



Control Hijacking

Basic Control Hijacking Attacks

程序流劫持

Control hijacking attacks

- Attacker's goal:
 - Take over target machine (e.g. web server)
 - Execute arbitrary code on target by hijacking application control flow

常用解决方法 = DEP.

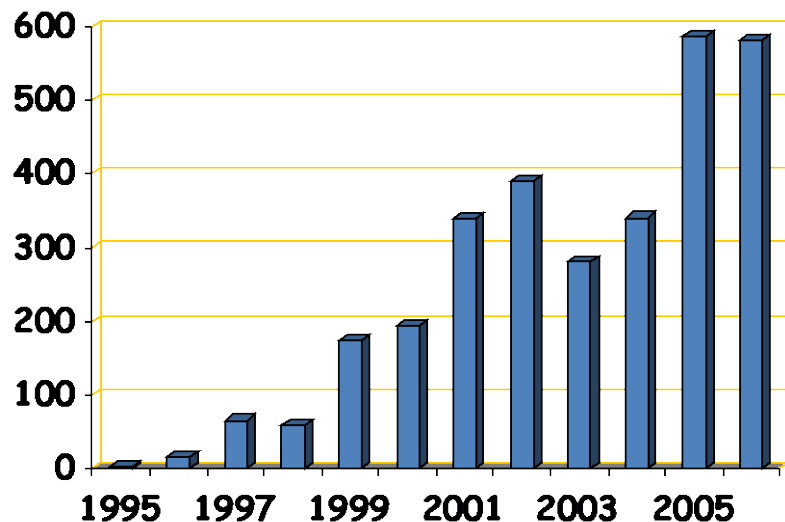
- Examples.

ASLR

- Buffer overflow attacks
- Integer overflow attacks
- Format string vulnerabilities 格式化字符串攻击

Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
 - First major exploit: 1988 Internet Worm. fingerd.



≈20% of all vuln.

2005-2007: ≈ 10%

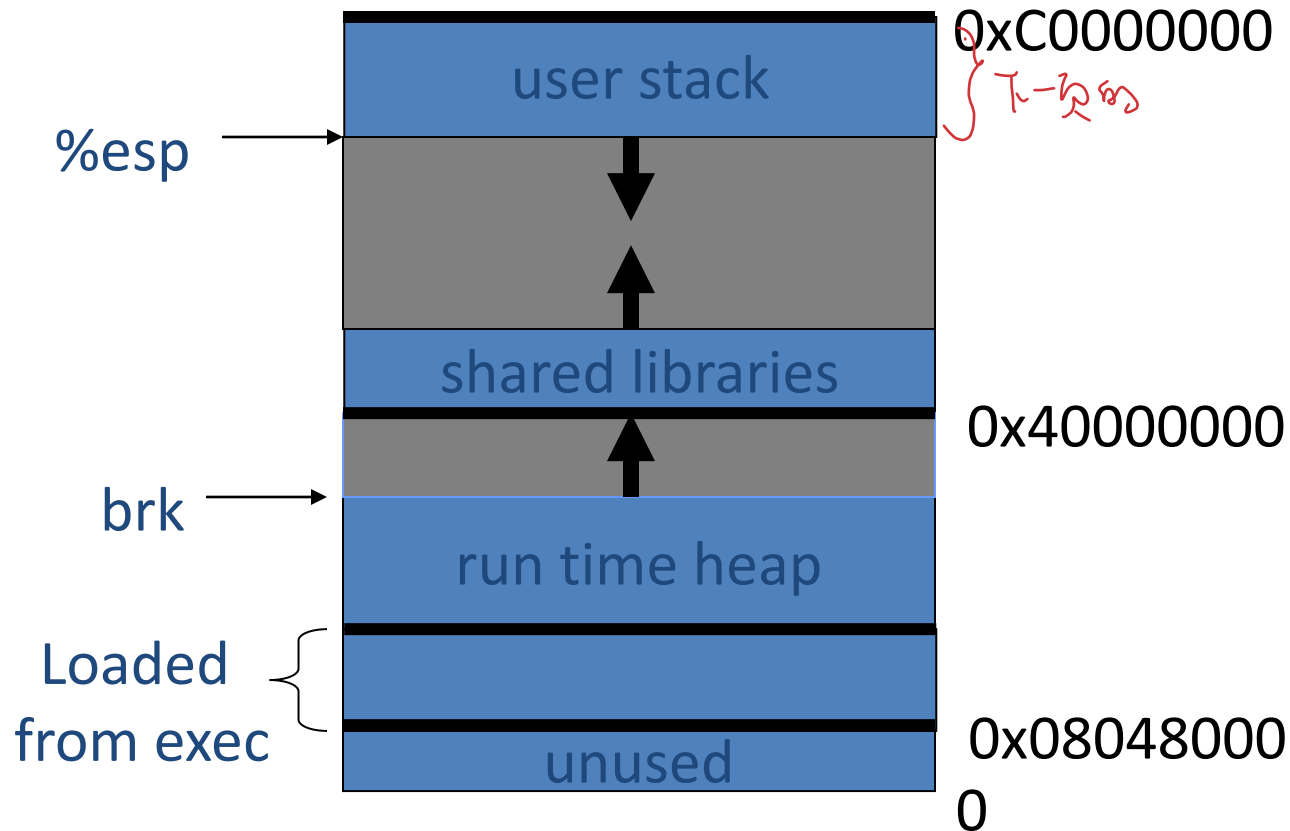
Source: NVD/CVE

Inline 函数不调用stack

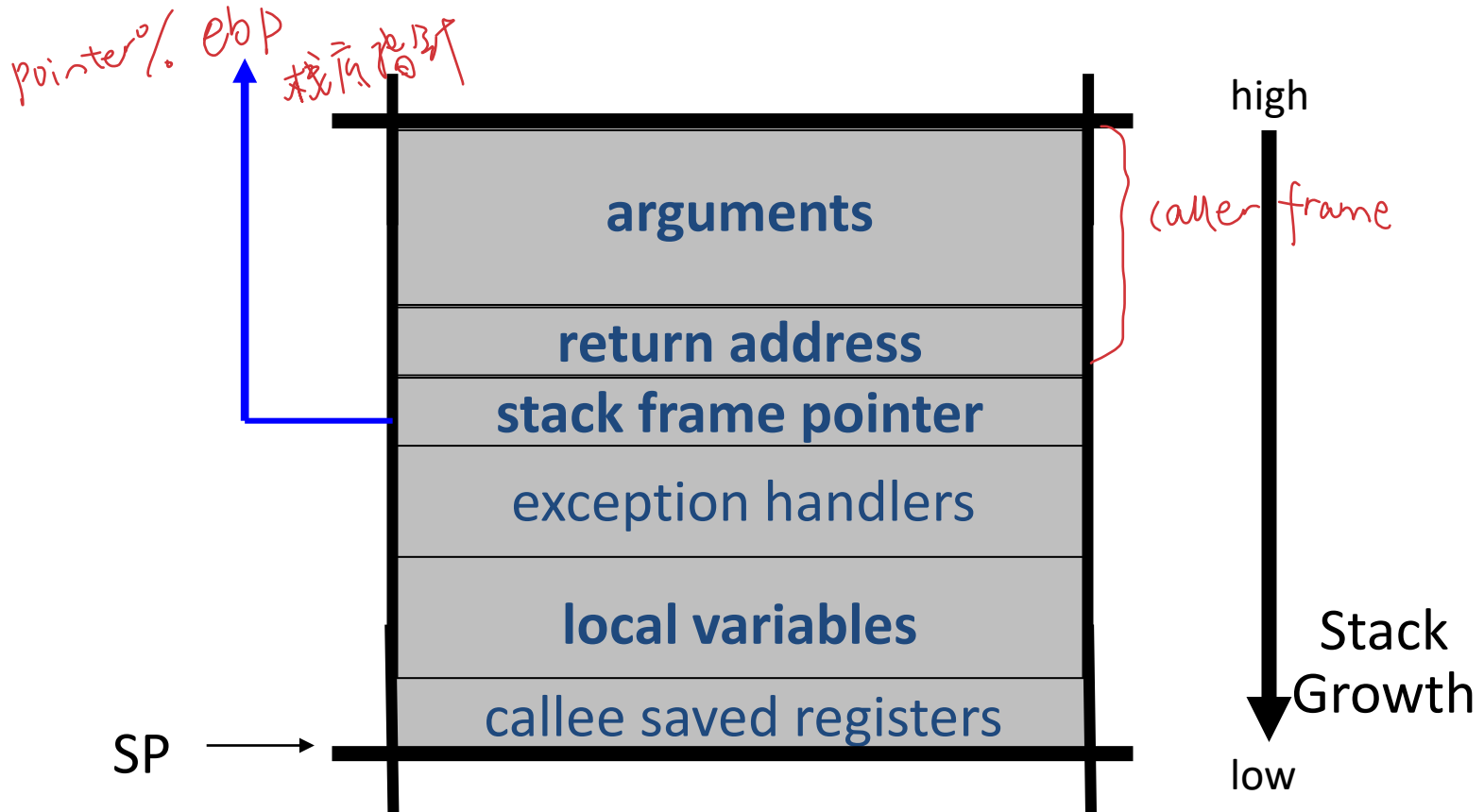
What is needed

- Understanding C functions, the stack, and the heap.
 - Know how system calls are made
 - The `exec()` system call
-
- Attacker needs to know which CPU and OS used on the target machine:
 - Our examples are for x86 running Linux or Windows
 - Details vary slightly between CPUs and OSs:
 - Little endian vs. big endian (x86 vs. Motorola)
 - Stack Frame structure (Unix vs. Windows)

Linux process memory layout



Stack Frame



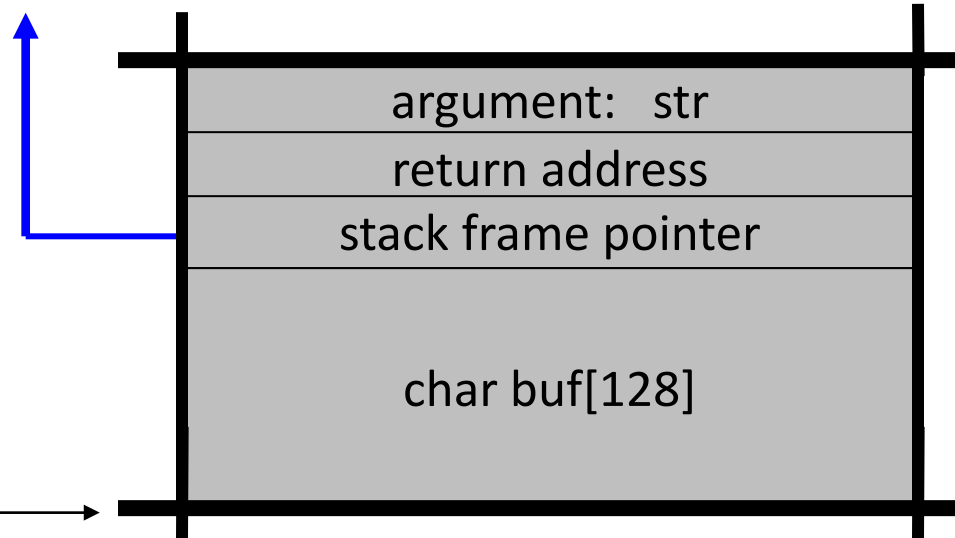
Attacker Model

- Critical to have a precise and detailed definition of the capabilities of an attacker
 - Has access to the code of your system (reason is that they can simply buy it from the vendor)
 - Your system is connected to the network (remote attacks)
 - Attacker has unlimited resources to analyze your code and find weaknesses. Specifically, they have access to program analysis tools, highly optimized SAT/SMT solvers, and deep understanding of your system
 - Attacker does NOT have access to real-time generated random numbers and keys (assuming no insider attack or some other ways in which keys can be leaked, e.g., side-channel attacks)

What are buffer overflows?

Suppose a web server contains a function:

When func() is called stack looks like:

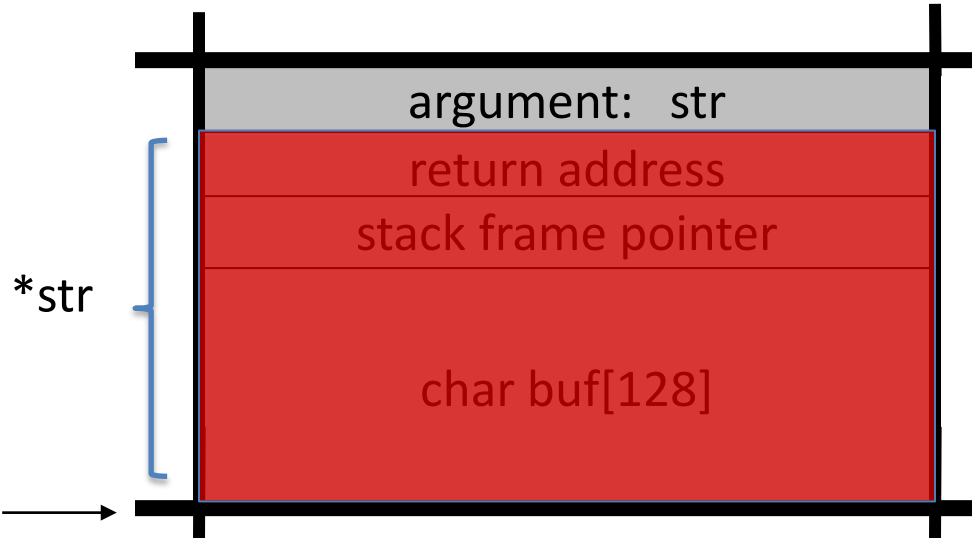


```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    do-something(buf);  
}
```


What are buffer overflows?

What if `*str` is 136 bytes long?

After `strcpy`:



```
void func(char *str) {  
    char buf[128];  
  
    strcpy(buf, str);  
    do-something(buf);  
}
```

Problem:

no length checking in `strcpy()`

Basic stack exploit

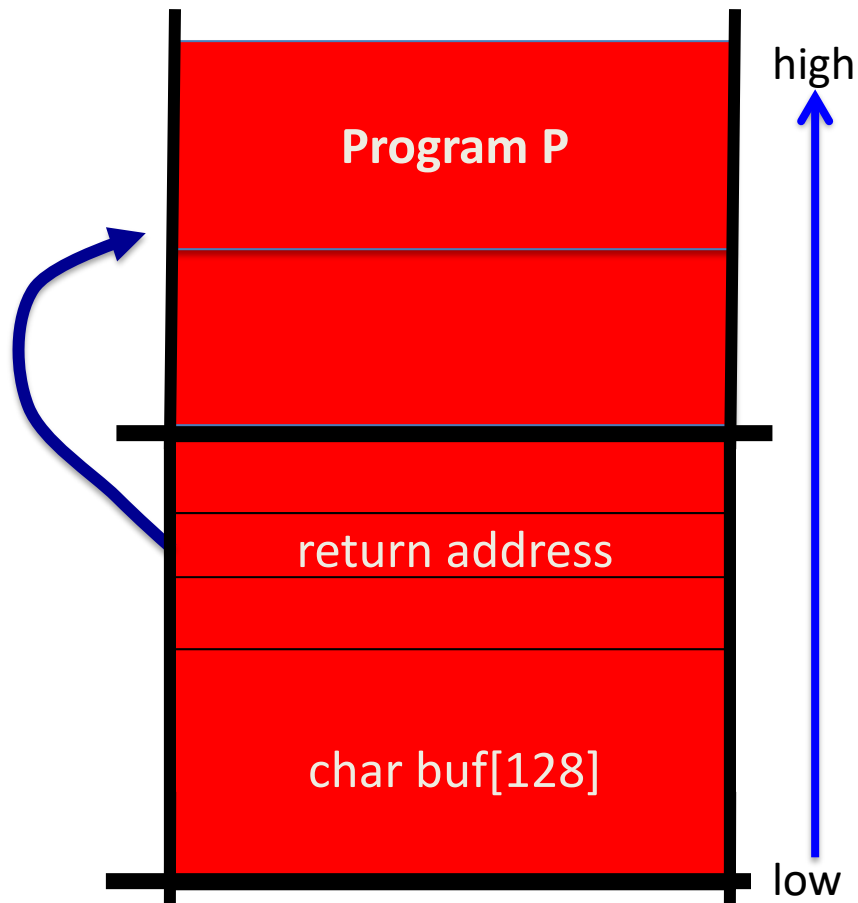
Suppose `*str` is such that
after `strcpy` stack looks like:

Program P: `exec("/bin/sh")`
(exact shell code by Aleph One)
此时stack中全部被覆盖掉了

可以重定向到return addr 执行自己的代码

When `func()` exits, the user gets shell !

Note: attack code P runs *in stack*.

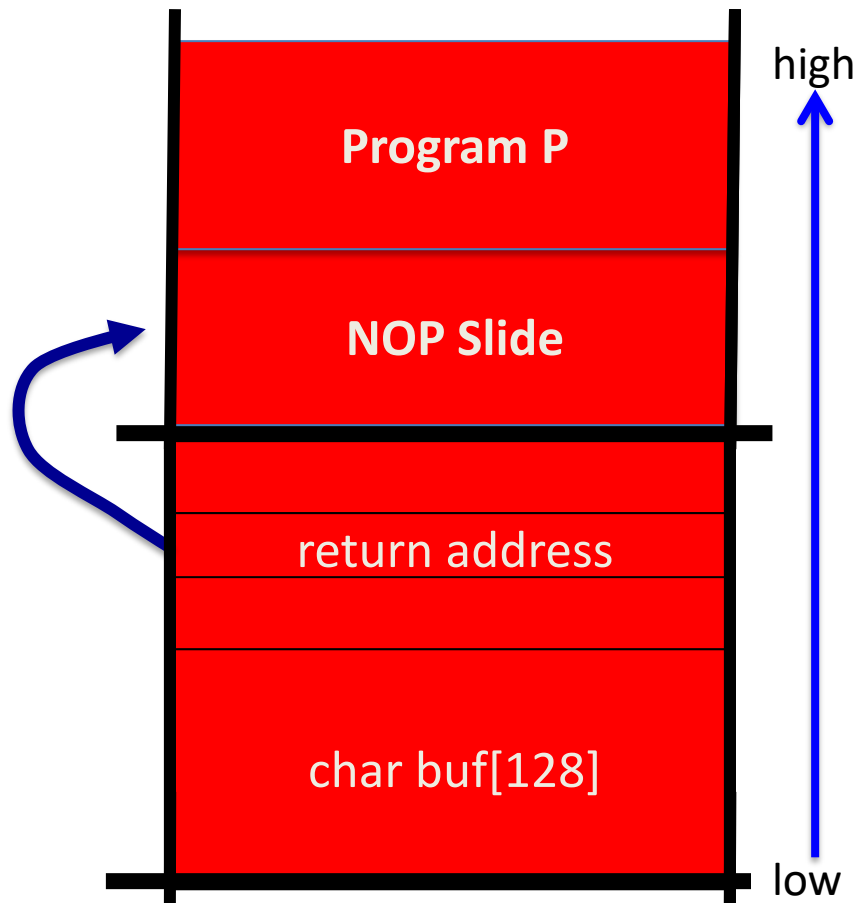


The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide 无操作指令

- Guess approximate stack state when `func()` is called 滑到下一组可执行操作
- Insert many NOPs before program P:
`nop , xor eax,eax , inc ax`





Details and examples

- Some complications:
 - Program P should not contain the '\0' character.
 - Overflow should not crash program before func() exists.
- (in)Famous remote stack smashing overflows:
 - (2007) Overflow in Windows animated cursors (ANI). `LoadAniIcon()`
 - (2005) Overflow in Symantec Virus Detection
`test.GetPrivateProfileString "file", [long string]`

Many unsafe libc functions

`strcpy` (char *dest, const char *src)

`strcat` (char *dest, const char *src)

`gets` (char *s)

`scanf` (const char *format, ...) and many more.

- “Safe” libc versions `strncpy()`, `strncat()` are misleading
 - e.g. `strncpy()` may leave string unterminated.
 - Windows C run time (CRT):
 - `strcpy_s (*dest, DestSize, *src)`: ensures proper termination
-

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks)
 - Overwrite the address of an exception handler in stack frame.
- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)



- Overflowing buf will override function pointer.
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
 - Overflowing buf next to pos overrides value of pos.

Virtual functions in object-oriented languages

```
Class c {
```

```
    private:
```

```
    protected:
```

```
        foo();
```

```
}
```

```
Class T : public class C{
```

```
    foo() { } //this is not the same as the parent class method foo()
```

```
}
```

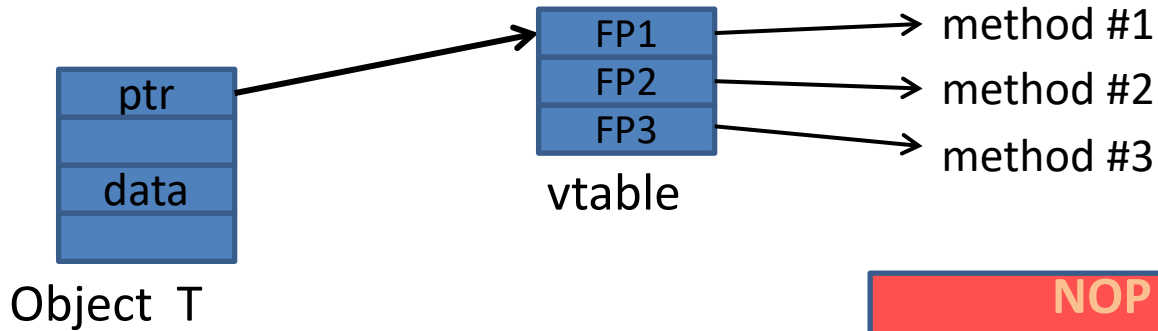
```
Class C * c;
```

```
c = new class T();
```

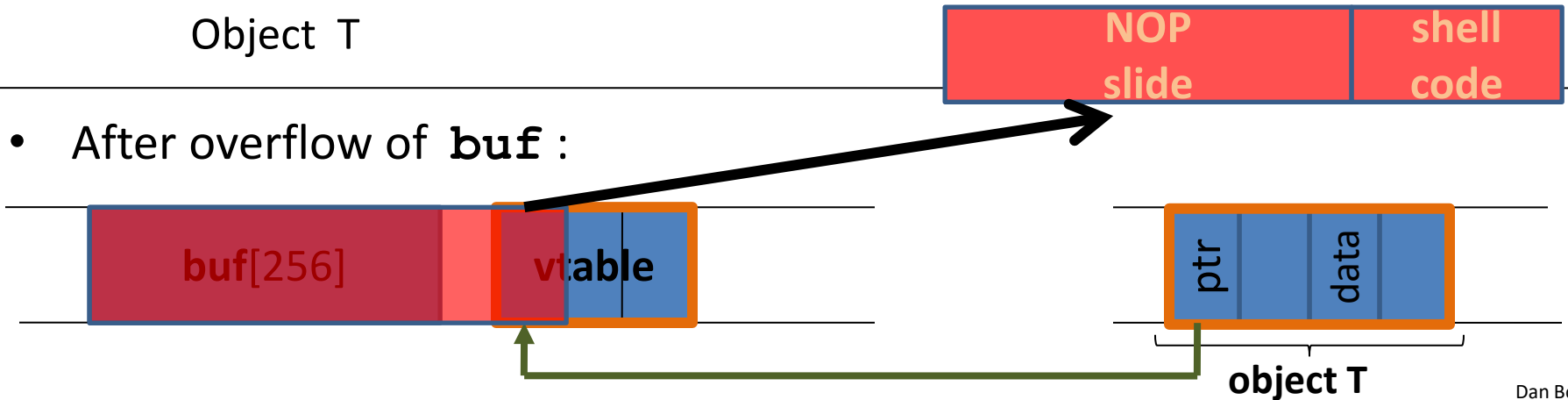
```
c -> foo(); //virtual functions
```

Corrupting method pointers

- Compiler generated function pointers (e.g. C++ code)



- After overflow of **buf** :



Finding buffer overflows

- To find overflow:
 - Run web server on local machine
 - Issue malformed requests (ending with “\$\$\$\$\$”)
 - Many automated tools exist (called fuzzers – next module)
 - If web server crashes,
 - search core dump for “\$\$\$\$\$” to find overflow location
- Construct exploit (not easy given latest defenses)



Control Hijacking

More Control Hijacking Attacks

More Hijacking Opportunities

- **Integer overflows:** (e.g. MS DirectX MIDI Lib)
- **Double free:** double free space on heap.
 - Can cause memory mgr to write data to specific location
 - Examples: CVS server
- **Format string vulnerabilities**

What is an integer overflow?

Let `Int_1` denote a variable of type 1-bit bit-vector

```
Int_1 x, y;
```

```
x = 1;
```

```
y = x+x;
```

$1+1 = 0$ in single-bit bit-vector arithmetic

CVE database of vulnerabilities and exploits:
cve.mitre.org

Integer Overflows (see Phrack 60)

Problem: what happens when int exceeds max value?

int m; (32 bits)

short s; (16 bits)

char c; (8 bits)

$$c = 0x80 + 0x80 = 128 + 128$$

$$\Rightarrow c = 0$$

$$s = 0xff80 + 0x80$$

$$\Rightarrow s = 0$$

$$m = 0xffffffff80 + 0x80$$

$$\Rightarrow m = 0$$

Can this be exploited?

An example

```
void func( char *buf1, *buf2,  unsigned int len1, len2) {  
    char temp[256];  
    if (len1 + len2 > 256) {return -1}           // length check  
    memcpy(temp, buf1, len1);                     // cat buffers  
    memcpy(temp+len1, buf2, len2);  
    do-something(temp);                           // do stuff  
}
```

What if **len1 = 0x80**, **len2 = 0xffffffff80** ?

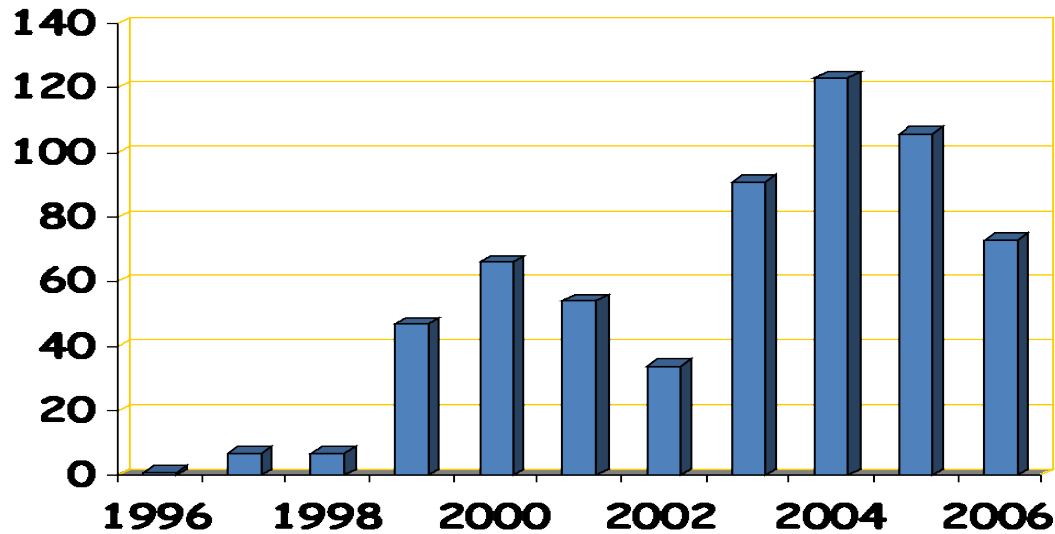
⇒ len1+len2 = 0

Second memcpy() will overflow the temp buffer !!

其实还是溢出, 但由于 integer 溢出结果=0,

memcpy 上出错
buffer overflow

Integer overflow exploit stats



Source: NVD/CVE



Control Hijacking

Platform Defenses

Preventing hijacking attacks

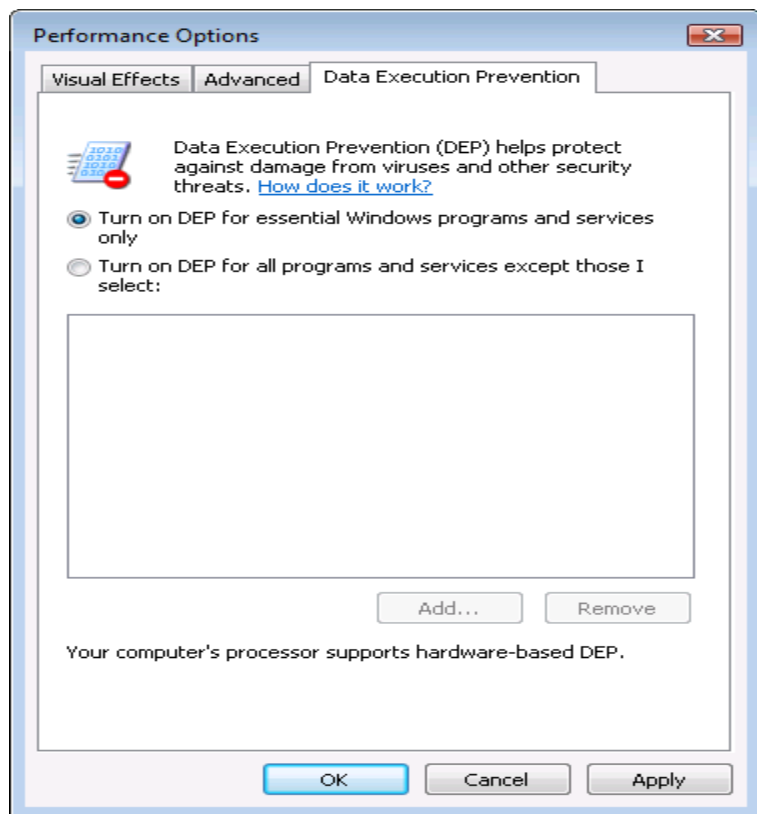
1. Fix bugs:
 - Audit software
 - Automated tools: Coverity, Prefast/Prefix.
 - Rewrite software in a type safe language (Java, ML)
 - Difficult for existing (legacy) code ...
2. Concede overflow, but prevent code execution
3. Add runtime code to detect overflows exploits
 - Halt process when overflow exploit detected
 - StackGuard, LibSafe, ...

Marking memory as non-execute (W^X)

Prevent attack code execution by marking stack and heap as **non-executable**

- NX-bit on AMD Athlon 64, XD-bit on Intel P4 Prescott
 - NX bit in every Page Table Entry (PTE)
- Deployment:
 - Linux (via PaX project); OpenBSD
 - Windows: since XP SP2 (DEP)
 - Visual Studio: **/NXCompat[:NO]**
- Limitations:
 - Some apps need executable heap (e.g. JITs).
 - Does not defend against **Return Oriented Programming** exploits

Examples: DEP controls in Windows



DEP terminating a program

Linking libraries

At compile time, the compiler has to check and make sure that the code of the library function is available.

- 1) Static linking: library is linked into your binary. *.a. The problem with static linking code bloat.
- 2) Dynamic linking: The linker is aware that the library is stored in a common place in memory. During linking, it will replace library functions calls with jump instructions whose addresses refer to the locations of the corresponding in the stored library.
(* .so) (* .DLL)

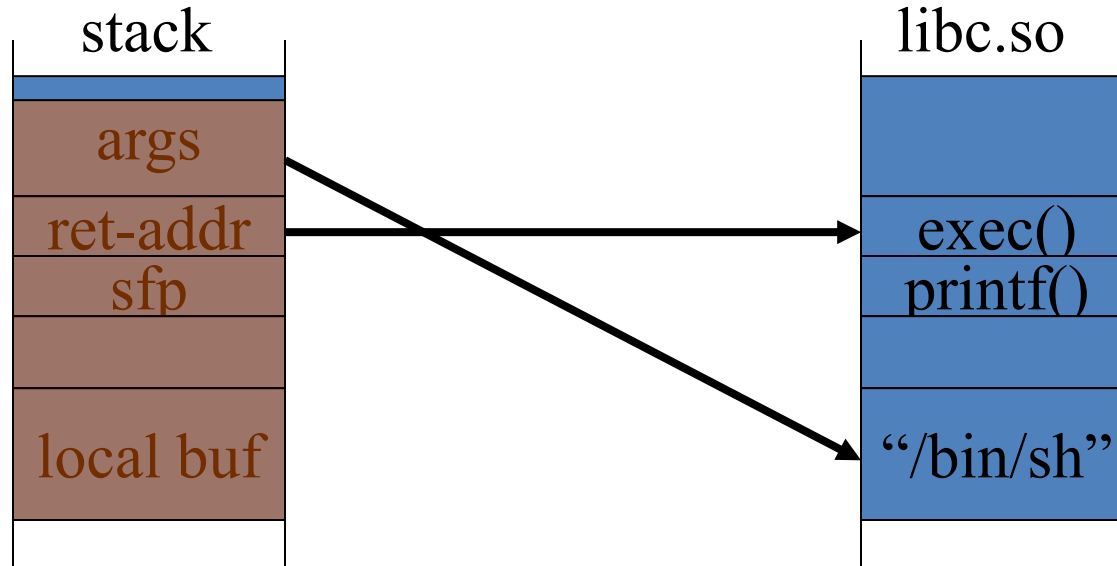
ROP

扫描dll库和exe文件, 提取可利用指令片段gadget
gadget 以 ret 指令结尾. ROP攻击以 ret 结尾的gadget
操作栈相关寄存器. 控制 process

- Return-oriented programming is an approach to attack a system using its own code (without necessarily injected external code into the memory of the running process)
- String these “gadgets” together such that you now have an attack

Attack: Return Oriented Programming (ROP)

- Control hijacking without executing code





Response: randomization

- **ASLR**: (Address Space Layout Randomization)
 - Map shared libraries to rand location in process memory
⇒ Attacker cannot jump directly to exec function
 - **Deployment**: (/DynamicBase)
 - **Windows Vista**: 8 bits of randomness for DLLs
 - aligned to 64K page in a 16MB region ⇒ 256 choices
 - **Windows 8**: 24 bits of randomness on 64-bit processors
- **Other randomization methods**:
 - Sys-call randomization: randomize sys-call id's
 - Instruction Set Randomization (ISR)

ASLR Example

Booting 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

Note: everything in process memory must be randomized
stack, heap, shared libs, image

- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

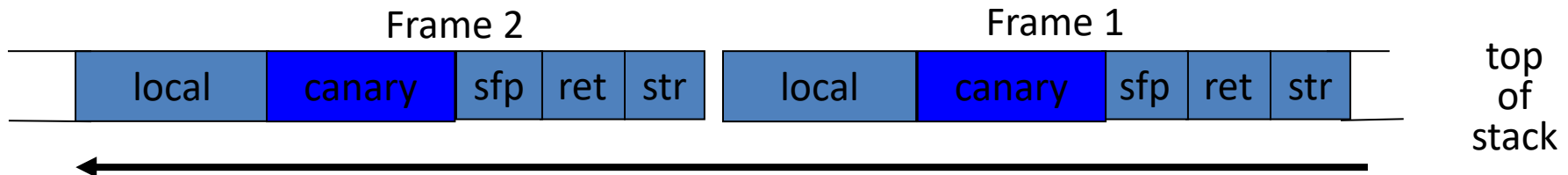


Control Hijacking

Run-time Defenses

Run time checking: StackGuard

- Many run-time checking techniques ...
 - we only discuss methods relevant to overflow protection
- Solution 1: StackGuard
 - Run time tests for stack integrity.
 - Embed “canaries” in stack frames and verify their integrity prior to function return.



Canary Types

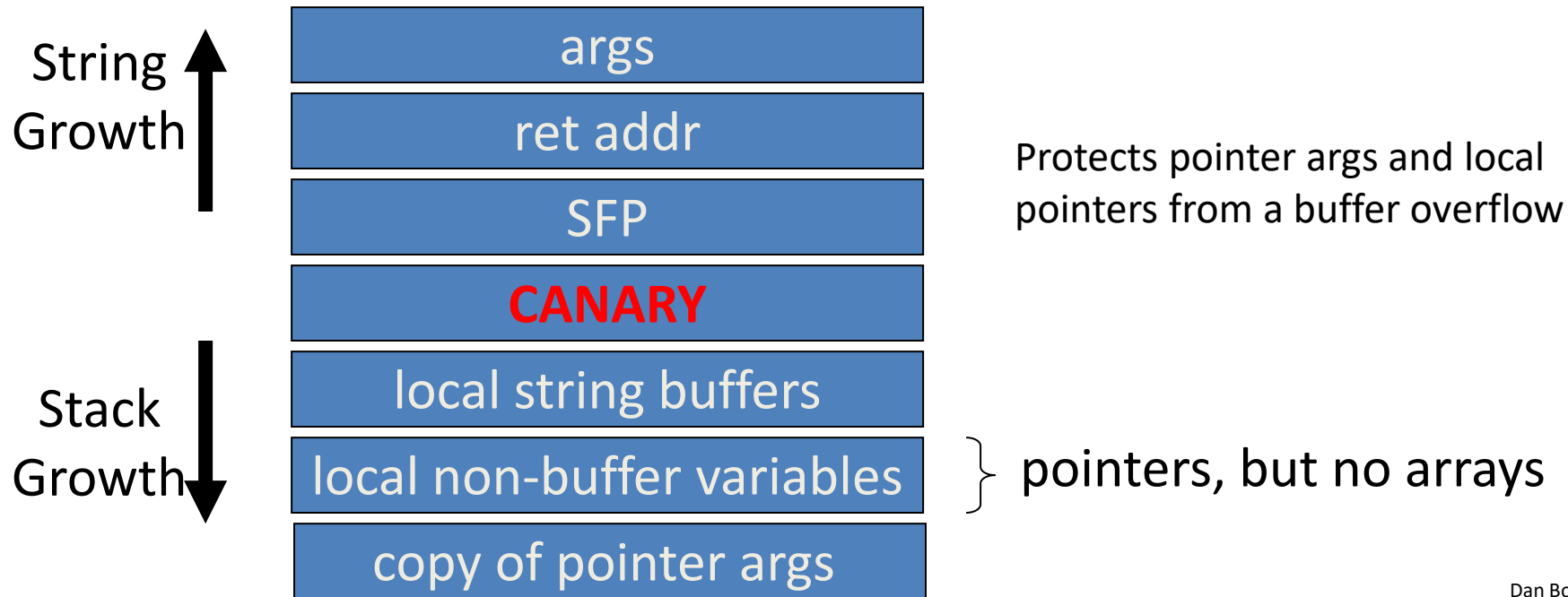
- Random canary:
 - Random string chosen at program startup.
 - Insert canary string into every stack frame.
 - Verify canary before returning from function.
 - Exit program if canary changed. Turns potential exploit into DoS.
 - To corrupt, attacker must learn current random string.
- Terminator canary: Canary = {0, newline, linefeed, EOF}
 - String functions will not copy beyond terminator.
 - Attacker cannot use string functions to corrupt stack.

StackGuard (Cont.)

- StackGuard implemented as a GCC patch
 - Program must be recompiled
- Minimal performance effects: 8% for Apache
- Heap protection: PointGuard
 - Protects function pointers and setjmp buffers by encrypting them: e.g. XOR with random cookie
 - Less effective, more noticeable performance effects

StackGuard enhancements: ProPolice

- ProPolice (IBM) - gcc 3.4.1. (-fstack-protector)
 - Rearrange stack layout to prevent ptr overflow.

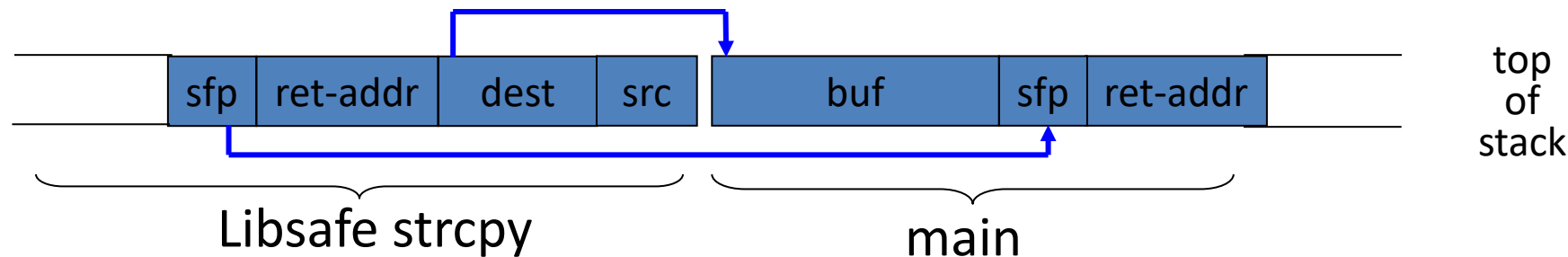


Summary: Canaries are not bullet proof

- Canaries are an important defense tool, but do not prevent all control hijacking attacks:
 - Heap-based attacks still possible
 - Integer overflow attacks still possible
 - /GS by itself does not prevent Exception Handling attacks
(also need SAFESSEH and SEHOP)

What if can't recompile: Libsafe

- Solution 2: Libsafe (Avaya Labs)
 - Dynamically loaded library (no need to recompile app.)
 - Intercepts calls to `strcpy(dest, src)`
 - Validates sufficient space in current stack frame:
$$|\text{frame-pointer} - \text{dest}| > \text{strlen}(\text{src})$$
 - If so, does `strcpy`. Otherwise, terminates application



More methods ...

➤ StackShield

- At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
- Upon return, check that RET and SFP is equal to copy.
- Implemented as assembler file processor (GCC)

➤ Control Flow Integrity (CFI)

- A combination of static and dynamic checking
 - Statically determine program control flow
 - Dynamically enforce control flow integrity

Format string bugs

Format string problem

```
int func(char *user) {  
    fprintf( stderr, user);  
}
```

Problem: what if `*user = "%s%s%s%s%s%s%s"` ??

- Most likely program will crash: DoS.
- If not, program will print memory contents. Privacy?
- Full exploit using `user = "%n"`

Correct form: `fprintf(stdout, "%s", user);`

Vulnerable functions

Any function using a format string.

Printing:

printf, fprintf, sprintf, ...

vprintf, vfprintf, vsprintf, ...

Logging:

syslog, err, warn

Exploit

- Dumping arbitrary memory:
 - Walk up stack until desired pointer is found.
 - `printf("%08x.%08x.%08x.%08x| %s |")`
- Writing to arbitrary memory:
 - `printf("hello %n", &temp)` -- writes '6' into temp.
 - `printf("%08x.%08x.%08x.%08x.%n")`



Control Hijacking

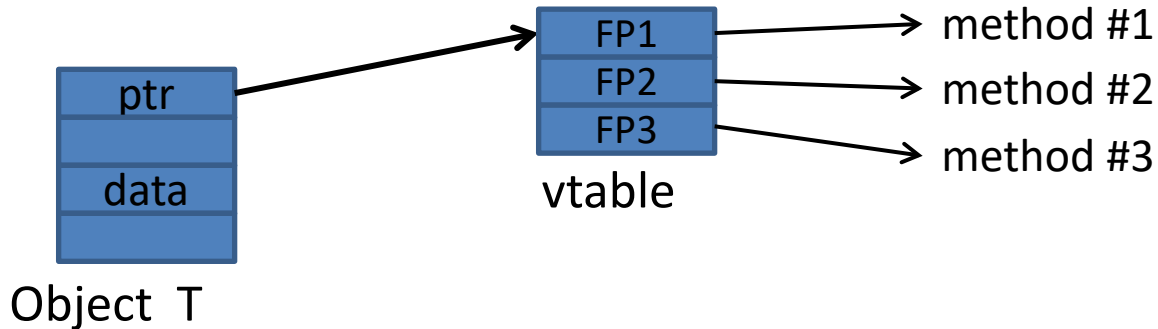
Advanced Hijacking Attacks

Heap Spray Attacks

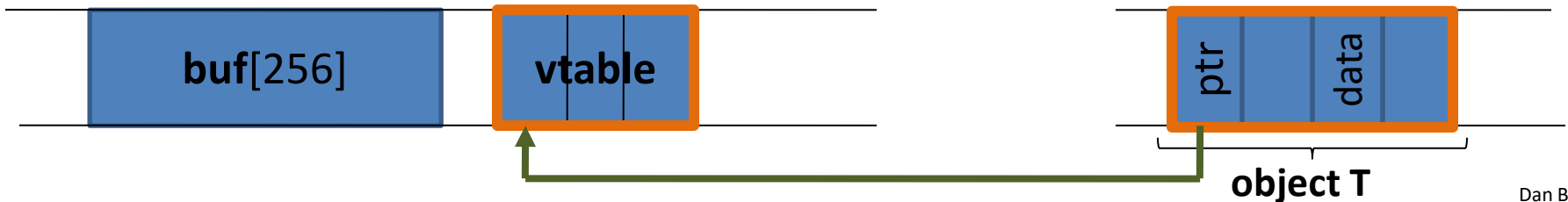
A reliable method for exploiting heap overflows

Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)

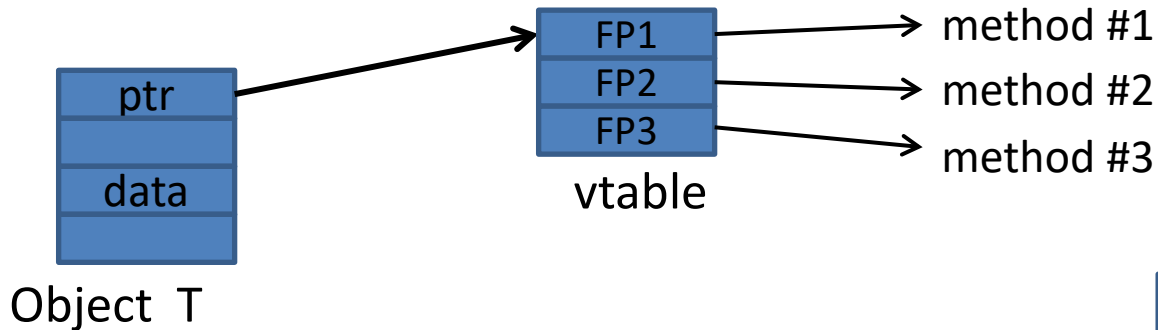


- Suppose `vtable` is on the heap next to a string object:

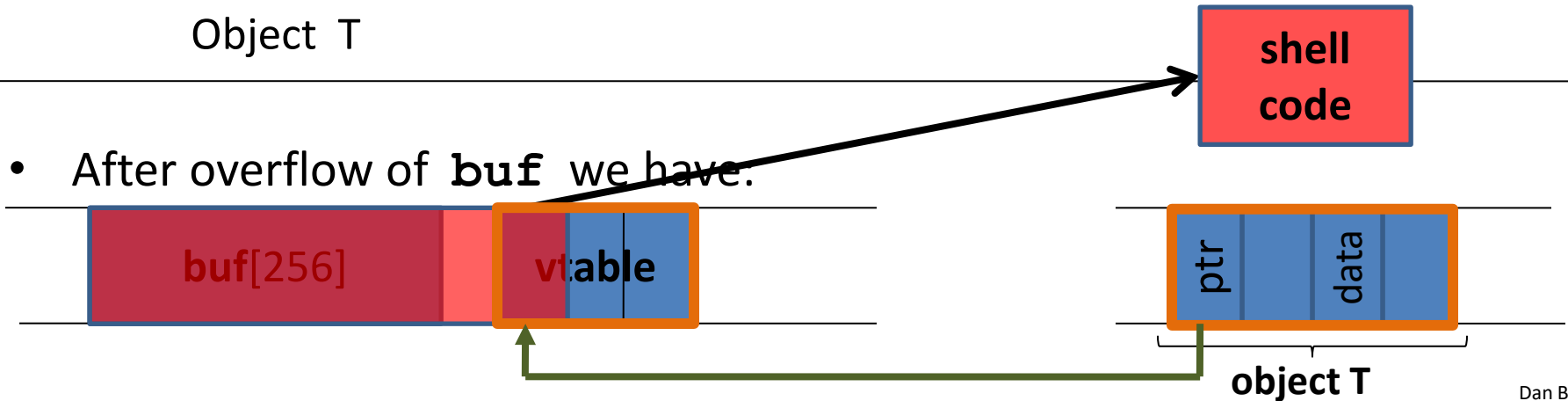


Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)



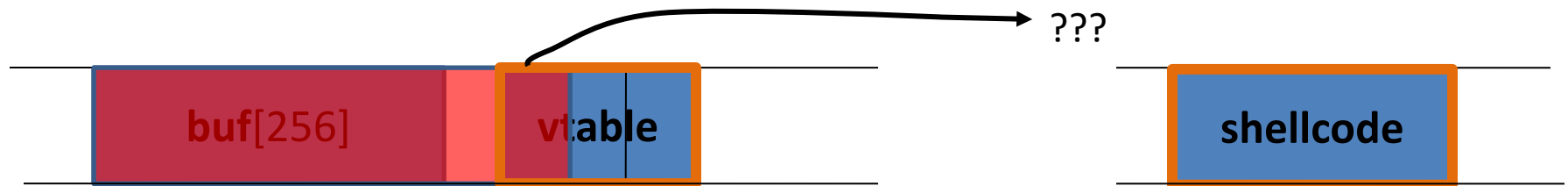
- After overflow of **buf** we have.



A reliable exploit?

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

Problem: attacker does not know where browser places **shellcode** on the heap

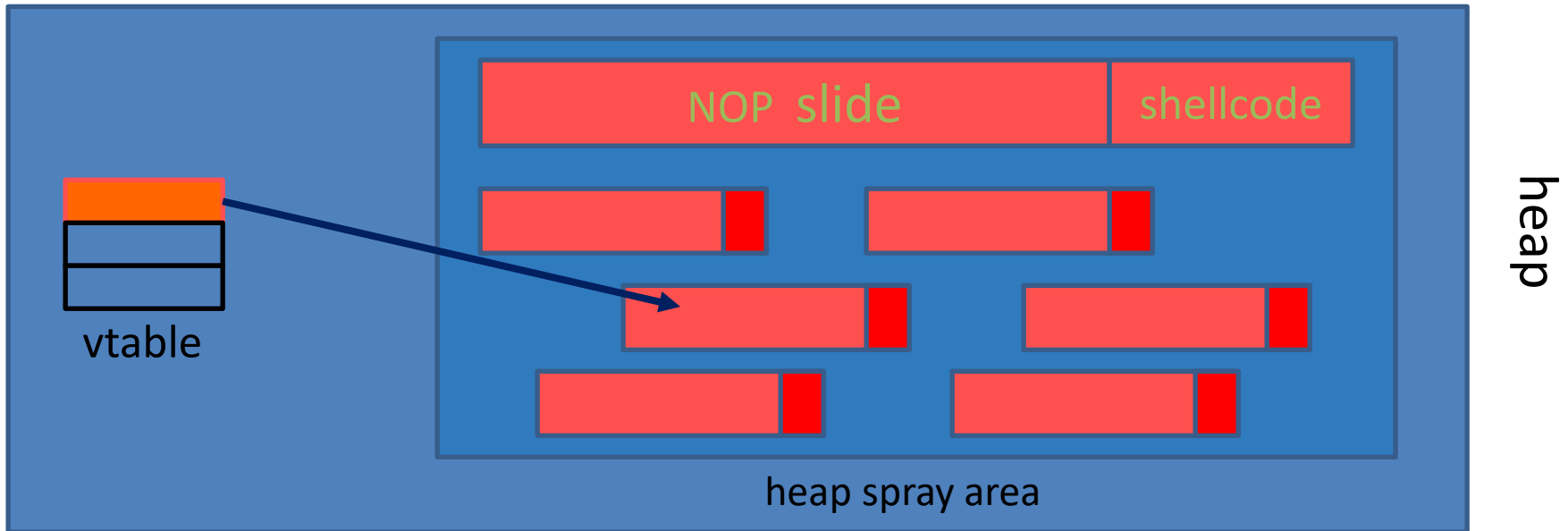


Heap Spraying

[SkyLined 2004]

Idea:

1. use Javascript to spray heap with shellcode (and NOP slides)
2. then point vtable ptr anywhere in spray area



Javascript heap spraying

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

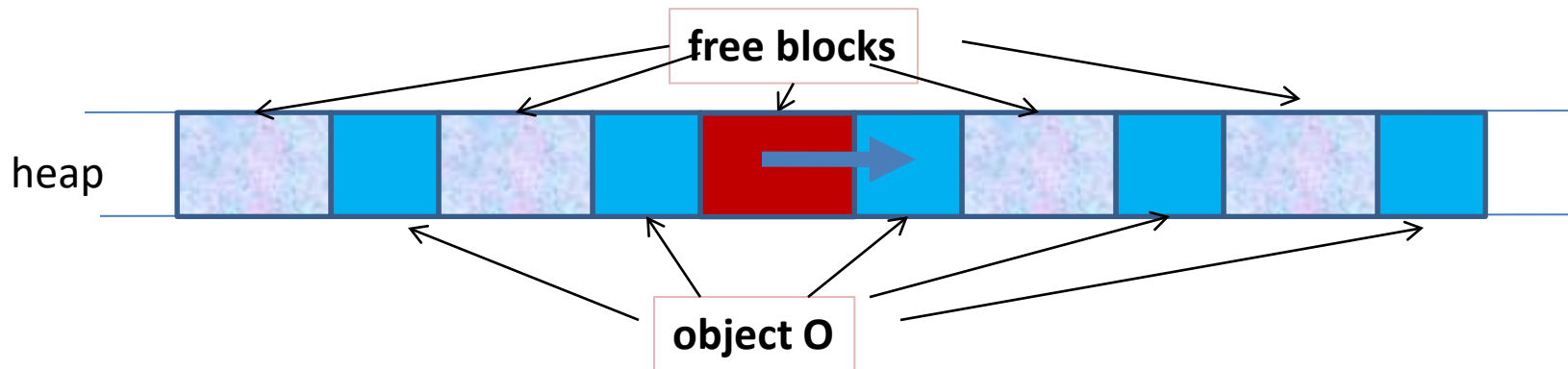
```
var shellcode = unescape("%u4343%u4343%...");
```

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

- Pointing func-ptr almost anywhere in heap will cause shellcode to execute.

Vulnerable buffer placement

- Placing vulnerable `buf[256]` next to object O:
 - By sequence of Javascript allocations and frees make heap look as follows:



- Allocate vuln. buffer in Javascript and cause overflow
- Successfully used against a Safari PCRE overflow [DHM'08]

Many heap spray exploits

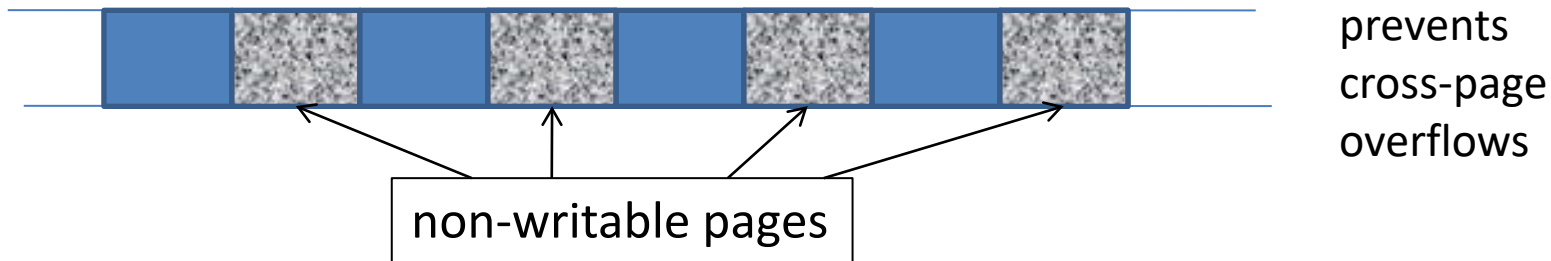
Date	Browser	Description
11/2004	IE	IFRAME Tag BO
04/2005	IE	DHTML Objects Corruption
01/2005	IE	.ANI Remote Stack BO
07/2005	IE	javaprxy.dll COM Object
03/2006	IE	createTextRange RE
09/2006	IE	VML Remote BO
03/2007	IE	ADODB Double Free
09/2006	IE	WebViewFolderIcon setSlice
09/2005	FF	0xAD Remote Heap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

[RLZ'08]

- Improvements: Heap Feng Shui [S'07]
 - Reliable heap exploits **on IE** without spraying
 - Gives attacker full control of IE heap from Javascript

(partial) Defenses

- Protect heap function pointers (e.g. PointGuard)
- Better browser architecture:
 - Store JavaScript strings in a separate heap from browser heap
- OpenBSD heap overflow protection:



- Nozzle [RLZ'08]: detect sprays by prevalence of code on heap

References on heap spraying

- [1] **Heap Feng Shui in Javascript**,
by A. Sotirov, *Blackhat Europe 2007*
- [2] **Engineering Heap Overflow Exploits with JavaScript**
M. Daniel, J. Honoroff, and C. Miller, *WooT 2008*
- [3] **Nozzle: A Defense Against Heap-spraying Code Injection Attacks**,
by P. Ratanaworabhan, B. Livshits, and B. Zorn
- [4] **Interpreter Exploitation: Pointer inference and JiT spraying**,
by Dion Blazakis

End of Segment