Computer Systems Security
CS 628A

Pramod Subramanyan
Indian Institute of Technology Kanpur

MODULE 1: CONTROL HIJACKING

Acknowledgements

- Suman Jana
- Sandeep Shukla
- Dan Boneh (Stanford University)
- John C. Mitchell (Stanford University)
- Nicolai Zeldovich (MIT)
- Jungmin Park (Virginia Tech)
- Patrick Schaumont (Virginia Tech)
- Web Resources

Control Hijacking

Basic Control Hijacking Attacks

Example 1

```c
#include <stdio.h>
int main()
{
	volatile int number = 0;
	char buffer[8];

	printf("number=%d\n", number);
	gets(buffer);
	printf("number=%d\n", number);
	return 0;
}
```

```
$ python -c 'print "a"*8' | ./stack0
$ python -c 'print "a"*9' | ./stack0
```

Example 2

```
int main(int argc, char* argv[])
{
      volatile int number = 0;
      char buffer[16];
      if (argc != 2) return 0;
      printf("number=%d\n", number);
      sprintf(buffer, "hello %s", argv[1]);
      printf("buffer=%s\n", buffer);
      printf("number=%d\n", number);
      return 0;
}
```

```
* $ ./stack1 `python -c 'print "a"*16'`
* $ ./stack1 `python -c 'print "a"*17'`
```

Example 3

```
#include <stdio.h>
#include <string.h>
int checkpwd(const char* pwd)
{
	int good = 0;
	char buffer[8];
	strcpy(buffer, pwd);
	if(strcmp(buffer, "blahblah") == 0) good = 1;
	if(strcmp(buffer, "wahhwahh") == 0) good = 1;
	return good;
}
int main(int argc, char* argv[])
{
	if (argc != 2) return 1;
	if (checkpwd(argv[1])) {
		printf("====================\n");
		printf("|| access granted  ||\n");
		printf("====================\n");
	} else { printf("denied!\n"); }
	return 0;
}
```

How do we exploit this?

Example 4

```
#include <stdio.h>
#include <stdlib.h>

void win()
{
	printf("you win!\n");
}

int main(int argc, char* argv[])
{
	char buffer[16];

	printf("win addr=%p\n", win);
	gets(buffer);
	printf("buffer=%s\n", buffer);

	return 0;
}
```

How do we exploit this?

Example 5

```
#include <stdio.h>

int main(int argc, char* argv[])
{
        volatile int number = 0;
        char *buffer = new char[16];
        if (argc != 2) return 0;

        printf("number=%d\n", number);
        sprintf(buffer, "hello %s", argv[1]);
        printf("buffer=%s\n", buffer);
        printf("number=%d\n", number);

        delete [] buffer;
        return 0;
}
```

How do we exploit this?

Example 6

See Heap1.cpp

How do we exploit this?

Resources

- The Hacking Book
  - Linked to in Moodle
  - Read the chapter on exploitation
  - These examples are based on code in the book
- Examples posted on course website
  - Run them on your computer
  - Play around with them in gdb: look at the addresses, step through them instr by instr
  - Do this now, and enjoy stress-free CTFs

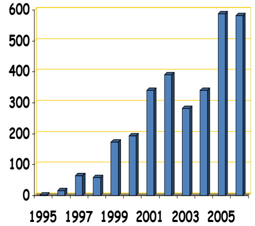
Control hijacking attacks

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

- Examples.
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities

Example 1: buffer overflows

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



≈20% of all vuln.

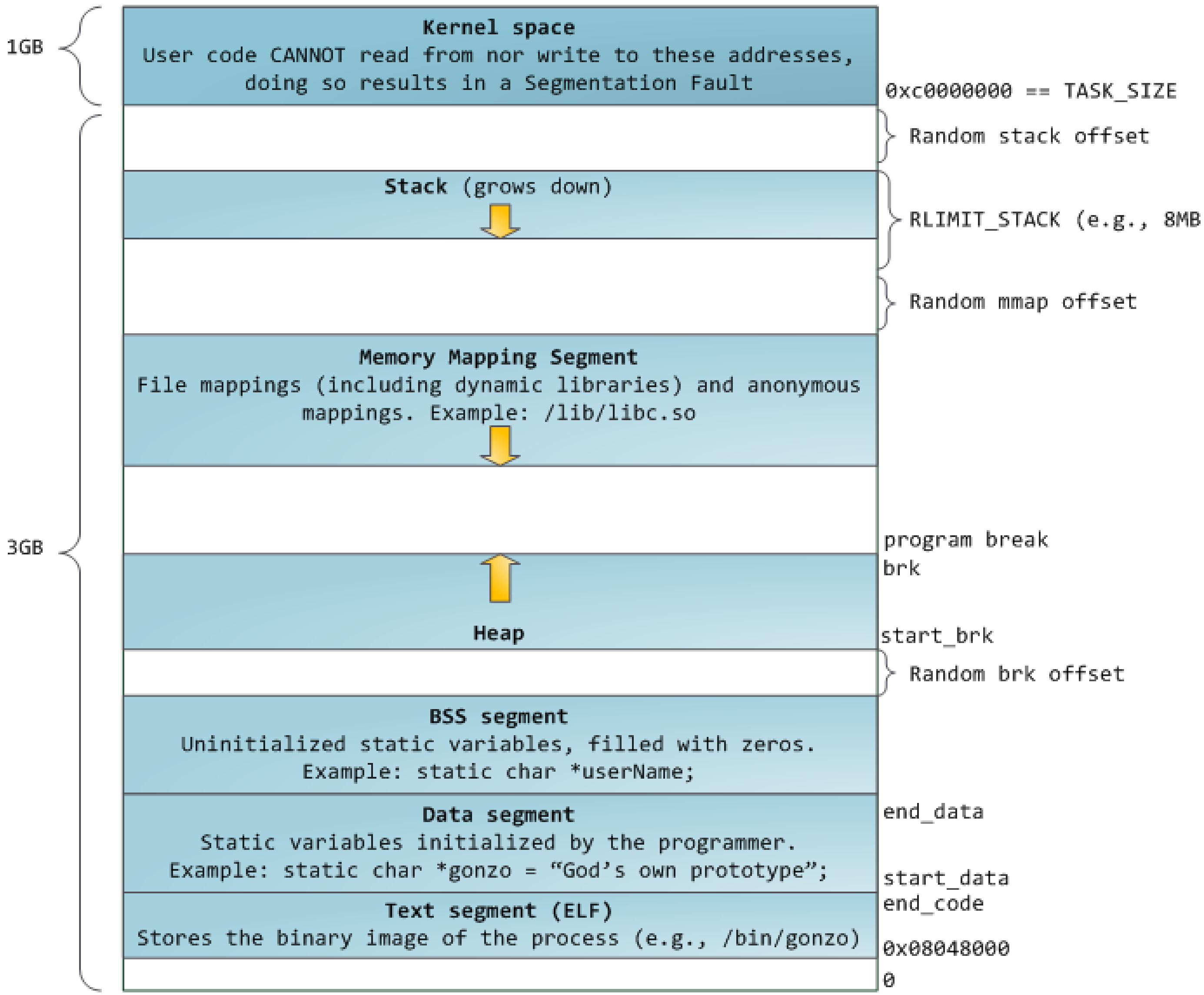
Source: NVD/CVE

What is needed

- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

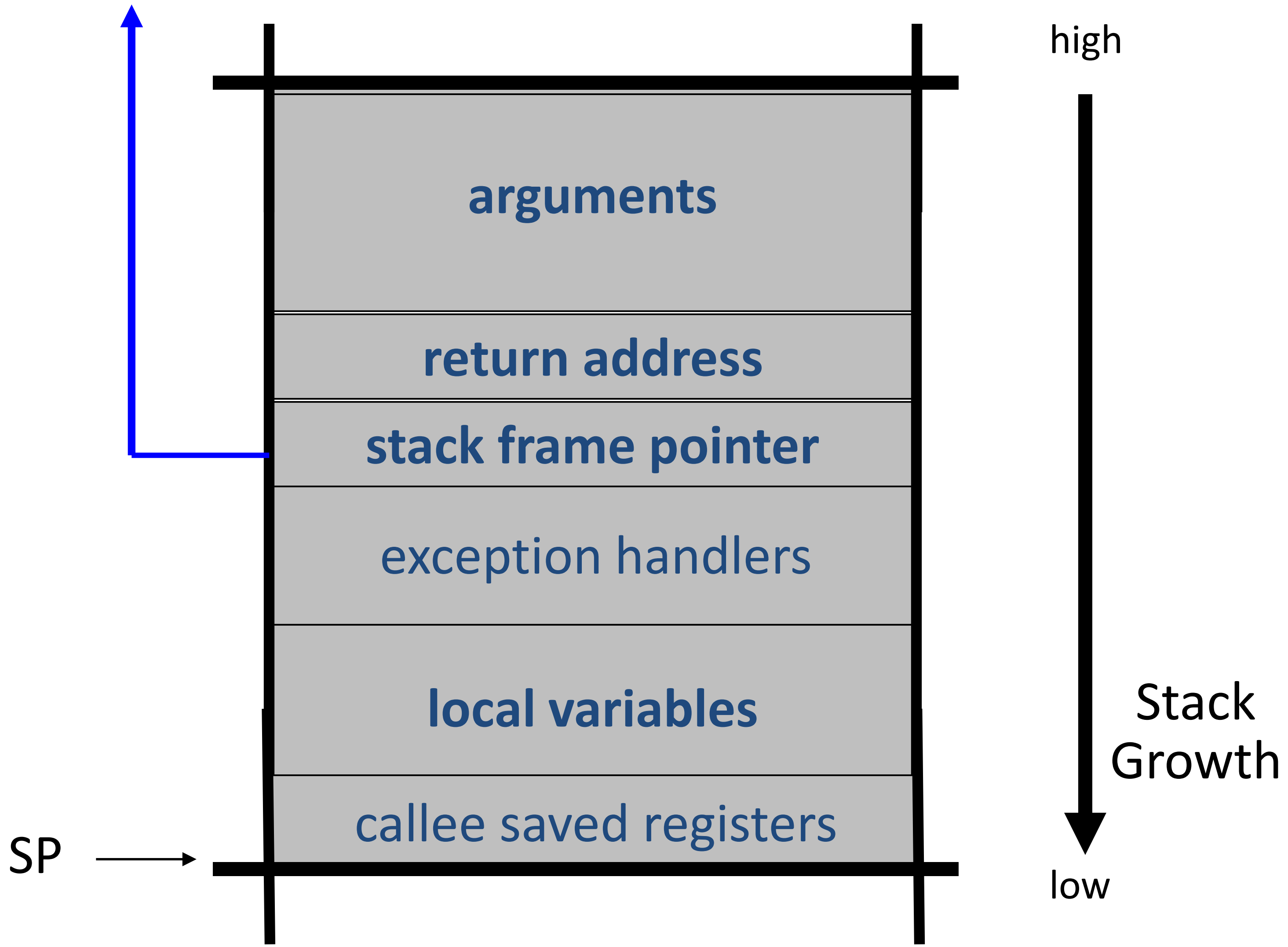
- Attacker needs to know which CPU and OS used on the target machine:
  - Our examples are for x86 running Linux or Windows
  - Details vary slightly between CPUs and OSs:
    - Little endian vs. big endian (x86 vs. Motorola)
    - Stack Frame structure (Unix vs. Windows)

Linux Process Memory Layout



Stack Frame

http://post.queensu.ca/~trd/377/tut5/stack.html

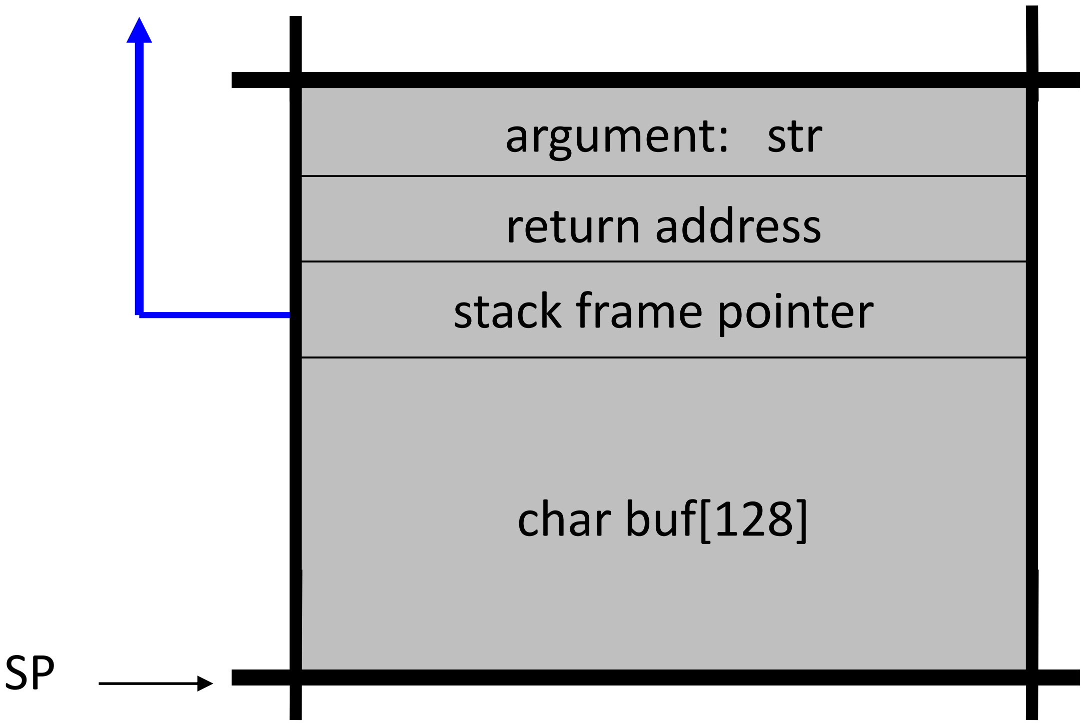


What are buffer overflows?

Suppose a web server contains a function:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

When func() is called stack looks like:

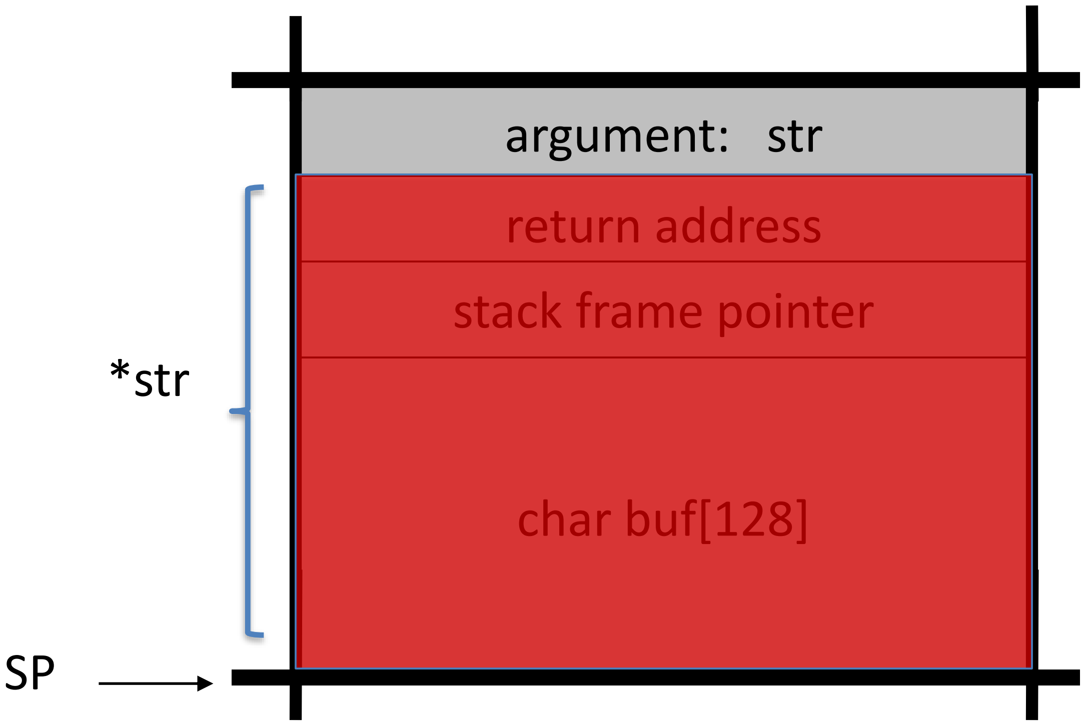


What are buffer overflows?

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()

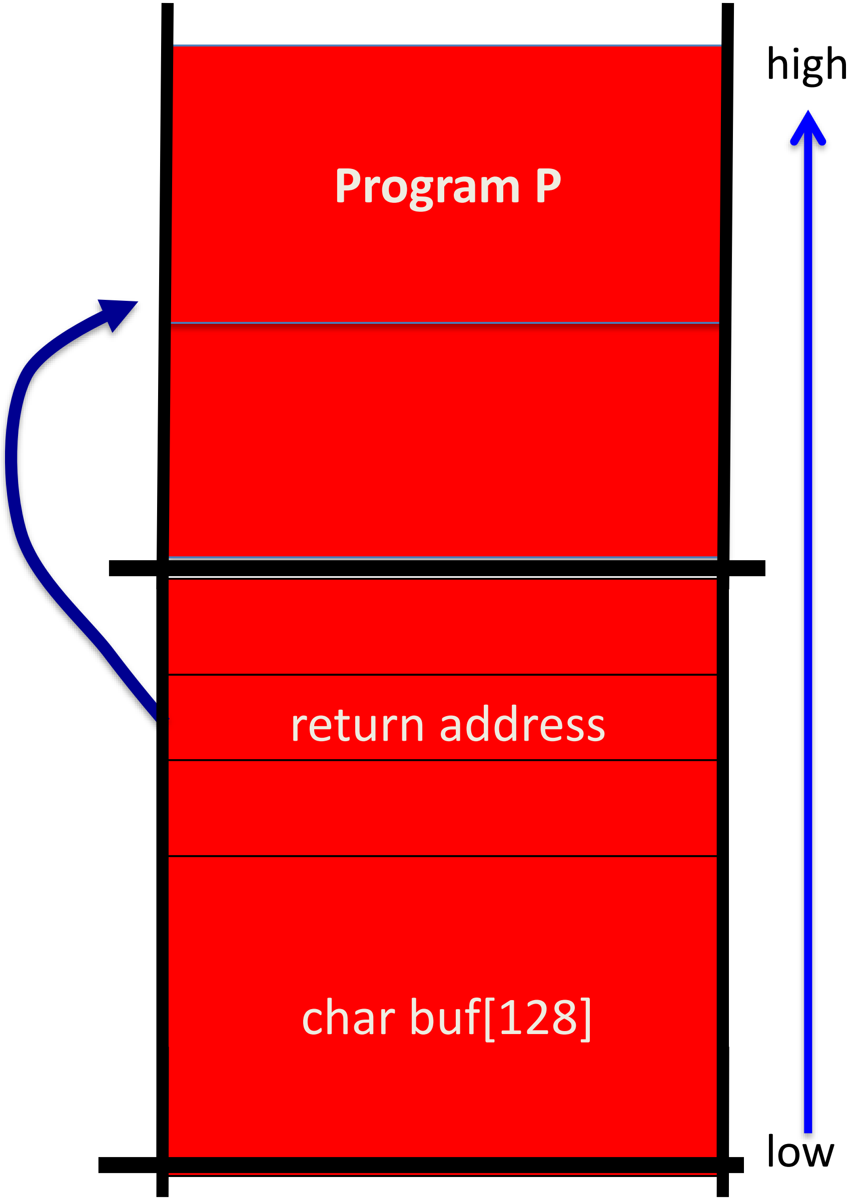
Basic stack exploit

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

Program P: exec("/bin/sh")

When func() exits, the user gets shell !

Note: attack code P runs *in stack*.



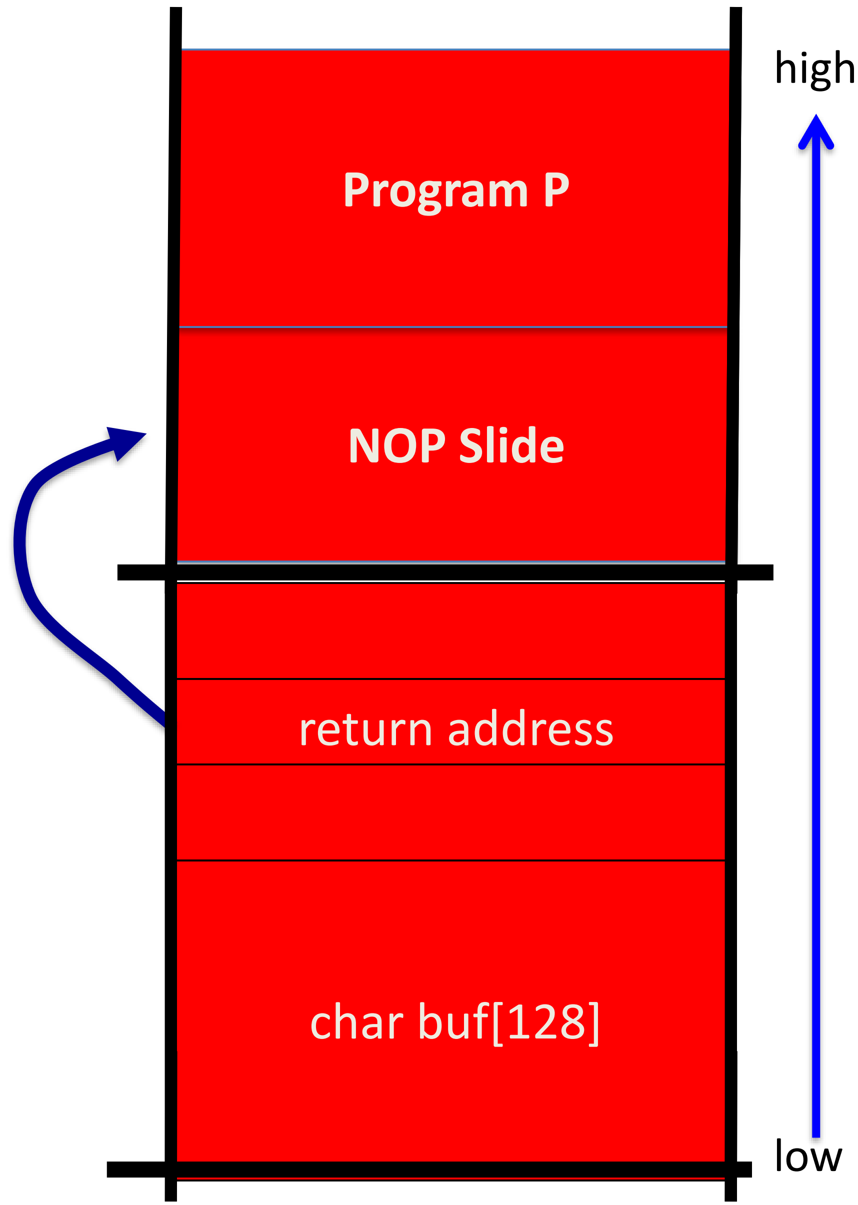
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when func() is called
- 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 <u>remote</u> stack smashing overflows:
  - (2007) Overflow in Windows animated cursors (ANI). LoadAniIcon() https://www.sans.org/reading-room/whitepapers/threats/ani-vulnerability-history-repeats-1926
  - (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)

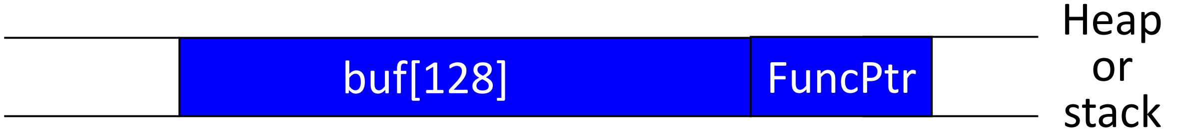
scanf (const char *format, …) and many more.

- “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)

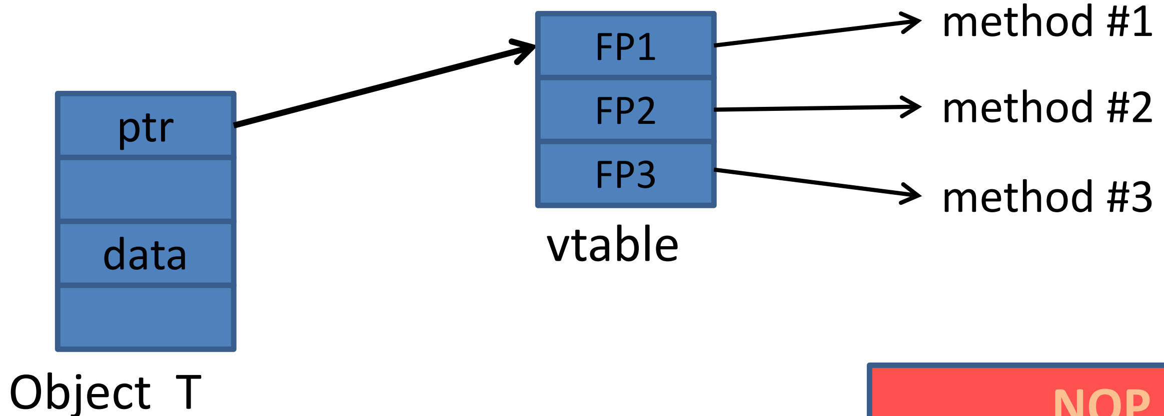


  - Overflowing buf will override function pointer.

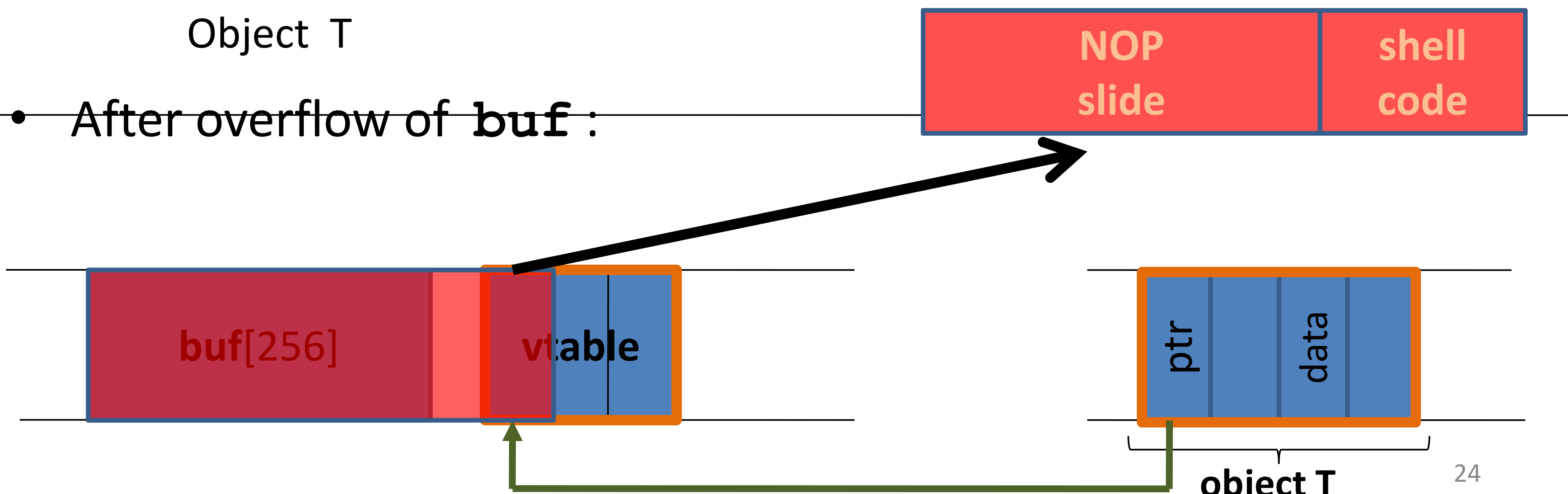
  - Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

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 "$$$$$")
    - Many automated tools exist (called fuzzers)
  - If web server crashes,
    search core dump for "$$$$$" 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
- **Use after free:** using memory after it is freed
- **Format string vulnerabilities**

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 ⇒ c = 0

s = 0xff80 + 0x80 ⇒ s = 0

m = 0xffffff80 + 0x80 ⇒ m = 0

Can this be exploited?

An example

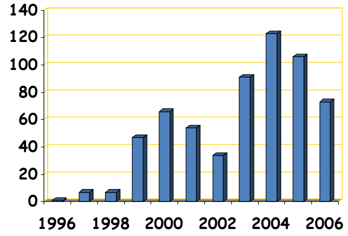
```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      if  (len1 + len2 > 256)  {return -1}          // length check
      memcpy(temp, buf1, len1);                     // cat buffers
      memcpy(temp+len1, buf2, len2);
      do-something(temp);                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

What if **len1 = 0x80,** **len2 = 0xffffff80** ?

$\Rightarrow$ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer overflow exploit stats



Source: NVD/CVE

Format string bugs

Format String Example 1

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Example of a %n format string
   printf("The number of bytes written up to this
point X%n is being stored in count_one, and the
number of bytes up to here X%n is being stored in
count_two.\n", &count_one, &count_two);

   printf("count_one: %d\n", count_one);
   printf("count_two: %d\n", count_two);

   // Stack Example
   printf("A is %d and is at %08x.  B is %x.\n", A,
&A, B);

   exit(0);
}
```

```
$ ./a.out
The number of bytes written up to this point X is being
stored in count_one, and the number of bytes up to
here X is being storied in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff7f4. B  is 7.
```

Format String Example 2

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Example of a %n format string
   printf("The number of bytes written up to this point X%n is
being stored in count_one, and the number of bytes up to here X%n
is being stored in count_two.\n", &count_one, &count_two);

   printf("count_one: %d\n", count_one);
   printf("count_two: %d\n", count_two);

   // Stack Example
   printf("A is %d and is at %08x.  B is %x.\n", A, &A);

   exit(0);
}
```

$ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number of bytes up to here X is being storied in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff7f4. B is b7fd6ff4

Format String Example 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
   char text[1024];
   static int test_val = -72;

   if(argc < 2) {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf("The right way to print user-controlled input:\n");
   printf("%s", text);


   printf("\nThe wrong way to print user-controlled input:\n");
   printf(text);

   printf("\n");

   // Debug output
   printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
test_val);

   exit(0);
}
```

$ ./fmt_vuln testing%x

$ ./fmt_vuln $(perl –e ‘print “%08x.”x40’)

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?

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

Platform Defenses

Preventing hijacking attacks

1. Fix bugs:
   – Audit software
     - Automated tools: Coverity, Prefast/Prefix.
   – Rewrite software in a type safe languange (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, …

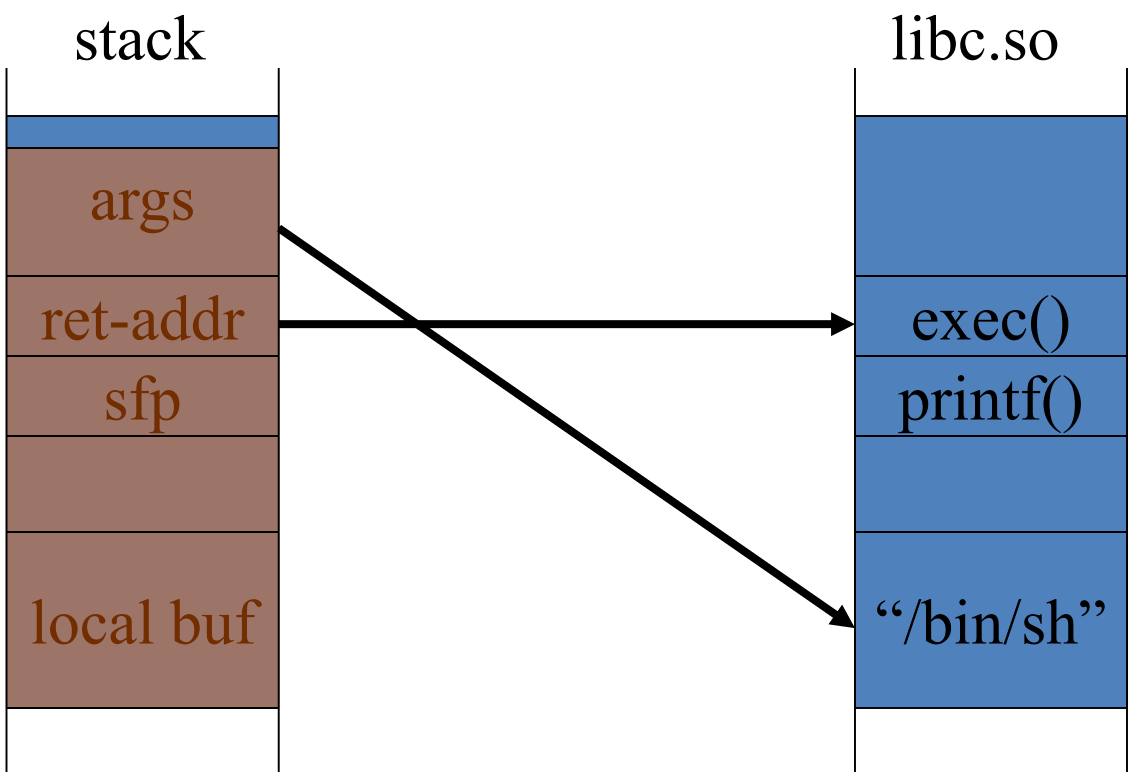
Marking memory as non-execute (W^X)

Prevent attack code execution by marking stack and heap as
non-executable – DEP – Data Execution Prevention

- 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

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 7**: 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, base image

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

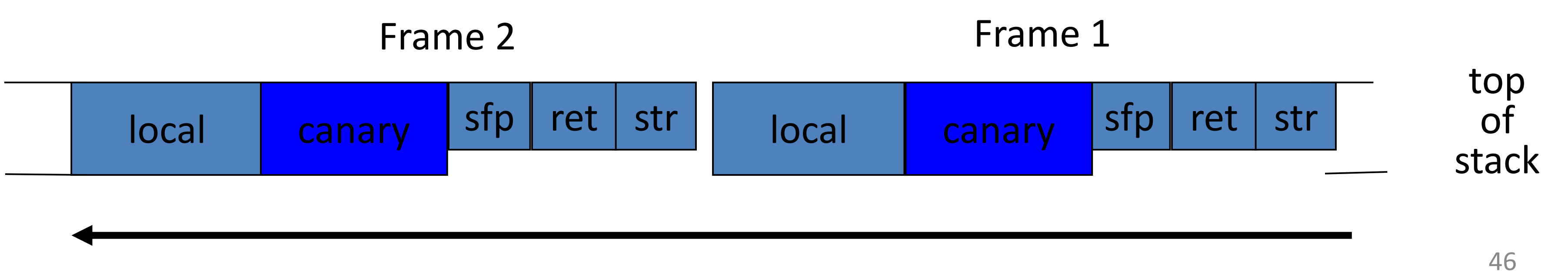
Control Hijacking

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

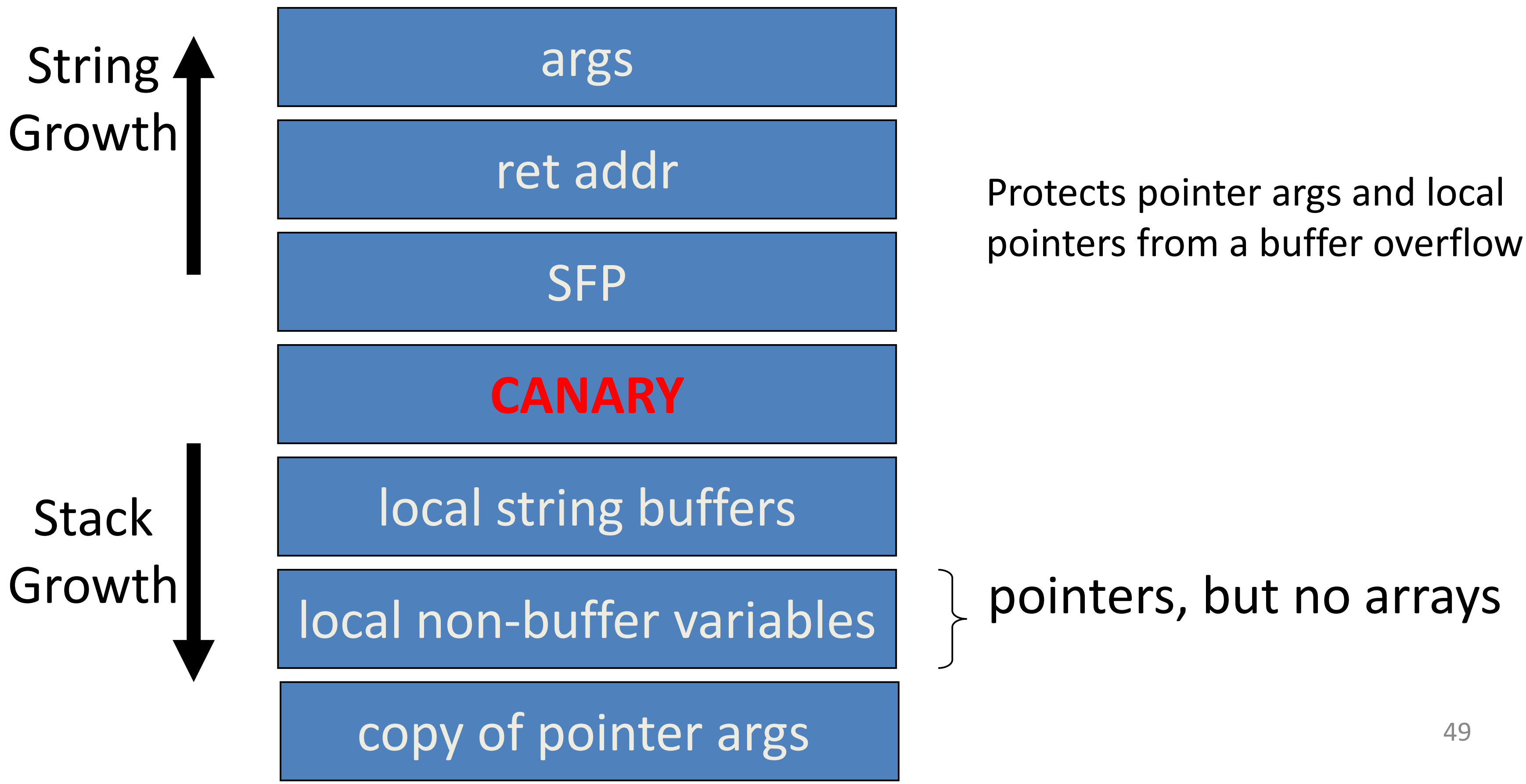
- Minimal performance effects: 8% for Apache

- Note: Canaries do not provide full protection
  - Some stack smashing attacks leave canaries unchanged

- 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.



MS Visual Studio /GS [since 2003]

Compiler /GS option:

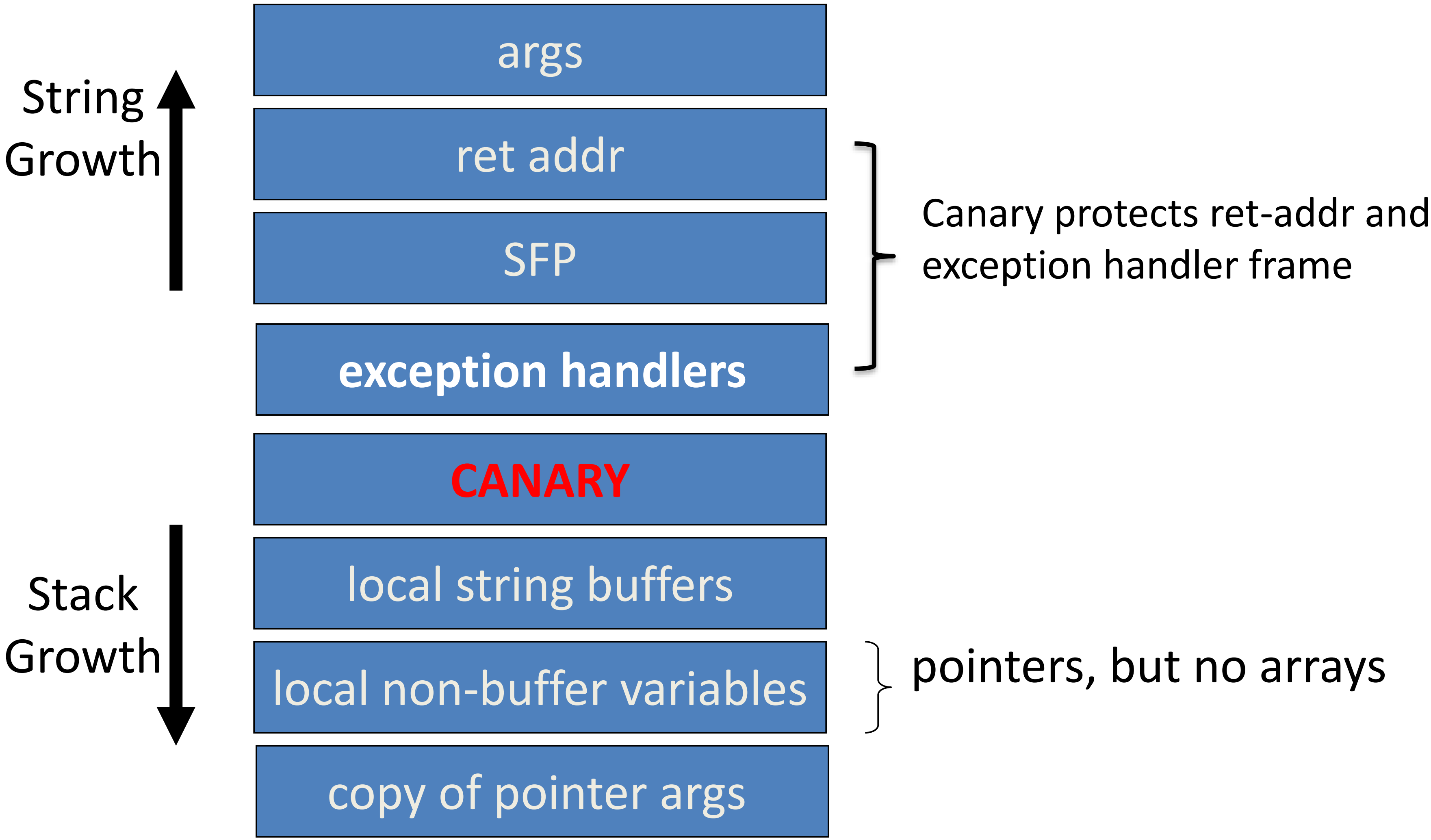
- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call **_exit(3)**

Function prolog:	Function epilog:
sub esp, 8 // allocate 8 bytes for cookie **mov eax, DWORD PTR ___security_cookie** **xor eax, esp** // xor cookie with current esp **mov DWORD PTR [esp+8], eax** // save in stack	**mov ecx, DWORD PTR [esp+8]** **xor ecx, esp** **call @__security_check_cookie@4** **add esp, 8**

Enhanced /GS in Visual Studio 2010:

- /GS protection added to all functions, unless can be proven unnecessary

/GS stack frame



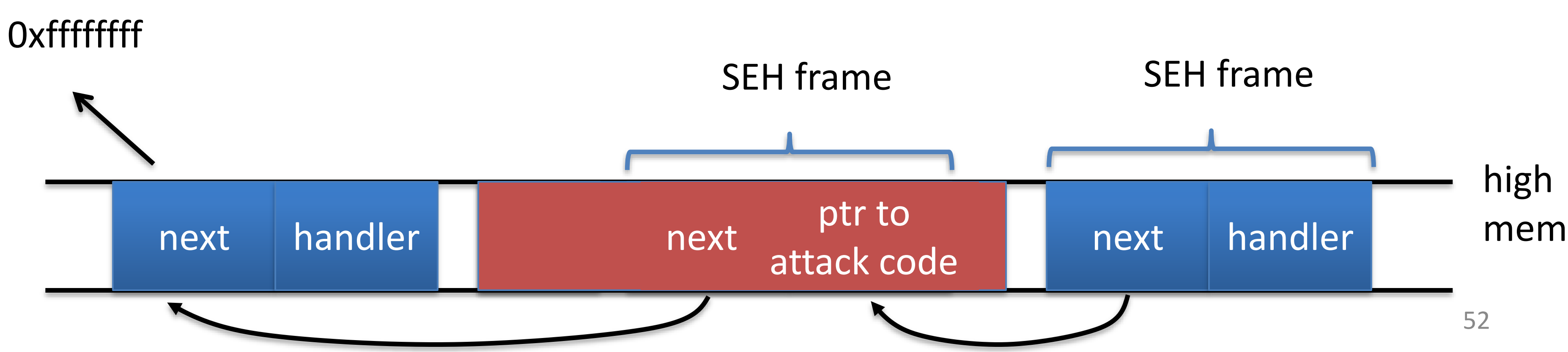
Evading /GS with exception handlers

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker's code

exception triggered ⇒ control hijack

Main point: exception is triggered before canary is checked



Defenses: SAFESEH and SEHOP

- /SAFESEH: linker flag
  - Linker produces a binary with a table of safe exception handlers
  - System will not jump to exception handler not on list

- /SEHOP: platform defense (since win vista SP1)
  - Observation: SEH attacks typically corrupt the “next” entry in SEH list.
  - SEHOP: add a dummy record at top of SEH list
  - When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.

Summary: Canaries are not full 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

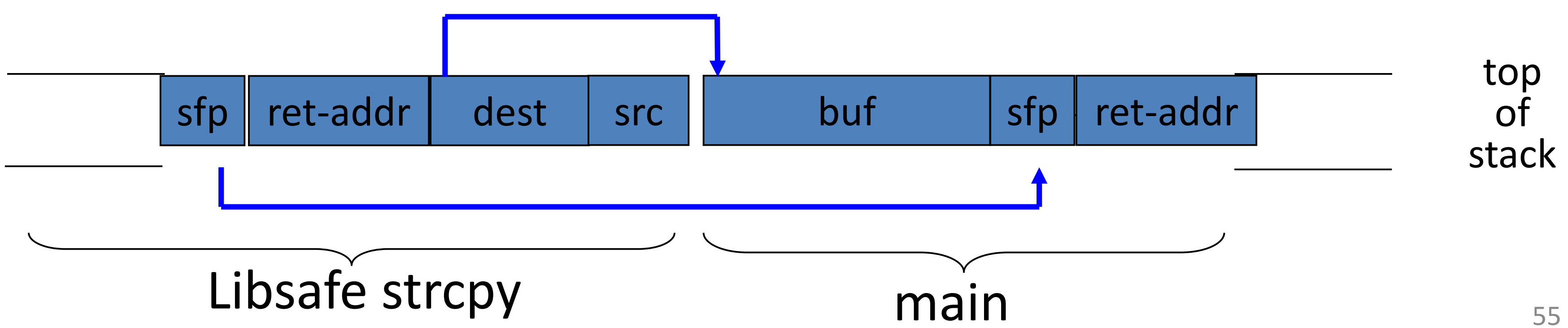
    (also need SAFESEH and SEHOP)

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:

      |frame-pointer – dest| > strlen(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

Control Hijacking

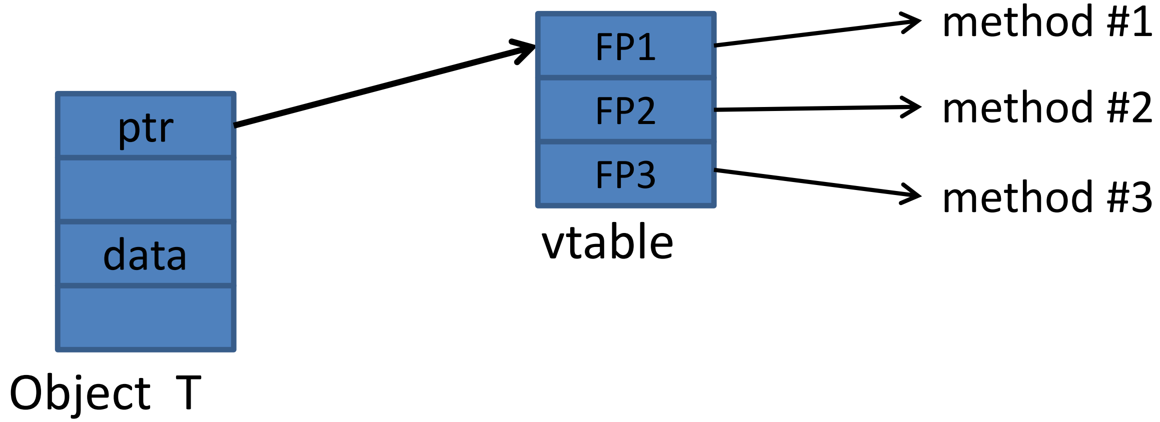
Advanced Hijacking Attacks

Heap Spray Attacks

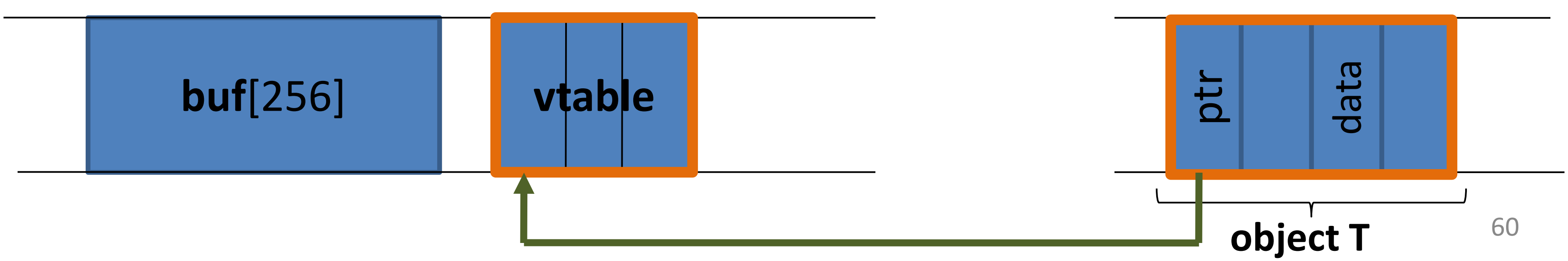
A reliable method for exploiting heap overflows

Heap-based control hijacking

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

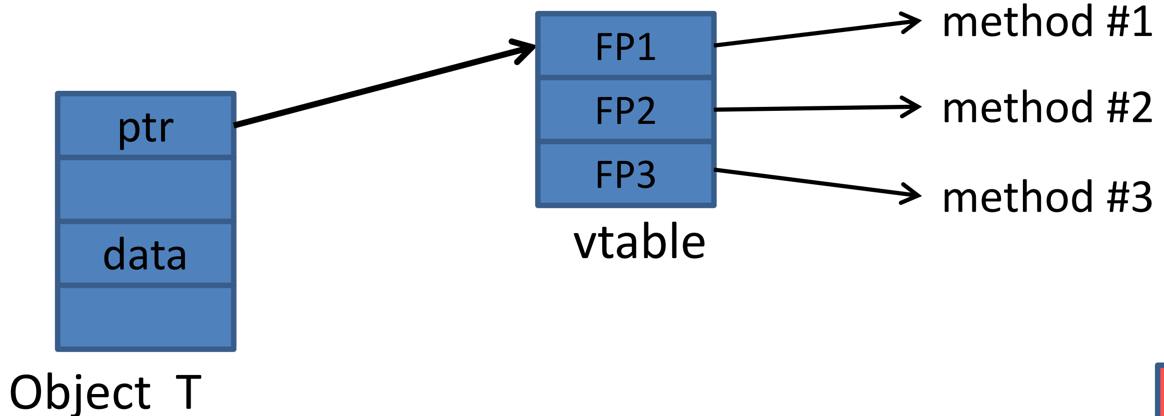


- Suppose   vtable   is on the heap next to a string object:

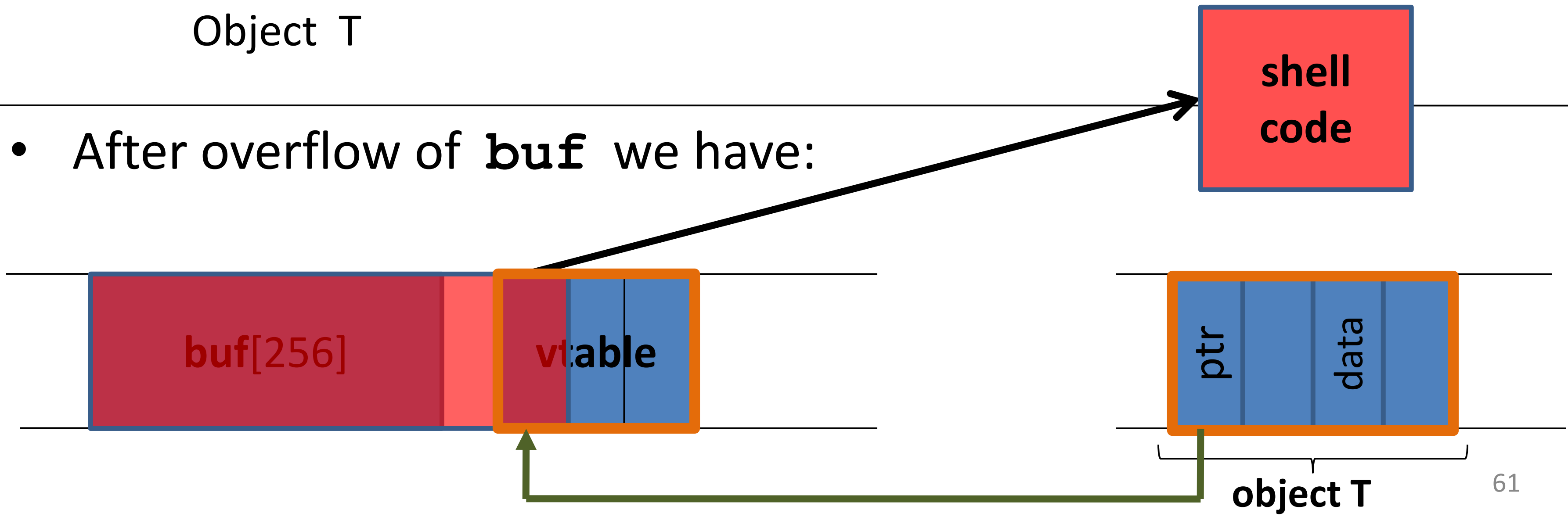


Heap-based control hijacking

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



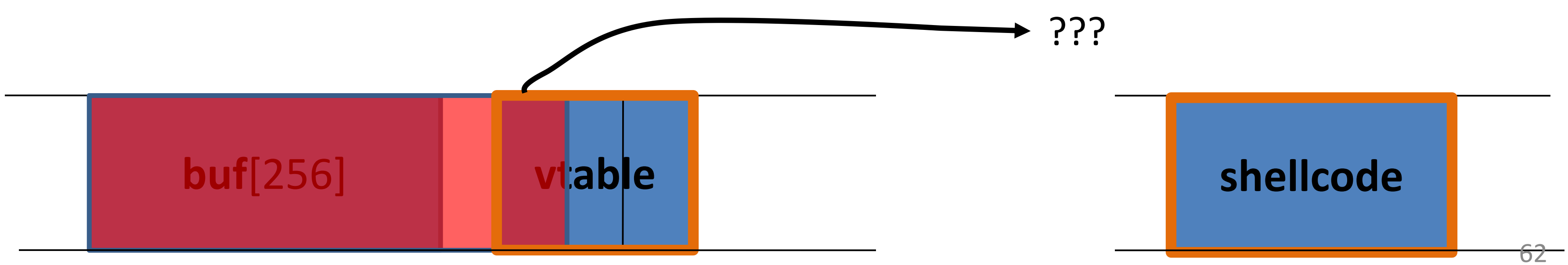
- After overflow of **`buf`** we have:



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

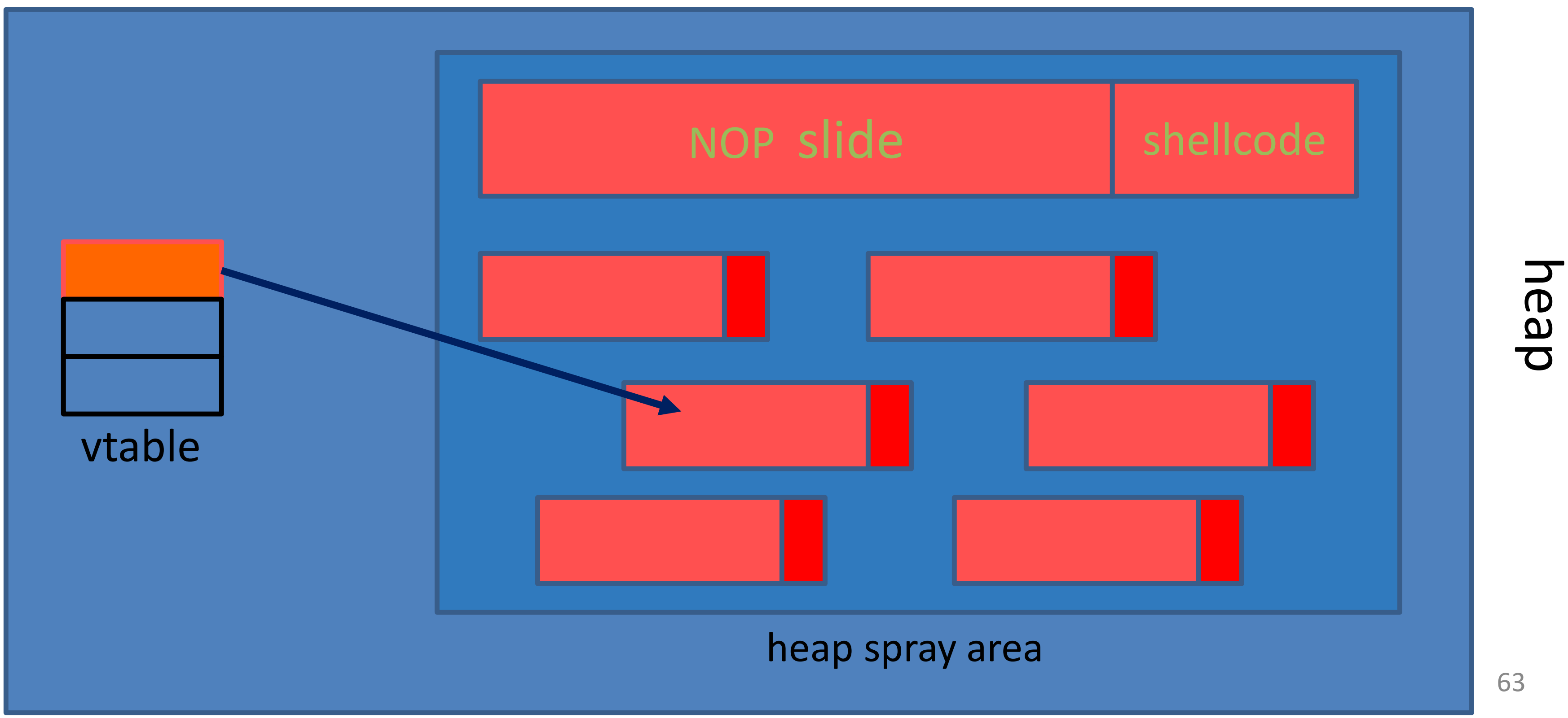


Heap Spraying [SkyLined 2004]

Idea:

1. use Javascript to spray heap
 with shellcode (and NOP slides)

2. then point vtable ptr anywhere in spray area



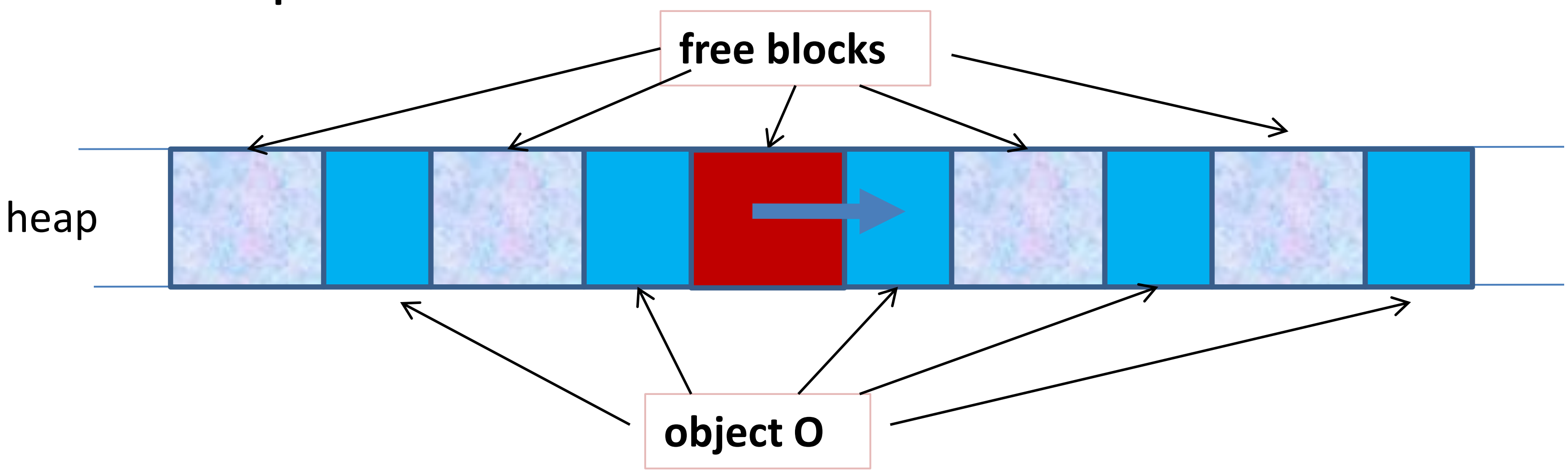
Javascript heap spraying

```
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

[RLZ'08]

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

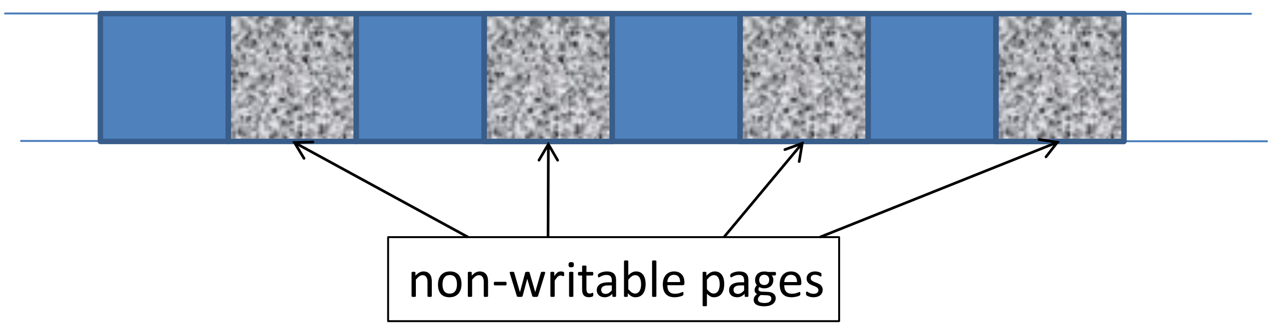
- Improvements: Heap Feng Shui [S'07]
  - Reliable heap exploits **on IE** without spraying
  - Gives attacker full control of IE heap from Javascript

(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:



prevents cross-page overflows

- 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