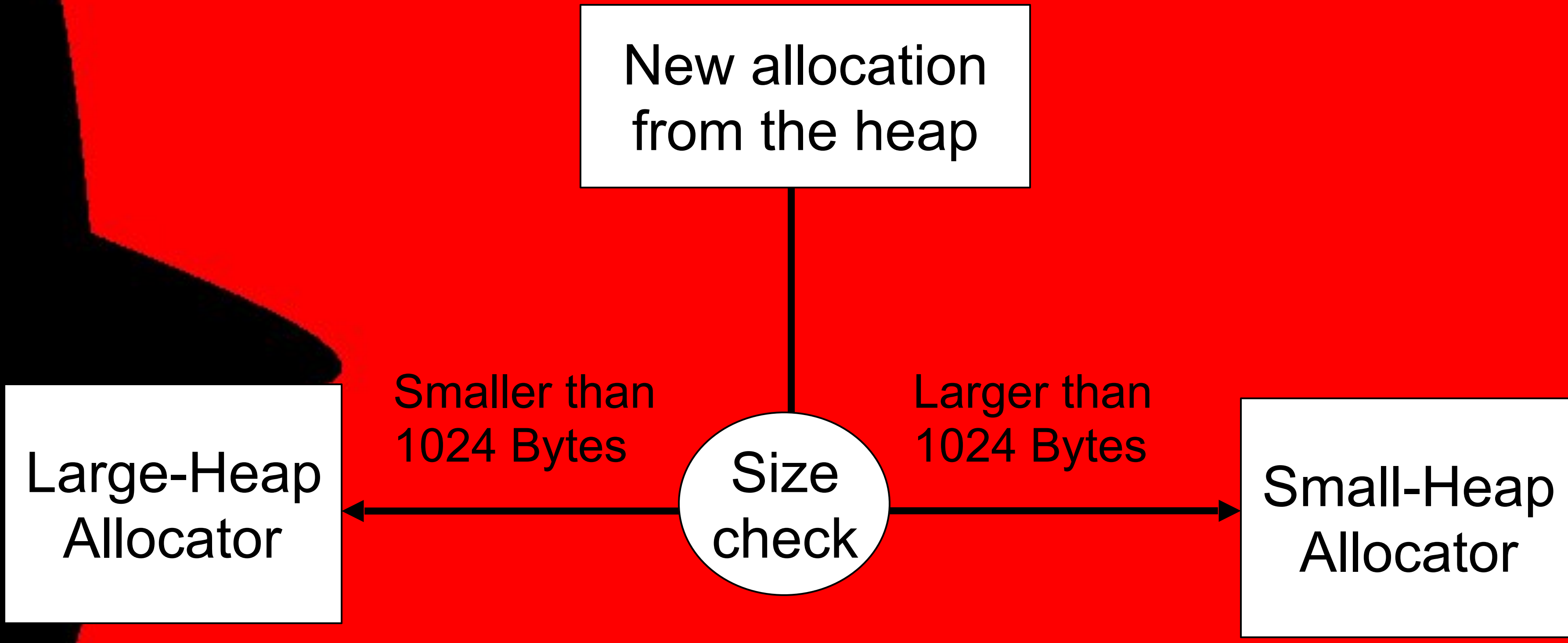
Win2k Heap Manager (I)



RtlAllocateHeap (I)



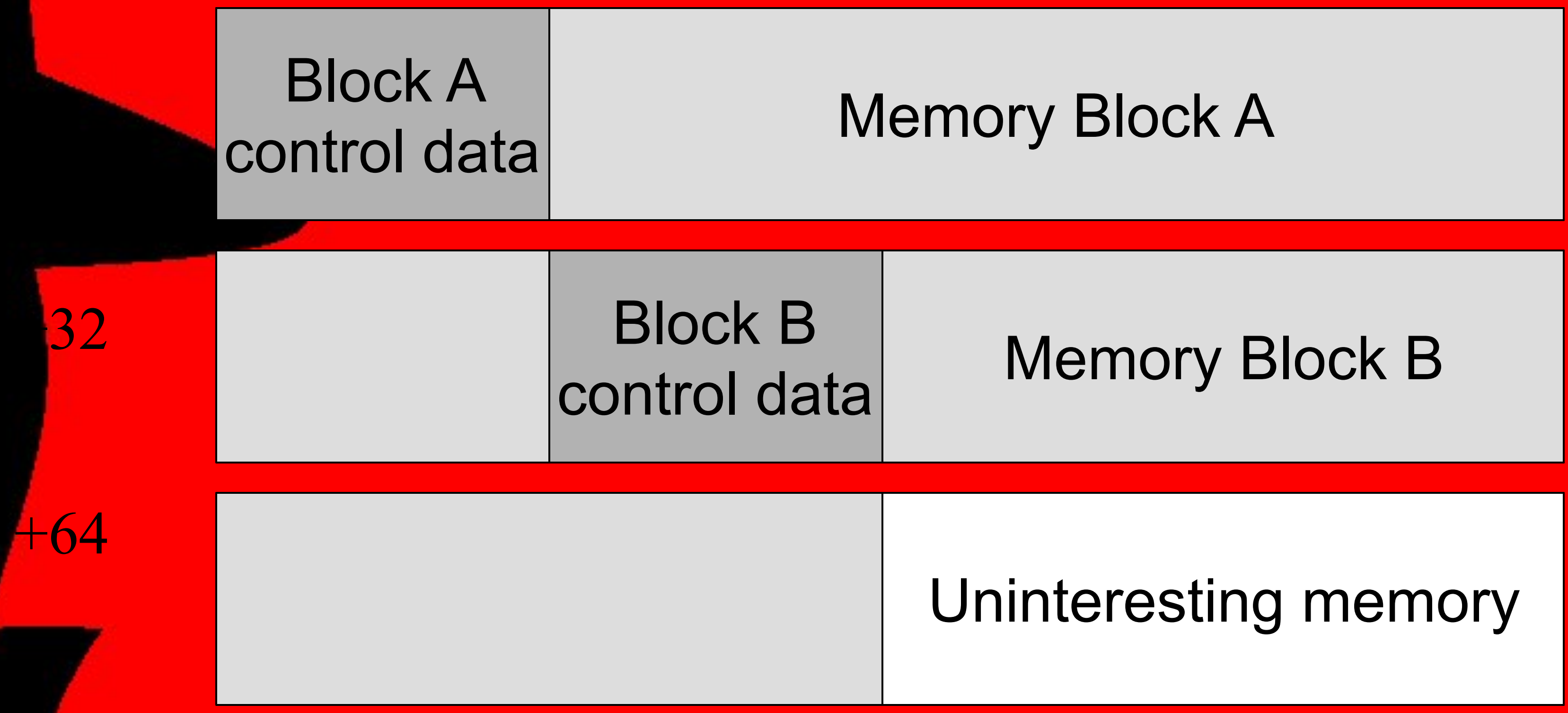
RtlAllocateHeap (II)



Heap Structure Exploits

Win2k Heap Manager (II)

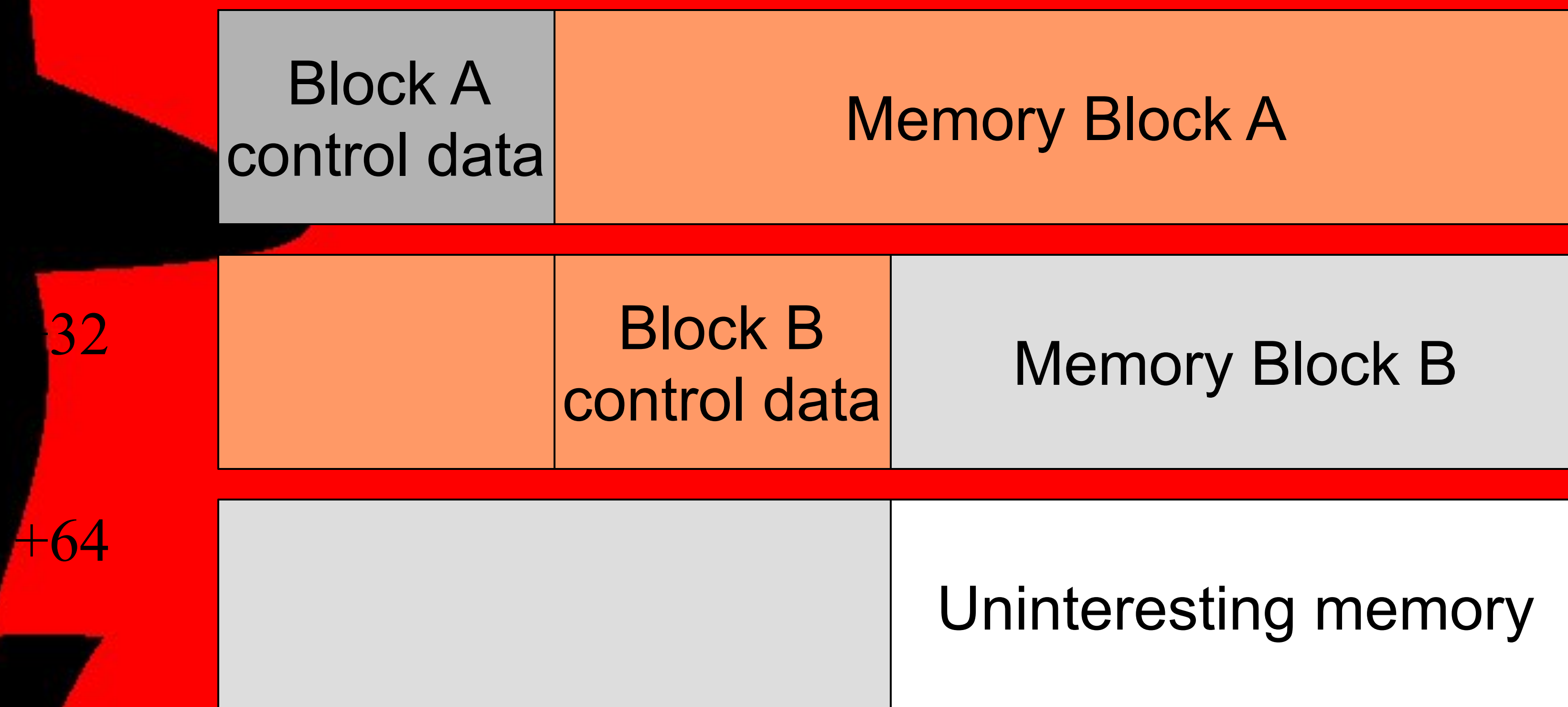
After two allocations of 32 bytes each our heap memory should look like this:



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:

	Size of this Block divided by 8		Size of the previous Block divided by 8
-4	Field_4	8 bit for Flags	

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

Heap Structure Exploits

Win2k Heap Manager (V)

Properties our block must have:

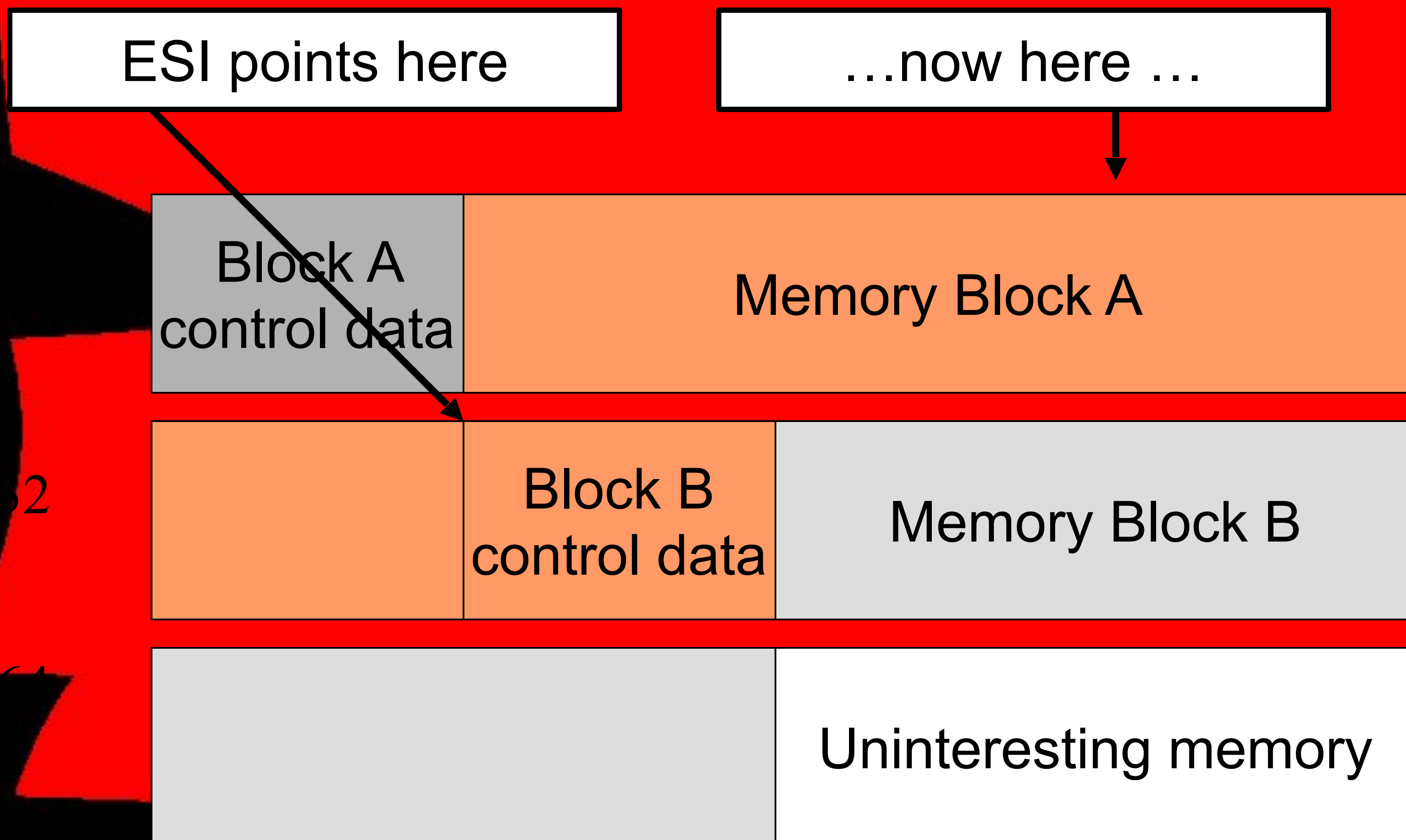
- Bit 0 of Flags must be set
- Bit 3 of Flags must be set
- [illegible]_4 must be smaller than 0x40
- The first field (own size) must be larger than 0x80

The block ‘XXXX99XX’ meets all requirements.
We reach the following code now:

Heap Structure Exploits

Win2k Heap Manager (VI)

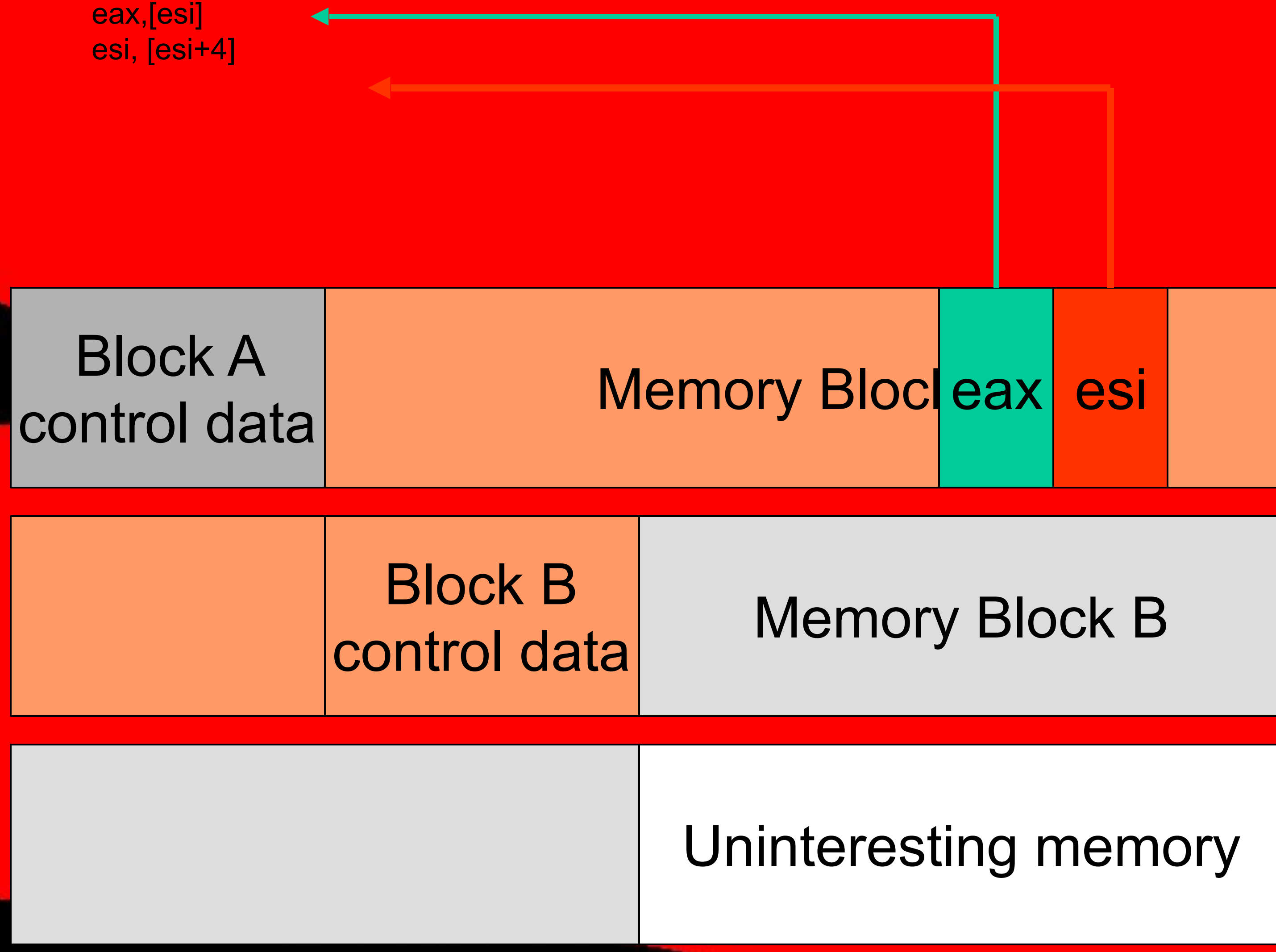
add esi, -24



Heap Structure Exploit

Win2k Heap Manager (VII)

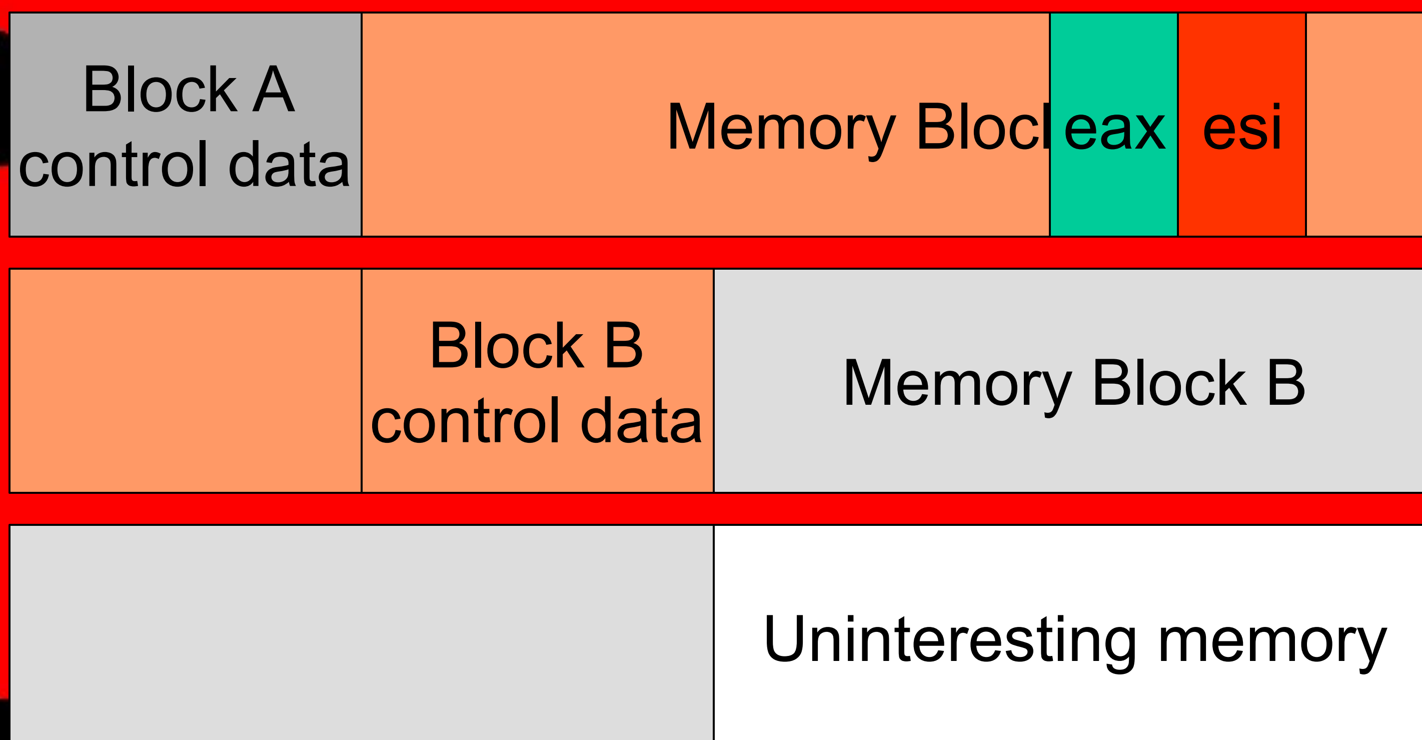
```
mov     eax,[esi]
mov     esi, [esi+4]
```



Heap Structure Exploits

Win2k Heap Manager (VIII)

mov [esi], eax ; Arbitrary memory overwrite



Heap Structure Exploits

Win2k Heap Manager (IX)

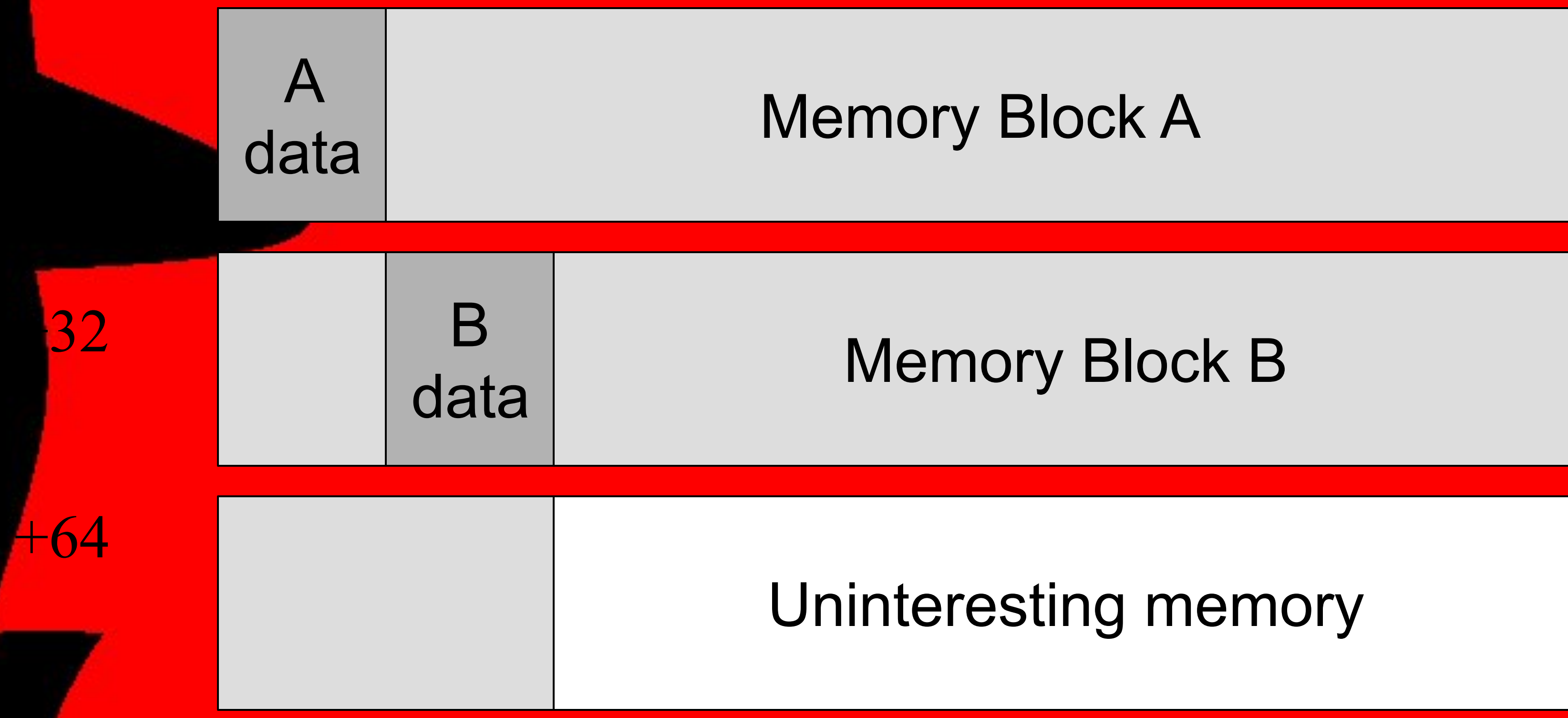
- If we can overwrite a complete control block (or at least 6 bytes of it) and have control over the data 24 bytes before that, we can easily write any value to any memory location.

- It should be noted that other ways of exploiting exist for smaller/different overruns – use your Disassembler and your imagination.

Heap Structure Exploits

Borland C++ run-time library (I)

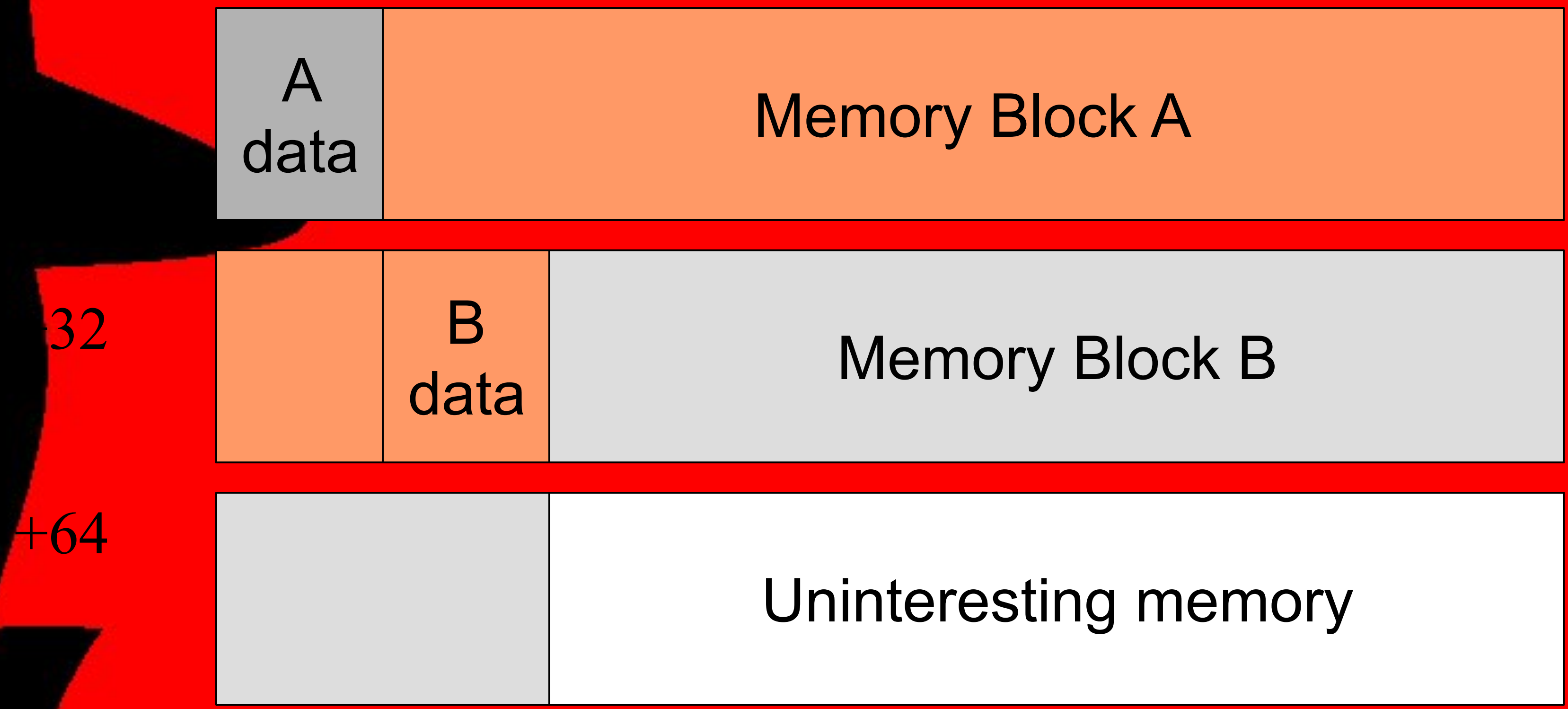
We have the same situation as before, but control blocks are 4 bytes in length only:



Heap Structure Exploits

Borland C++ run-time library (II)

The control structure is only one DWORD large.



Heap Structure Exploits

Borland C++ run-time library (III)

Control structure contains the size of the next allocated block

Libc checks: Is block smaller than 0x00100000 (ca. 1MB)

- If larger, page deallocator is called
- If smaller, small_free() – function is called

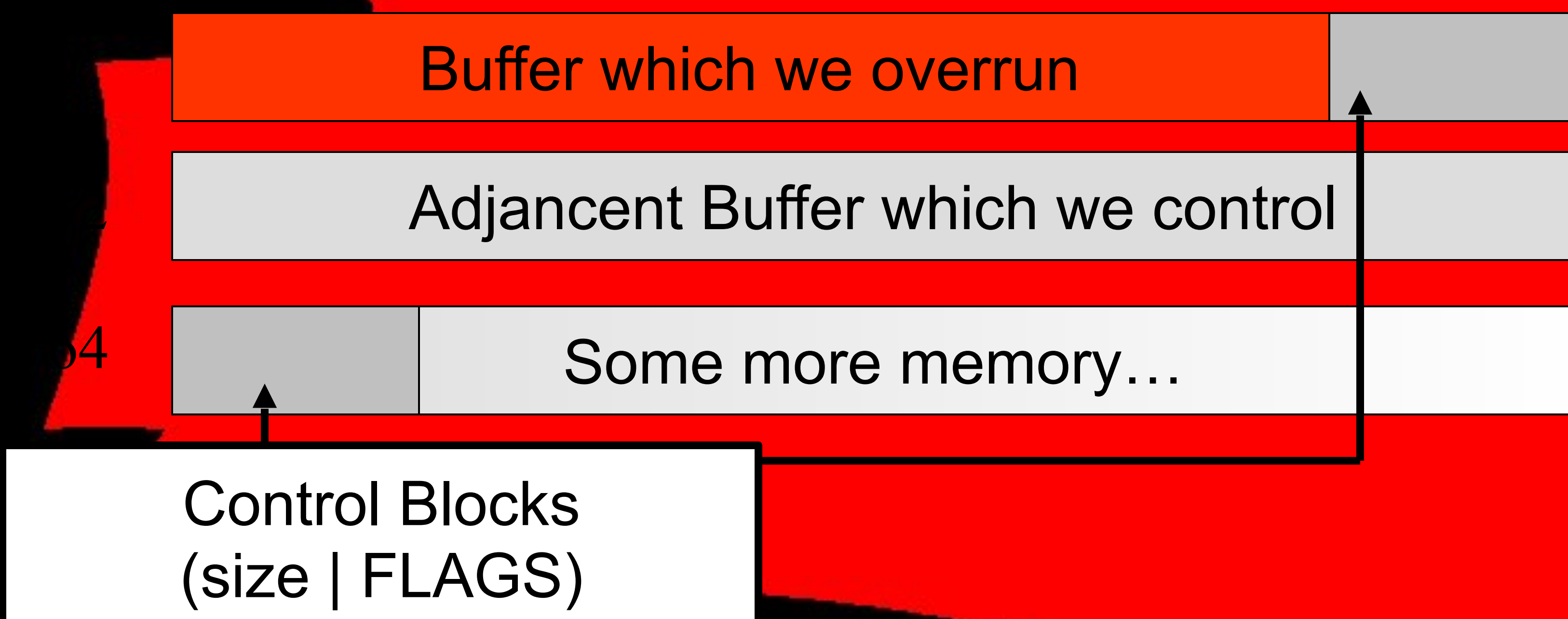
The dangerous code is in small_free()

e cannot overwrite the control block completely if we want do anything useful.

Heap Structure Exploits

Borland C++ off-by-one exploitation (I)

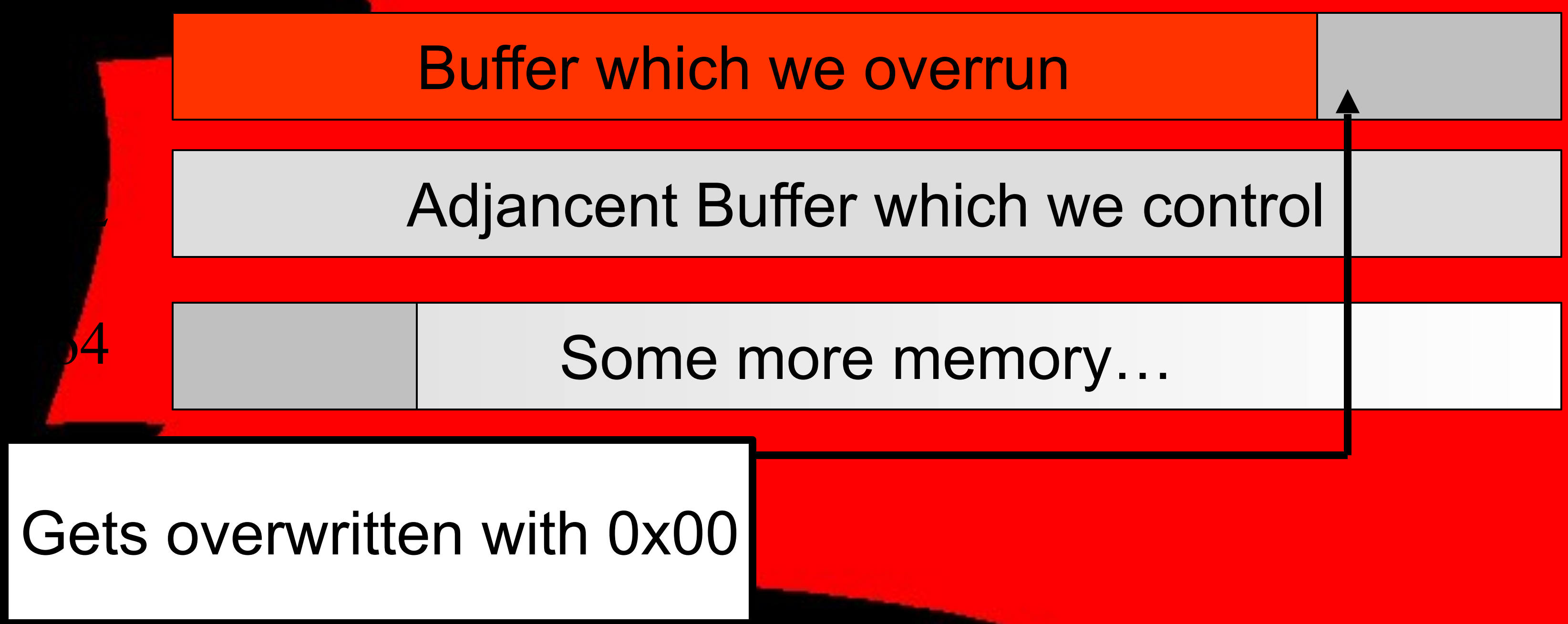
Assuming we overwrite the lowest byte of the control block of a 32-byte byte buffer which we control (which is **not** the one we overrun):



Heap Structure Exploits

Borland C++ off-by-one exploitation (II)

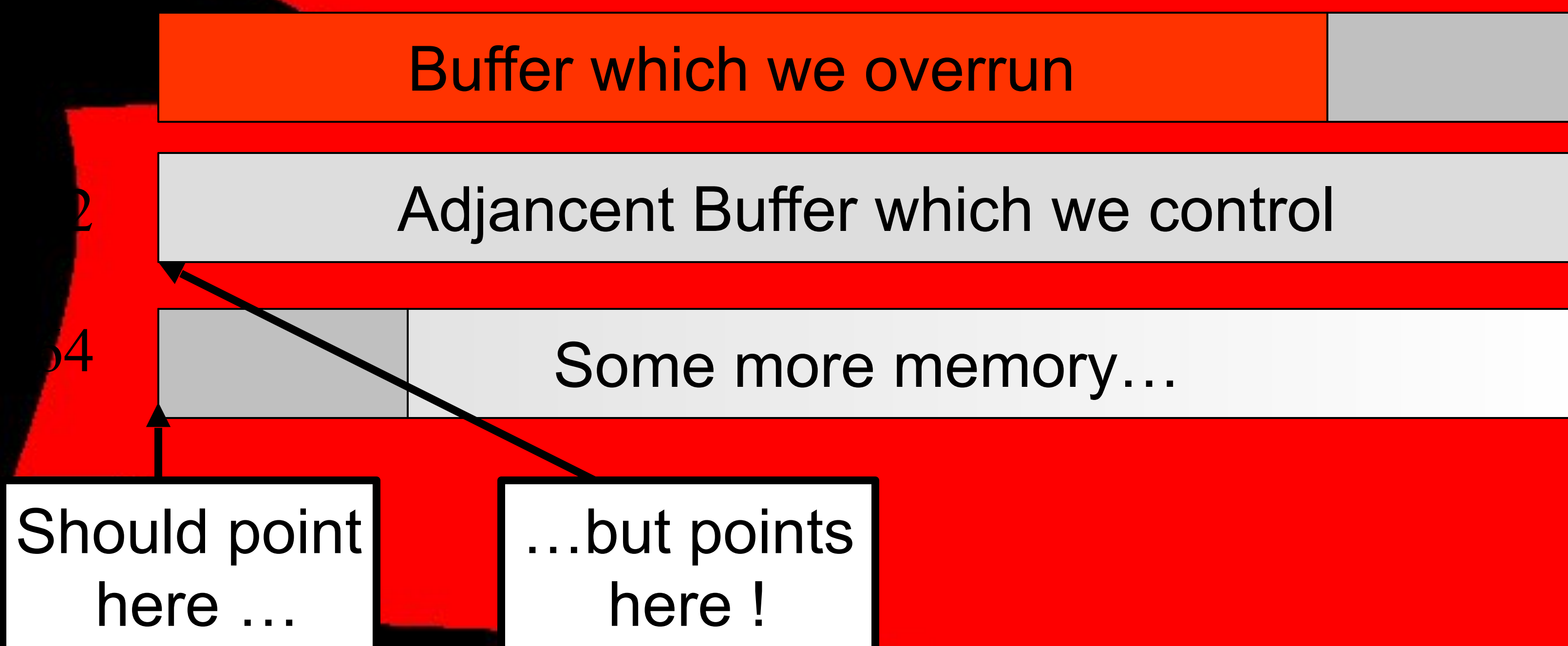
Instead of 0x20 OR’ed with the FLAGS, we get 0x00 due to the off-by-one NULL-byte.



Heap Structure Exploits

Borland C++ off-by-one exploitation (III)

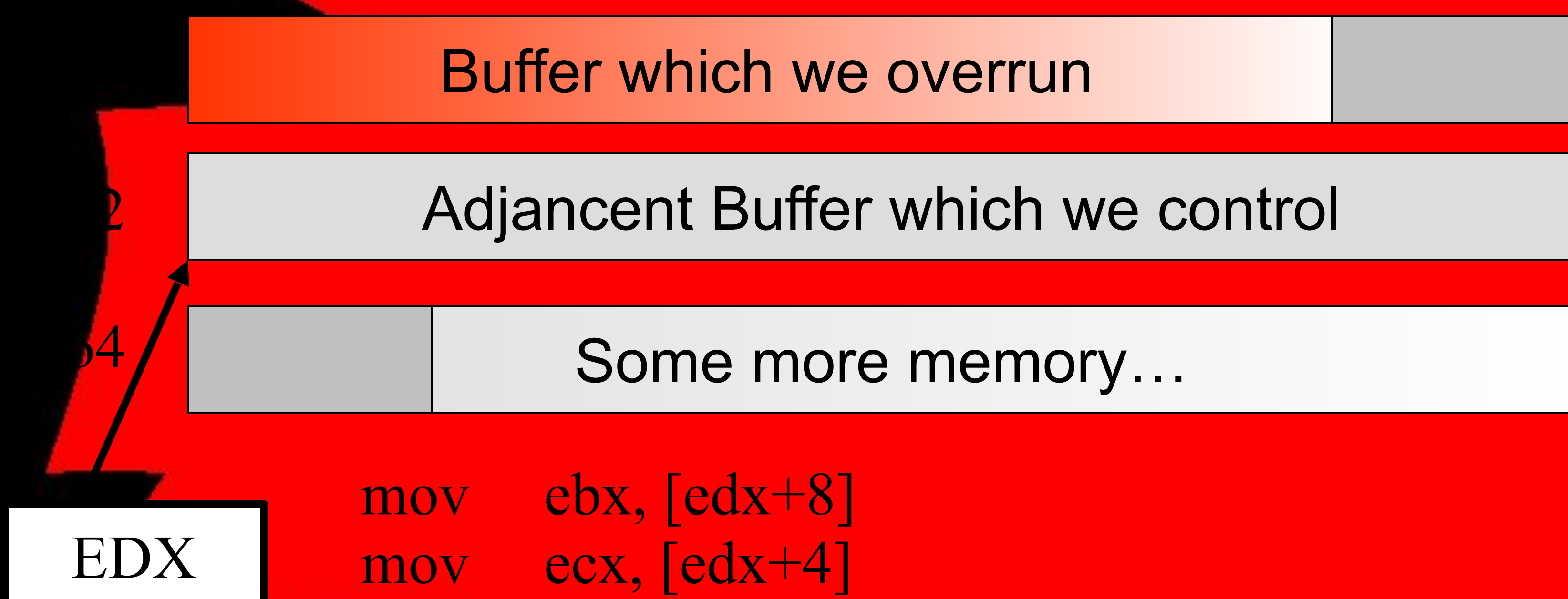
The libc tries to determine if the next buffer is a free buffer (to coalesce the two if so) – it attempts to skip the next block of memory by adding the value of the control structure. We modified this value, so it now points into the buffer we control.



Heap Structure Exploits

Borland C++ off-by-one exploitation (IV)

If we have bit 0 of the first byte of our trailing buffer set, the libc tries to coalesce the two “free” buffers using the code:




```
mov     ebx, [edx+8]
mov     ecx, [edx+4]
mov     [ecx+8], ebx     ; arbitrary memory overwrite
```

Heap Structure Exploits

Summary (I)

The only constant is change – especially in the world of bugs:

- Stack-based overflows are slowly “being hunted to near extinction”
- Biological Analogies can be seen: A particularly valuable and easy-to-hunt animal/bug has been hunted to near extinction (format string bugs)
- Some bug-hunters see bugs as a natural resource which is slowly being depleted – thus the ‘save the bugs movements’ and more push in the underground to keep bugs secret

Heap Structure Exploits

Summary (II)

New environments, new bugs…

- Majority of new code is C++/OOP/STL
- Pitfalls are not yet known – off-by-ones are possible, if not in strings, with other STL constructs
- New bugs are mostly heap overruns
- Due to their elusive nature, stress testing becomes useless: Goodbye Fuzz, Retina©, and 2 gazillion Perl-Scripts
- Reverse Engineers are at an advantage: They can document the inner workings of their compiler themselves
- Are you sure your JAVA runtime is working 100%ly correctly ?

Heap Structure Exploits

Summary (III)

Future of exploitation: Application Logic Corruption

- Traditional countermeasures attempt to prevent the execution of malicious code (StackGuard©, PaX)
- Non-executable data pages is a standard feature of new CPU architectures – goodbye shellcode
- New bug generation allows writing of arbitrary values to arbitrary addresses
- The attacker of the future will subvert the logic of the application by modifying it's variables – e.g. setting the *bool IsAuthenticated == TRUE*.
- Again, Reverse Engineers are useful – exploitation of closed-source applications without them is going to be hard to impossible

Heap Structure Exploits

Break

Any questions ?

Reliability

Exploitation Reliability (I)

Exploitation of buffer overruns under modern OS's

faces a bunch of difficulties:

- Variations in shared libraries & installs create incertainity concerning the right return address
- Multi-threading instead of forks create incertainity concerning the address of the stack
- Shooting down a web-server is not very stealthy
- Under NT (not 2k), services are not automatically restarted → one try and you're out

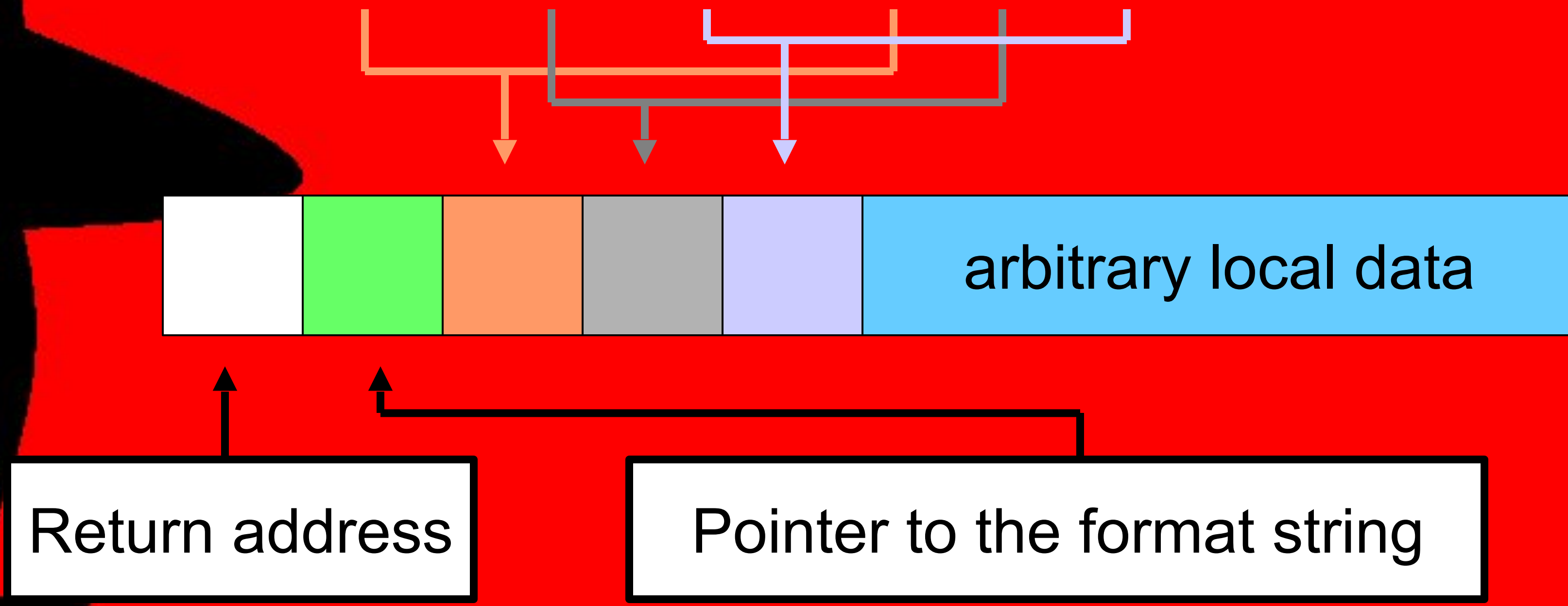
Methods are needed which improve reliability of exploitation !

Reliability

Format String Bugs (I)

Stack layout during regular printf()-call:

printf(“%lx---%s----%d”, v1, puf, var2);

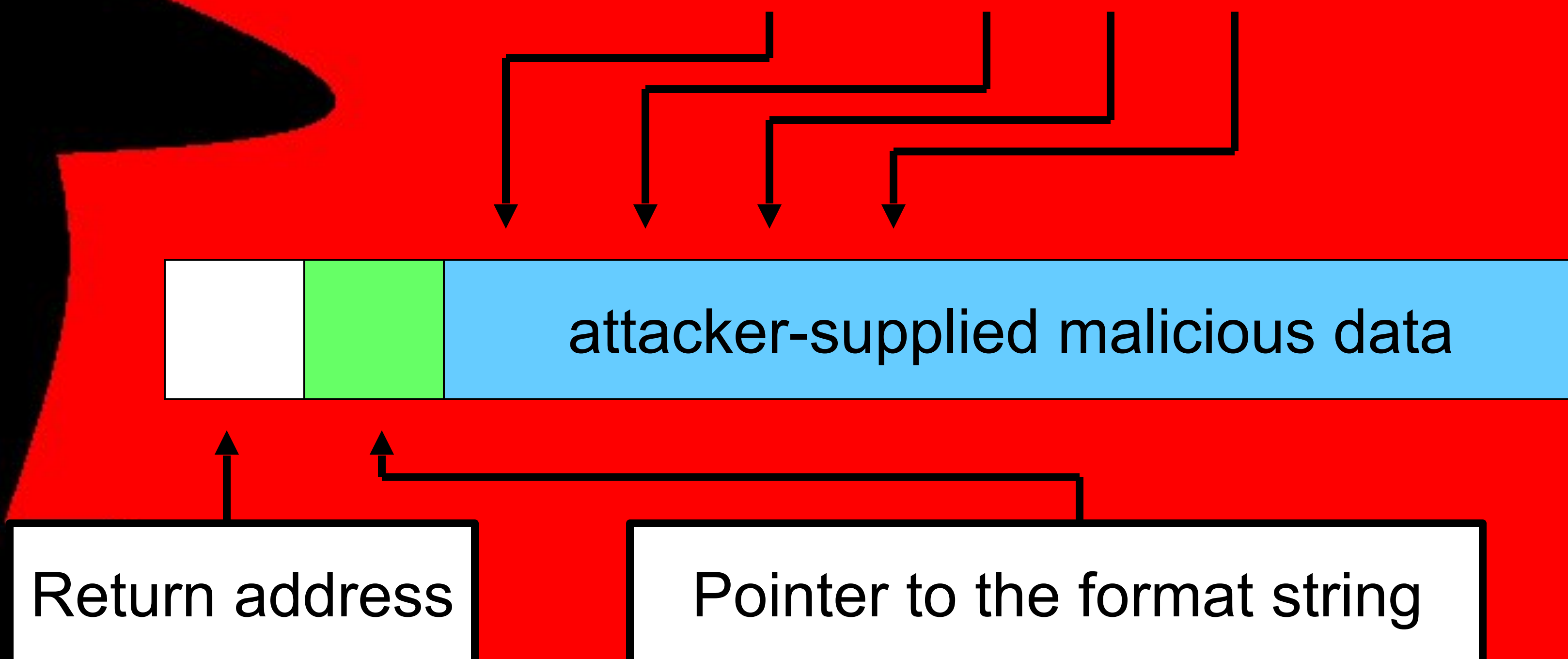


Reliability

Format String Bugs (II)

Stack layout during malicious printf()-call :

printf(stuff); // Stuff is set to contain
// “%.200lx%n%.40lx%n“



Reliability

Exploitation Reliability (II)

Windows NT/2k/ME provides a powerful feature which can be abused to increase reliability of exploitation:

Structured Exception Handling (SEH)

powerful features, this can be abused in various ways
– Two of them are:

1) Unhandled Exception Filter Attacks (UEFA)
2) Thread Exception Structure Overwrites (TESO)

Various other ways exists – where do you want to go today ?

Reliability

Exploitation Reliability (III)

Structured Exception Handling (SEH) allows an application to handle exceptions on it's own, similar to signal handlers under most UNIX variants.

of the key types of exception handlers are:

1) Final Exception Handlers installed through a function called *SetUnhandledExceptionFilter()*

2) Per-thread exception handlers installed by modifying a structure at fs:[0] and creating handler structures on the stack

Reliability

Exploitation Reliability (IV)

SetUnhandledExceptionFilter() installs a handler which will be called once all other handlers have failed, e.g. in a GPF or Page Fault (==UNIX SIG_SEGV)

A disassembly of the relevant function in KERNEL32.DLL looks like this:

```
mov    ecx, [esp+lpTopLevelExceptionFilter]
mov    eax, dword_77EE044C
mov    dword_77EE044C, ecx
retn   4
```

Reliability

Exploitation Reliability (V)

- Overwrite pointer at 0x7FEE044C with a pointer to our shellcode
- Trigger an exception → We seize control of the exception-handling thread

- Drawback: We need to know exact KERNEL32.DLL version, language (under NT) and loading address
- Advantage: We just need to write one DWORD and then trigger an exception

Reliability

Exploitation Reliability (VI)

A thread creates/installs a per-thread exception handler like this:

```
push        offset handler
push        dword fs:[0]
            fs:[0], esp
```

and creates a structure on the stack which looks like this:

+0	Pointer to next structure
+4	Pointer to handler code

Reliability

Exploitation Reliability (VII)

- fs:[0] forms a linked list of these structures
- The topmost handler gets called upon exception
- If it cannot handle the exception, control is passed to the next handler
- Repeat the above until no more exception handlers are left

- If we can overwrite the value at fs:[0] we can gain control !

Reliability

Exploitation Reliability (VIII)

- Cross-segment writing is impossible with string bugs and heap overwrites
- The structure starting at fs:[0] is called Thread Environment Block and is documented in both the NT [illegible] header files and by the Wine project
- Undocumented: The TEB's are created at highly predictable addresses
- By predicting these addresses and writing to the Thread Environment Block, we can hijack exception handlers

Reliability

Exploitation Reliability (IX)

Example of TEB allocation (identical on any NT2kXP):

1st Thread TEB: 0x7FFDE000
2nd Thread TEB: 0x7FFDD000
3rd Thread TEB: 0x7FFDE000

....

11th Thread TEB: 0x7FFD4000

12th Thread TEB: 0x7FFAF000
12+N-th Thread: 0x7FFAF000-N*0x1000

Reliability

Exploitation Reliability (X)

Example of TEB fragmentation:

Thread 1 is created
Thread 2 finishes & exits -- NONPAGED
Thread 3 is created
Thread 4 is created

Reliability

Exploitation Reliability (XI)

Example of TEB fragmentation:

Thread 1 is created
Thread 5 fills gap
Thread 3 is created
Thread 4 is created
Thread 6 is created
Thread 7 is created
Thread 8 is created
Thread 9 is created
Thread 10 is created

Reliability

Exploitation Reliability (XII)

Example of TEB fragmentation:

Thread 1 finishes & exits -- NONPAGED
Thread 5 fills gap
Thread 3 is created
Thread 4 finishes & exits -- NONPAGED
Thread 6 finishes & exits -- NONPAGED
Thread 7 is created
Thread 8 is created
Thread 9 finishes & exits -- NONPAGED
Thread 10 is created

Reliability

Exploitation Reliability (XIII)

We're facing some difficulties:

- We do not know which thread we're working with
- Thus we do not know where 'our' TEB is at
- The TEB-memory is fragmented due to constant dying/creating of threads in a production environment
- Thus we cannot overwrite them sequentially as odds are that we hit a page-fault before we get to our TEB

Reliability

Exploitation Reliability (XIV)

Strategy for exploitation:

- Create large number of threads
- Let lots of them die
- Create our exploiting thread
- Create a large number of additional threads to fill gaps
- Start overwriting TEBs sequentially

Results: 80-90% reliability independent of NT2kXP version, service pack or hotfix

Reliability

Any Questions ?