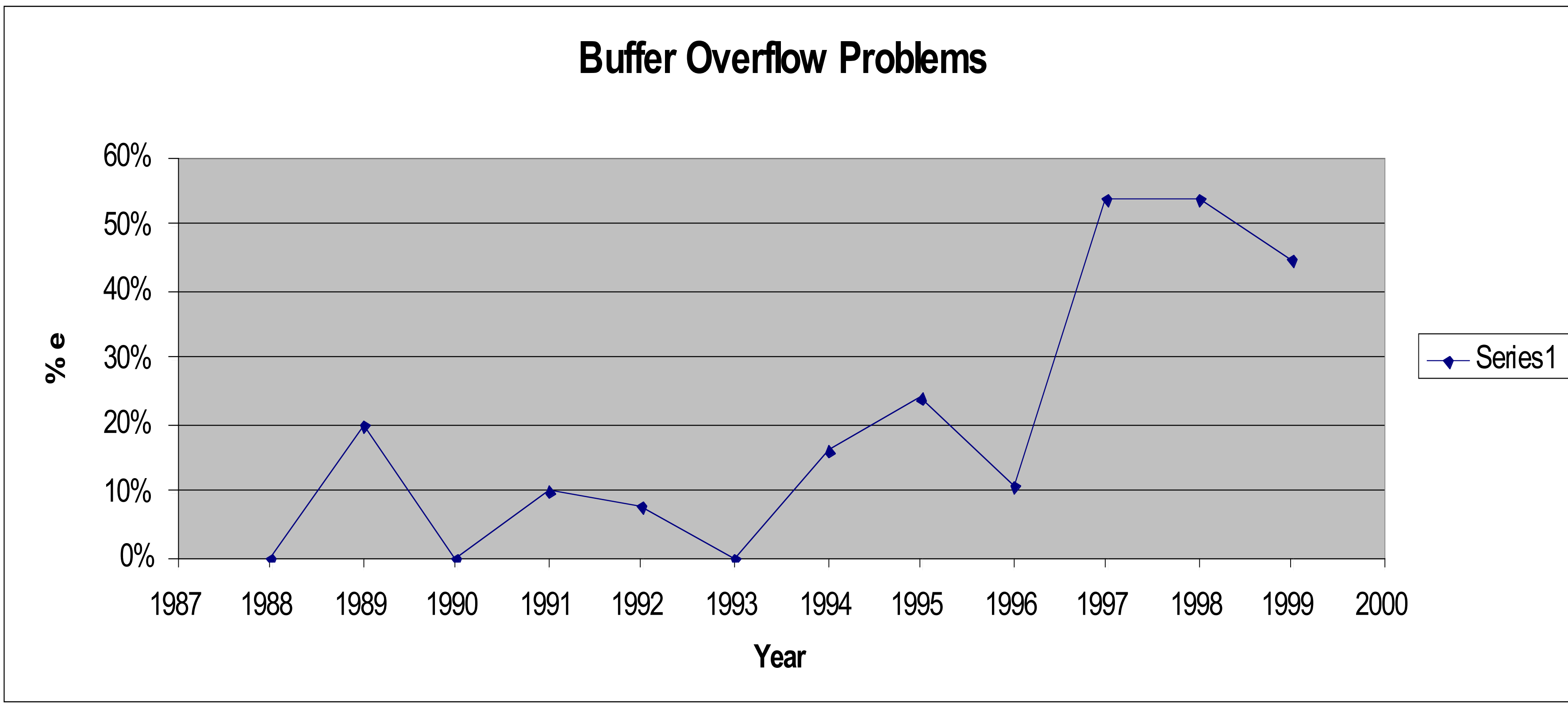
Buffer Overflow

Buffer Overflow

- Single biggest software security threat – the buffer overflow
- The most common form of security vulnerability till 2005 or so.
- Buffer overflow vulnerabilities dominate in the area of remote network penetration vulnerabilities
- Some statistics: Buffer overflow problems as % of CERT alerts



What is buffer overflow?

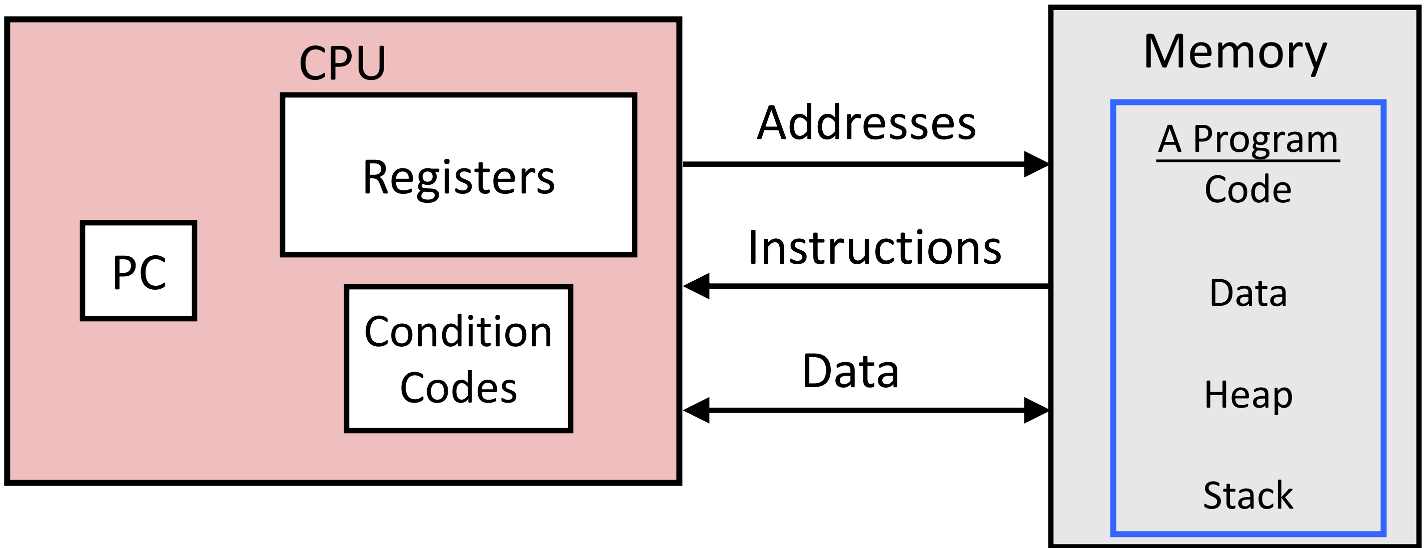
- Buffers in C/C++ program:
  - Heap: the kind of data when you call “malloc()” or “new”
  - Stack: non-static local variables and function parameters, e.g.

    ```
    int func( ) {
      char buf[12];      // a buffer of 12 bytes
    …
    ```
- Buffer overflow:
  - Store more data in the buffer (heap or stack) than it can hold
  - The next contiguous chunk of memory is overwritten
- Why in C/C++ language?
  - C/C++ language is inherently unsafe, i.e. it allows programs to overflow buffers at will
  - No runtime checks that prevent writing past the end of a buffer, e.g.

    ```
      strcpy(buf, “this string takes 27 bytes”);   // copy 27 bytes to 12 bytes buffer
    ```

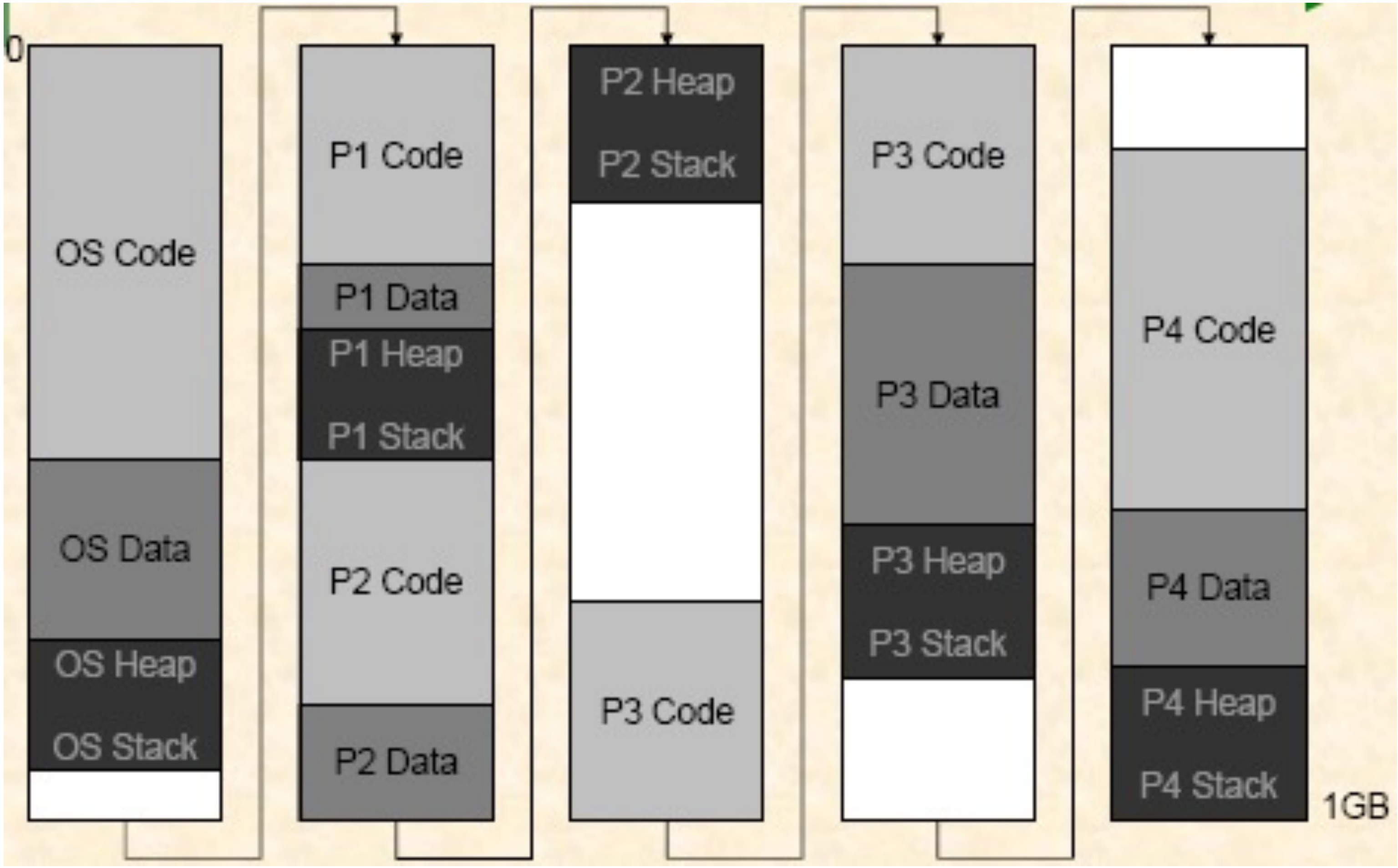
Technical Principles of Buffer Overflow

How a Computer executes a Program



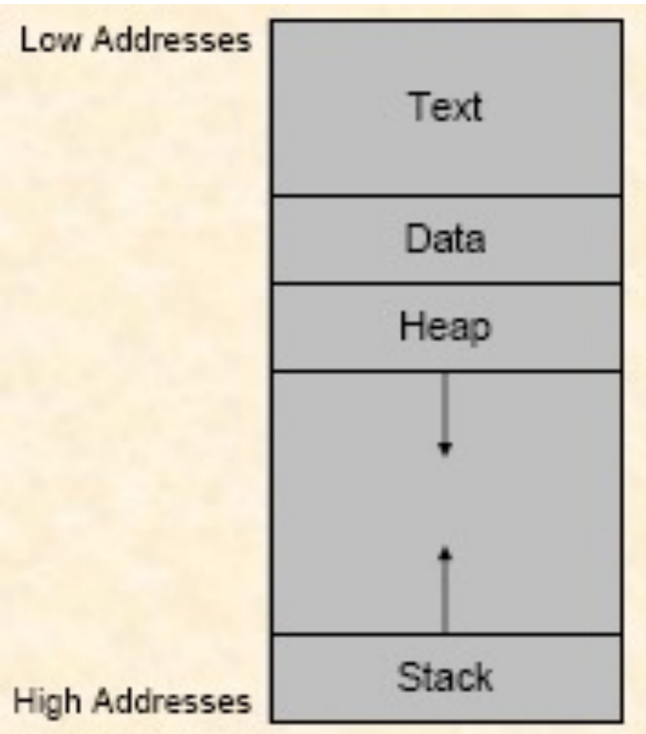
- **PC: Program Counter**
  - Store the Address of Next instruction
- **Registers**
  - Fast circuits to hold for Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
    - IF-THEN-ELSE
- **Memory**
  - Byte addressable array
  - Code: Instructions for the machine
  - Data for the program
  - Heap: for dynamic data storage
  - Stack to support "function" calls

Processes (Running Programs) in the Computer Memory



A Process in Memory

e.g.
000000

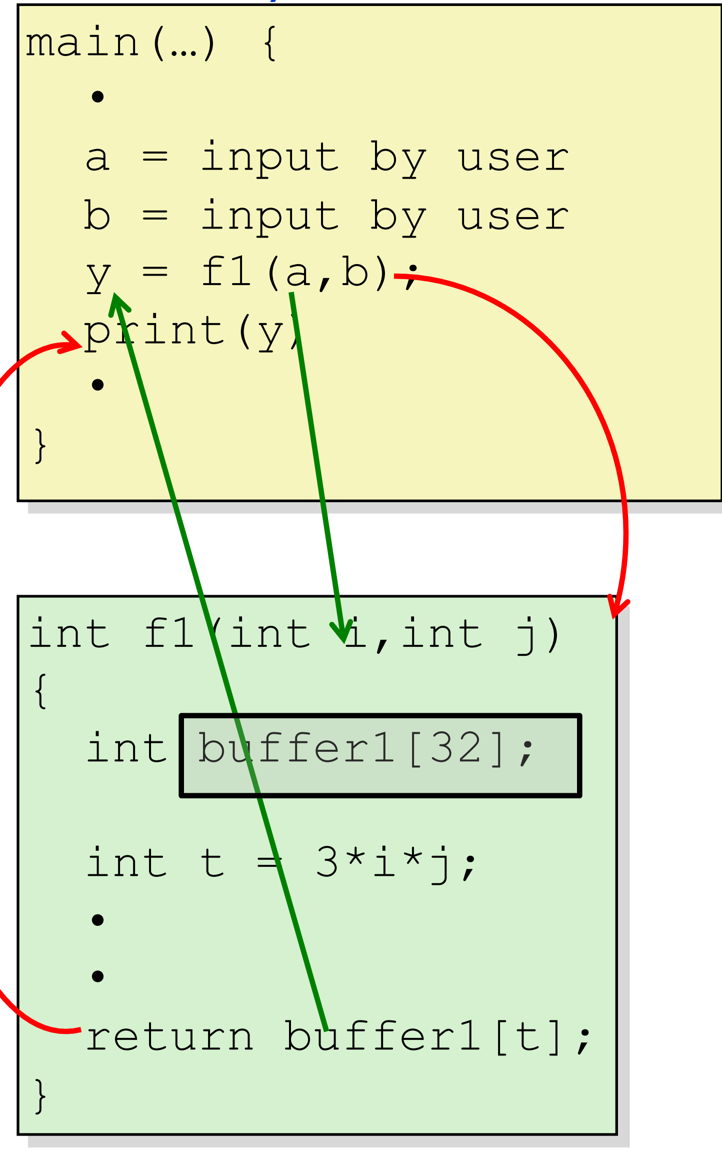


e.g.
FFFFFF

- Code (aka the Text segment) :
  - Program code;
    - marked read-only, so any attempts to write to it will result in segmentation fault
- Data segment:
  - Global and static variables, constants ;
- Heap:
  - Dynamic storage space allocated via via malloc( ), new( ) ;
- Stack:
  - also for storing Dynamic variables
  - Key data structure for implementing Function Calls !!

Mechanisms to Call a Function (aka subroutine/ procedure) within a Program

- Passing control (back and forth)
  - Jump to beginning of the function code (i.e. the starting address of `f1()`)
  - Jump Back to return point upon completion of function
- Passing data (back and forth)
  - Function arguments (i.e. `a,b`)
  - Return value (i.e. `buffer1[t]`)
- Memory management
  - Allocate local storage (i.e. `t`, `buffer1[32]`) during function execution
  - De-allocate those memory upon completion of function
- Mechanisms all implemented with machine instructions by storing required info in the Stack



What Info do we need to call a Function ?

- Values of Arguments Input to the Function (i.e. a, b for f1( ))
- Address to Return to when the Function call is completed
- Memory space for Local (Function) Variables (i.e. for buffer1[32])
- Way to Restore (clean-up) the Stack so that it looks the same as before the Function call
- Starting Address of the Function (f1)'s code

```
int f1(int a, int b){
        int buffer1[32] ;

        codes for the function ….
        ……………………..

}
```

How to call a function `f1(arg1, arg2)` ?

(1) push arg1, arg2 (i.e. a and b) into the Stack (before calling)

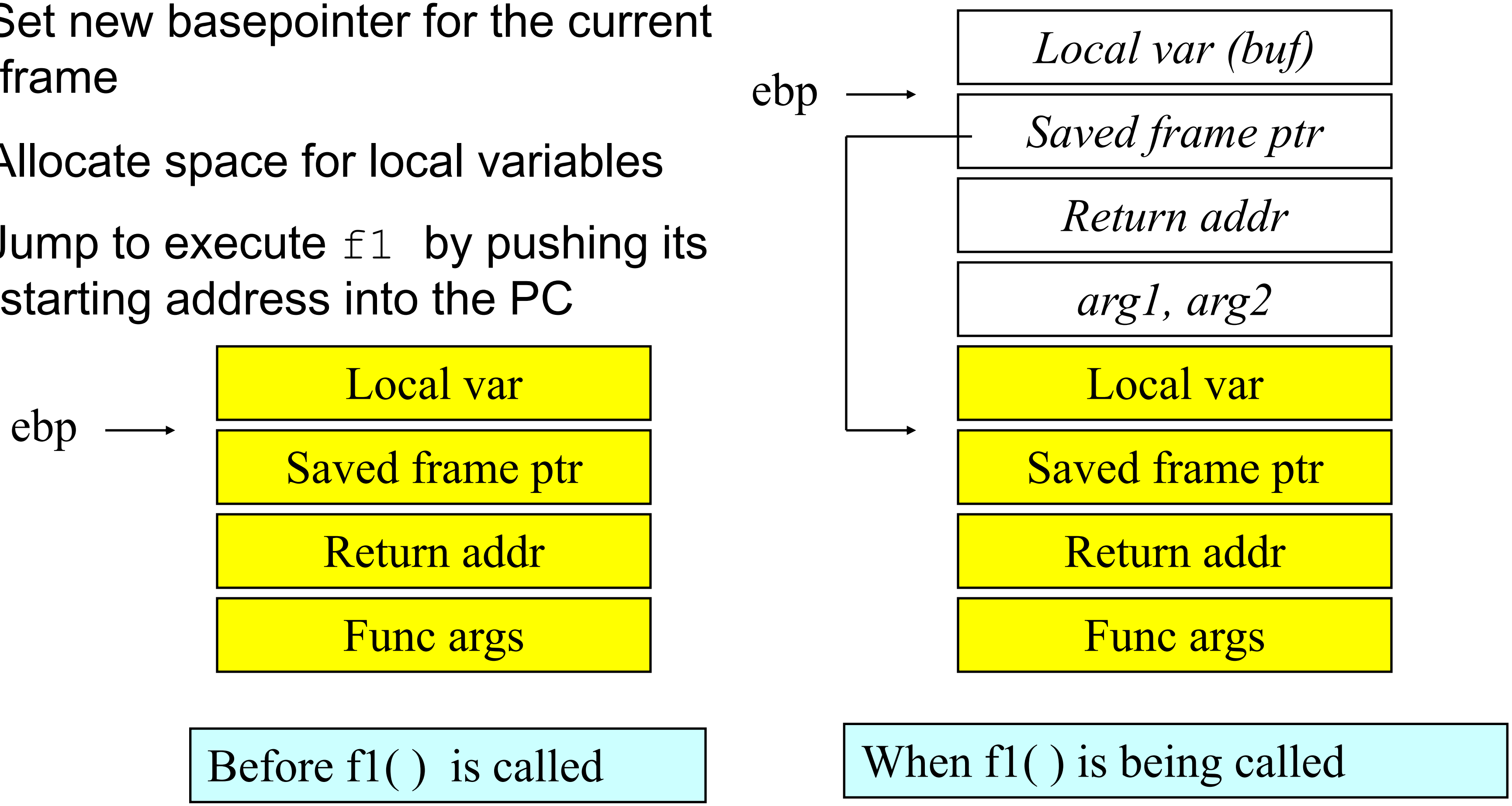
(2) Push return addr and

old basepointer (ebp) to stack

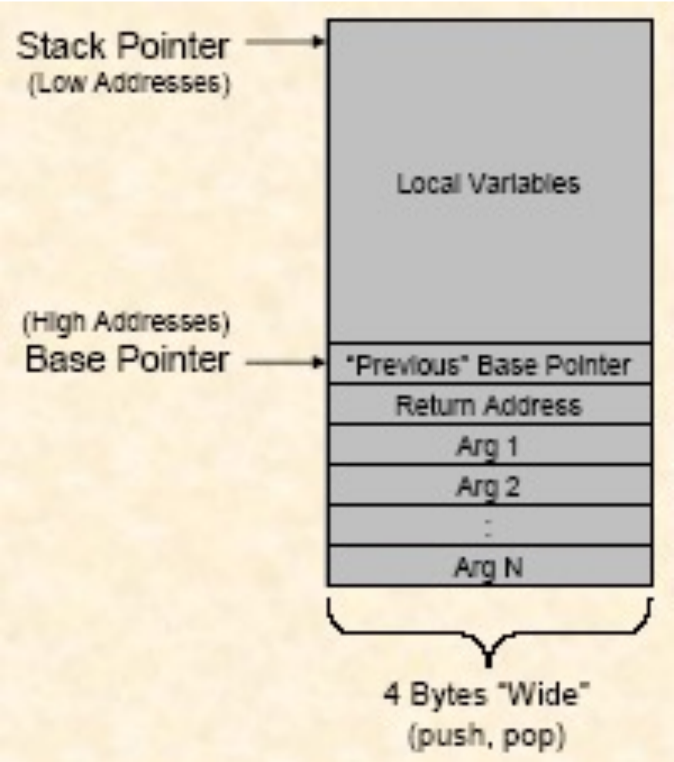
(3) Set new basepointer for the current frame

(4) Allocate space for local variables

(5) Jump to execute `f1` by pushing its starting address into the PC

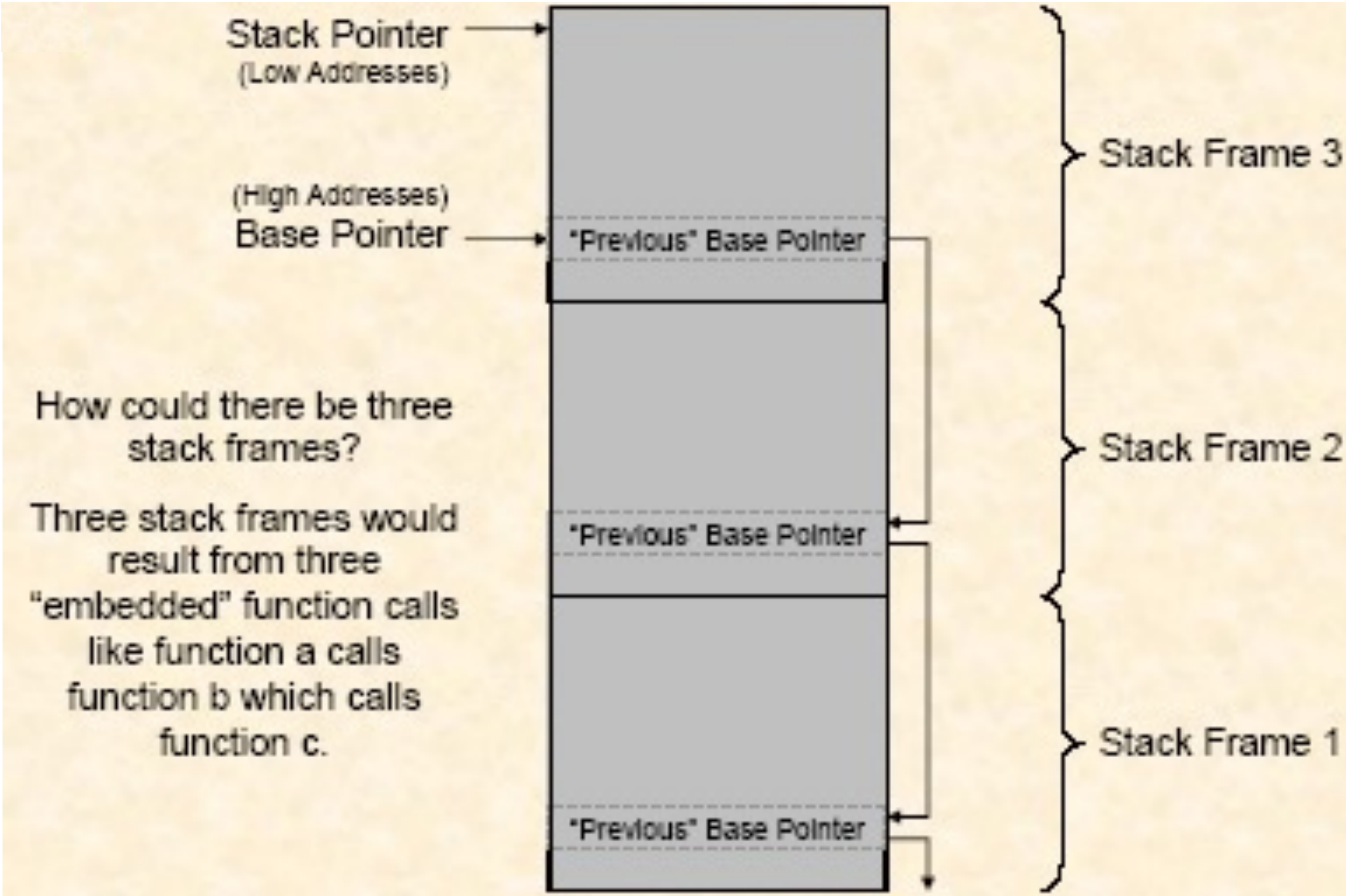


Stack Basics



- A stack consists of logical stack frames that are pushed when calling a function and popped when returning.
  - Base (frame) pointer – points to a fixed location within a frame.
- When a function is called, the function arguments, the return address, stack frame pointer and the variables are pushed on the stack (in that order).
- So the return address has a higher address than the Local Variables buffer section.
- When we overflow the buffer (of the Local variables section), the return address will be overwritten.

How does the Stack look like after several Nested Function Calls ?

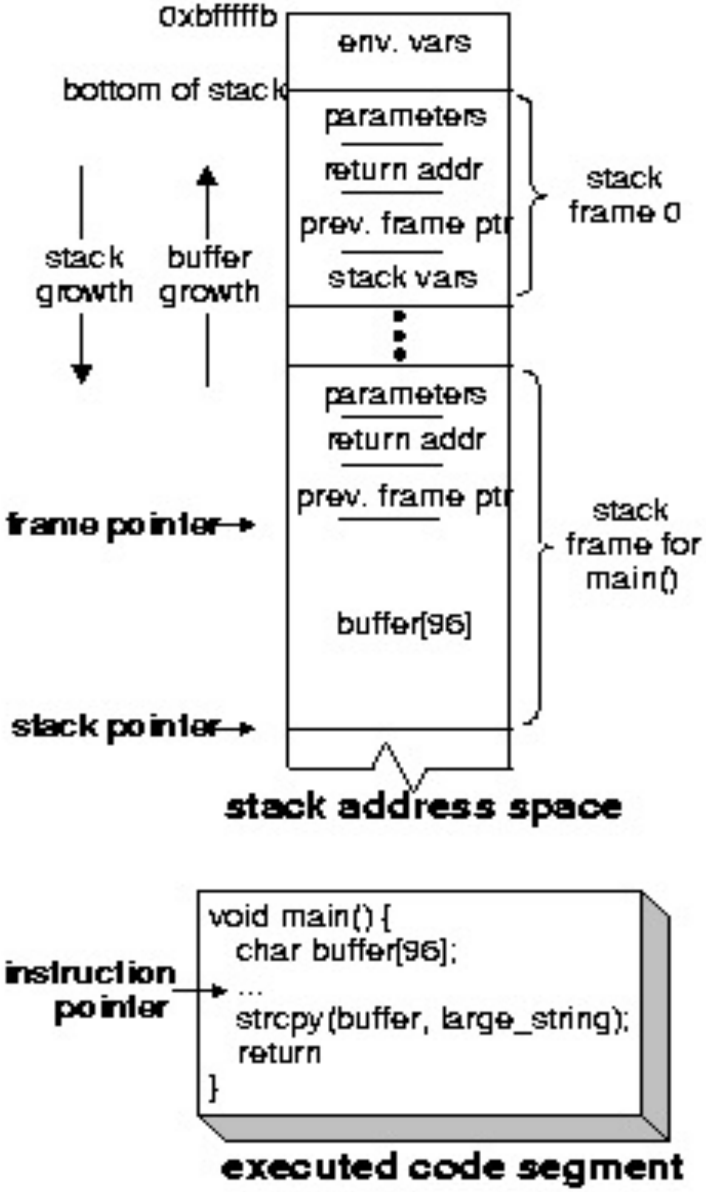


Two Required Tasks to realize a Security Attack (i.e. a system break-in)

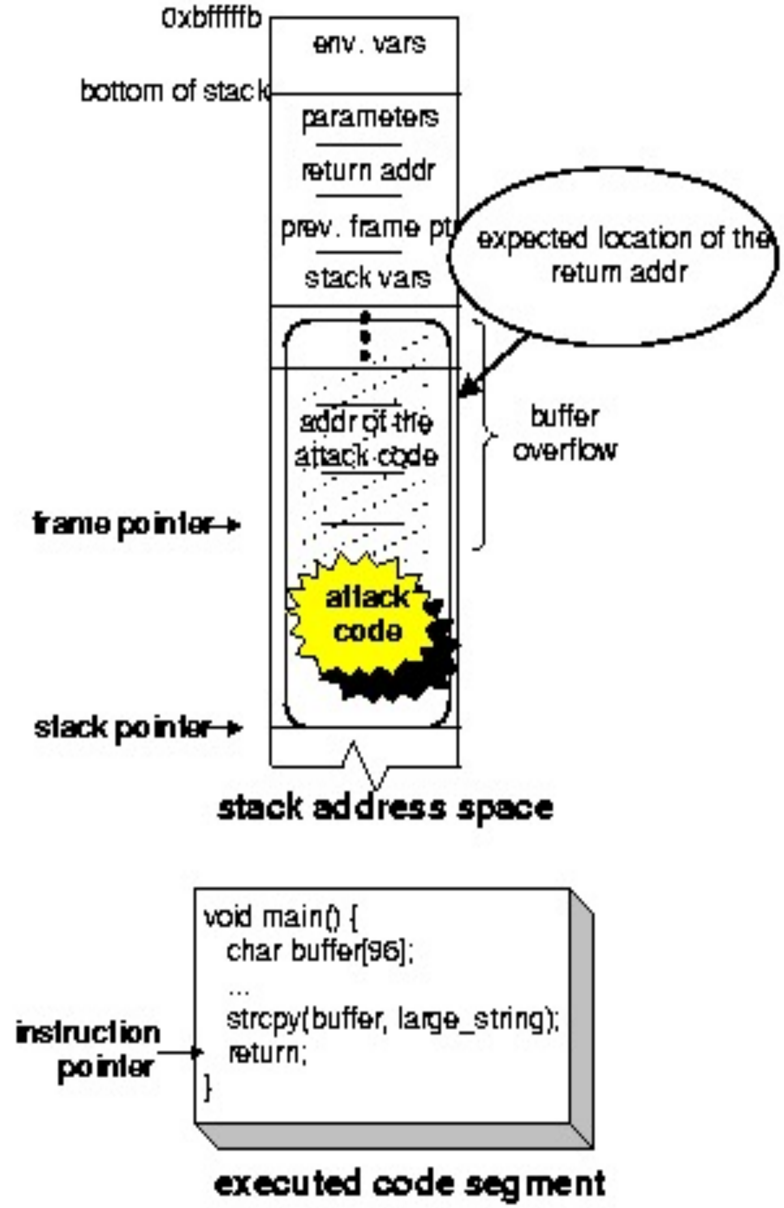
1. Inject the attack code, which is typically a small sequence of instructions that do bad things (e.g. spawns a command shell into a running process.)
2. Change the execution path of the running process to execute the attack code.

- Overflowing the Stack buffer can achieve both objectives simultaneously.

What happens when the Buffer is overflown ?



(i) Before the attack



(ii) after injecting the attack code

How can we place arbitrary instruction into its address space?

- First, place the ”shellcode” that you are trying to execute in the buffer we are overflowing
- Then, overwrite the return address so it points back into the ”shellcode” in the buffer.

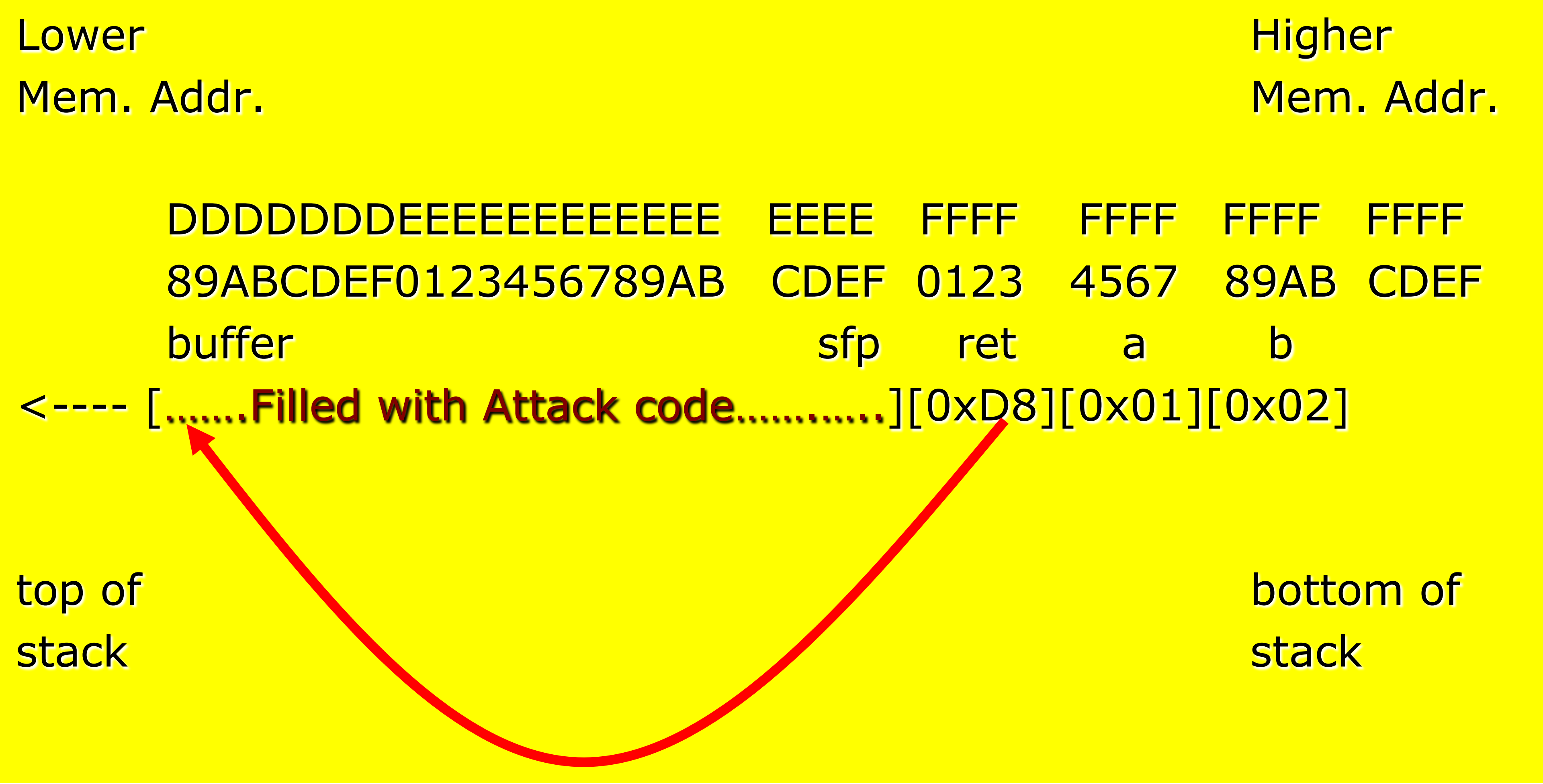
Stack layout for a Vulnerable Progam, e.g. main( ) in the previous slide

```
   Lower                                                  Higher
 Mem. addr.                                             Mem. Addr.

              buffer2       buffer1   sfp ret    a     b     c
 <------ [              ][          ][   ][   ][   ][   ][   ]

Top of stack                                        Bottom of
                                                        stack
```

After Buffer-flow, the stack looks like:



Sample Shellcode.c

```
#include<stdio.h>
void main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

vulnerable.c

```
void main(int argc, char *argv[ ]) {
    char buffer[512];
    if (argc > 1)
        strcpy(buffer,argv[1]);
}
```

How to defend against Buffer Overflow

- Write Secure Code
  - Instead of using “dangerous” functions/system calls, e.g. scanf( ), strcpy( ), strcat( ), getwd( ), gets( ), strcmp( ), sprintf( ). , use their “safe” counterparts: strncpy, strncmp etc.
- Perform Security-Focused Code Review
- Use other security checking-tools which will guard against array-boundary-overflow at run-time
- Operating Systems to Support of Non-executable-Stack Features, e.g. Windows’ Data Execution Protection (DEP) mode
  - Attackers respond by inventing Return-to-libc and
    - Defenders respond with Address Space Layout Randomization (ASLR)…
  - Hackers then invent Return-Oriented-Programming (ROP) attacks as well as side-channel attacks to circumvent ASLR

As of today, there is still an Ongoing Arm Race between the attackers and defenders

Additional References

- http://insecure.org/stf/smashstack.html
- http://www.ece.cmu.edu/%7Eadrian/630-f04/readings/cowan-vulnerability.pdf

Backup Slides

Defenses against Stack Buffer Overflow

Source: Profs. Dan Boneh, John Mitchell
Stanford University

Preventing Buffer Overflow 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-executable

- Prevent overflow code execution by marking stack and heap segments as **non-executable**
  - NX-bit on AMD Athlon 64, XD-bit on Intel P4
    - NX bit in every Page Table Entry (PTE)
  - Deployment:

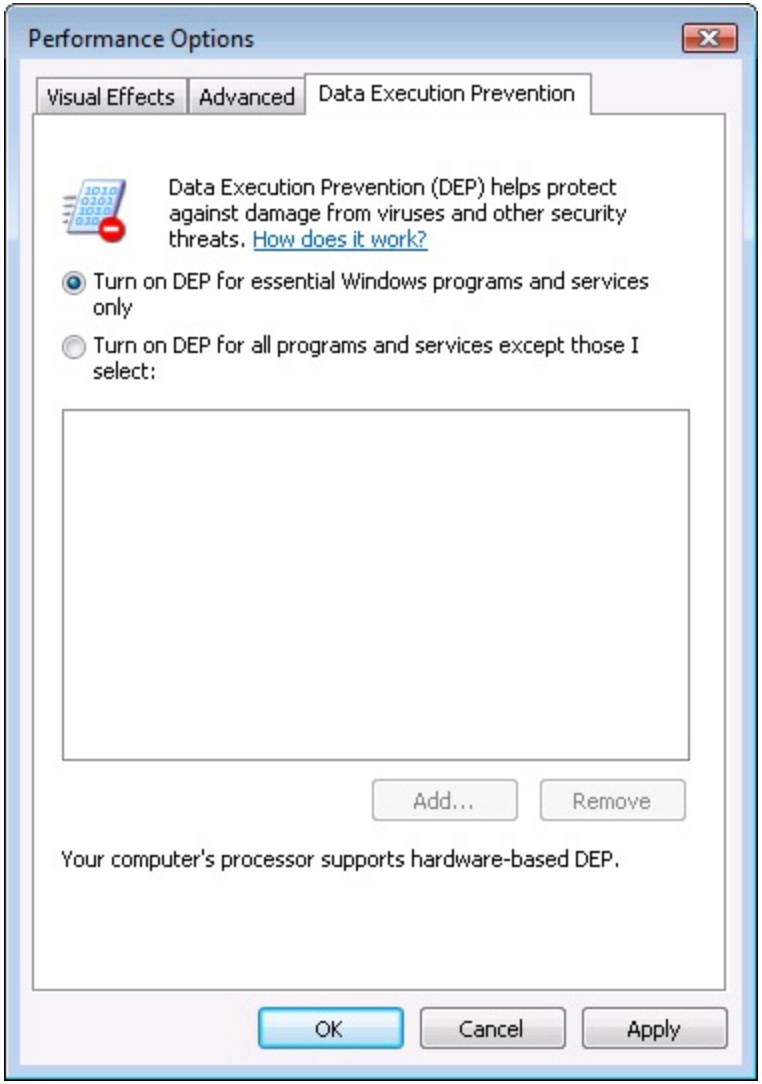
    Linux (via PaX project); OpenBSD

    Windows since XP SP2 (DEP = Data Execution Prevention)

    ✦ Boot.ini : **/noexecute=OptIn** or **AlwaysOn**

- Limitations:
  - Some apps need executable heap (e.g. JIT = Just-In-Time compiler).
  - Does not defend against \`**return-to-libc**’ exploit

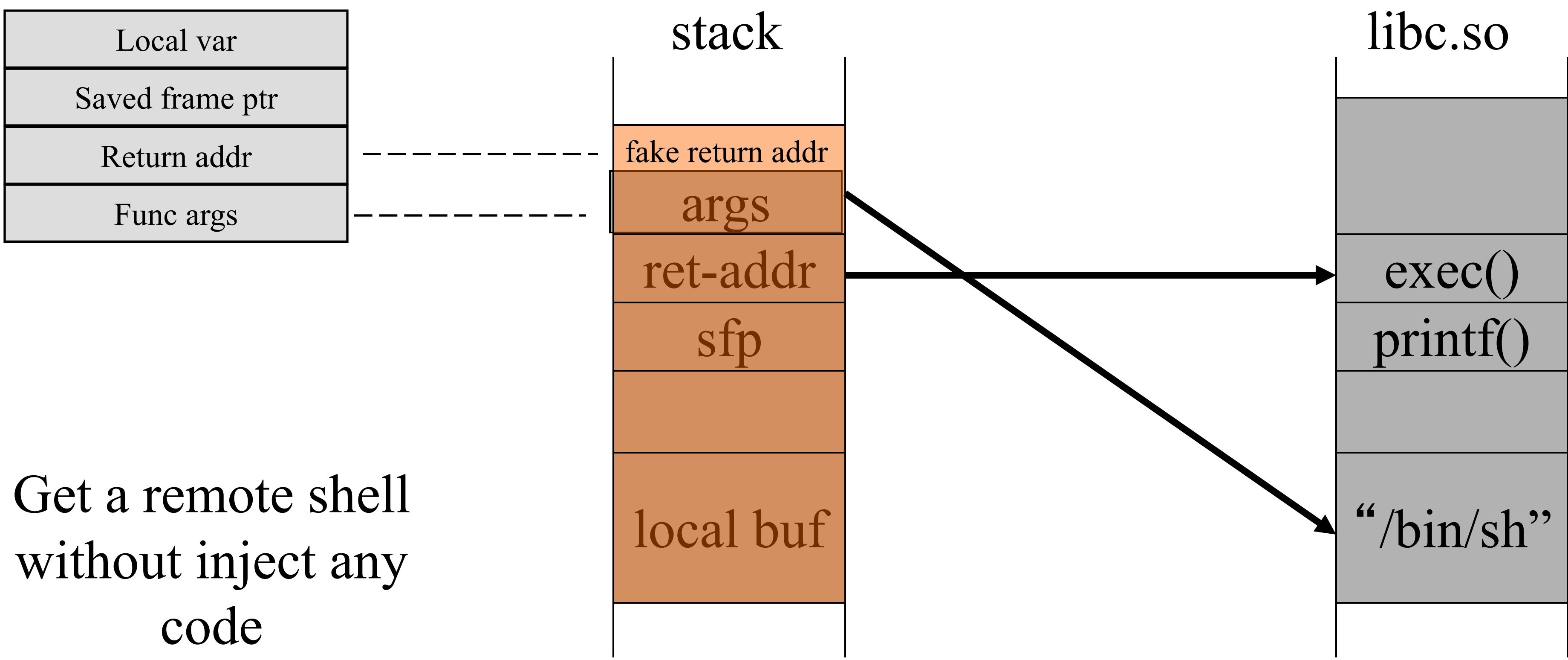
Examples: DEP controls in Windows OSes



DEP terminating a program

Return-to-libc Attacks

- Control hijacking without executing code



Get a remote shell without inject any 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:
    - Windows Vista: 8 bits of randomness for DLLs
      - aligned to 64K page in a 16MB region ⇒ 256 choices
    - **Linux** (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures
- Other randomization methods:
  - Sys-call randomization: randomize sys-call id’s

ASLR Example

Booting Windows 7 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

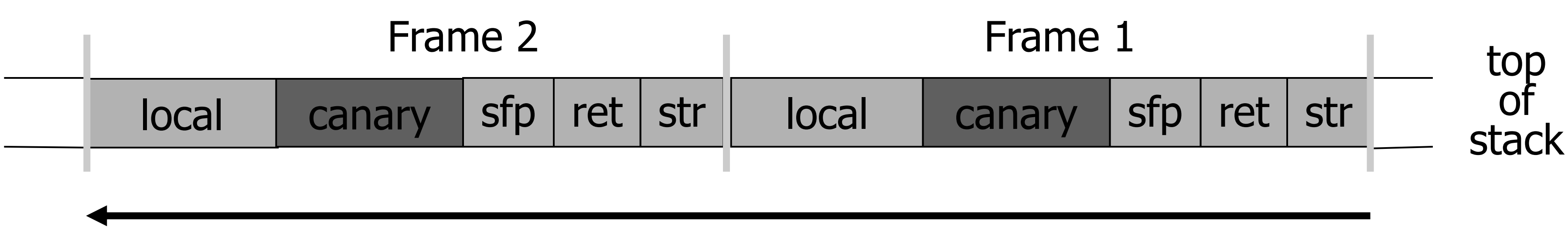
Note: ASLR is only applied to images for which the dynamic-relocation flag is set

Run time checking: StackGuard

- Many 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.

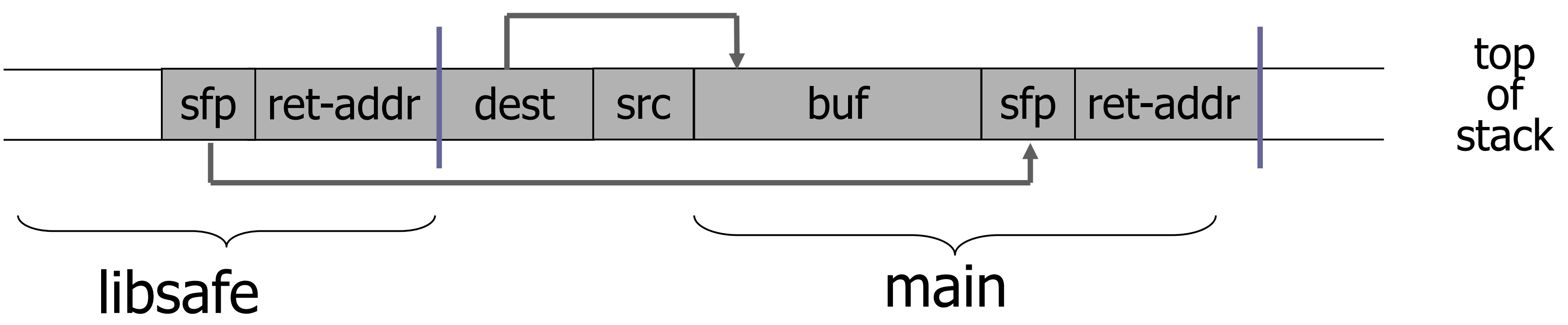
Real-world Examples taken similar approach:

- ProPolice (IBM) - gcc 3.4.1. (the **-fstack-protector** option)
- MS Visual Stdio compiler: the /GS option



Run time checking: 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:
    - If so, does strcpy,
      otherwise, terminates application



Other Buffer Overflow Attacks

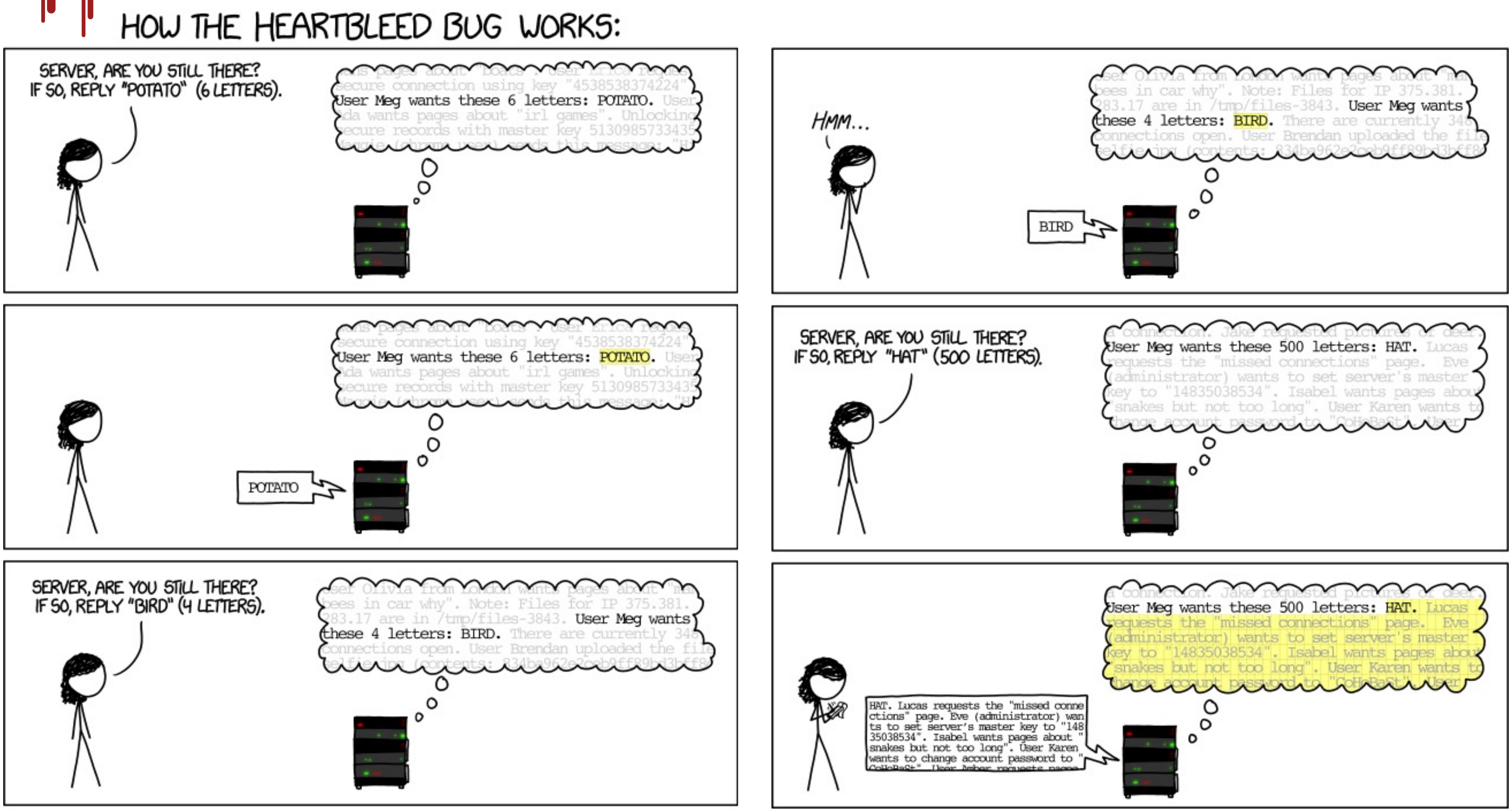
Integer Overflow attacks

- <u>Integer overflows</u>: (e.g. MS DirectX MIDI Lib) Phrack60

```
void func(int a, char v) {
  char buf[128];
  init(buf);
  buf[a] = v;
}
```

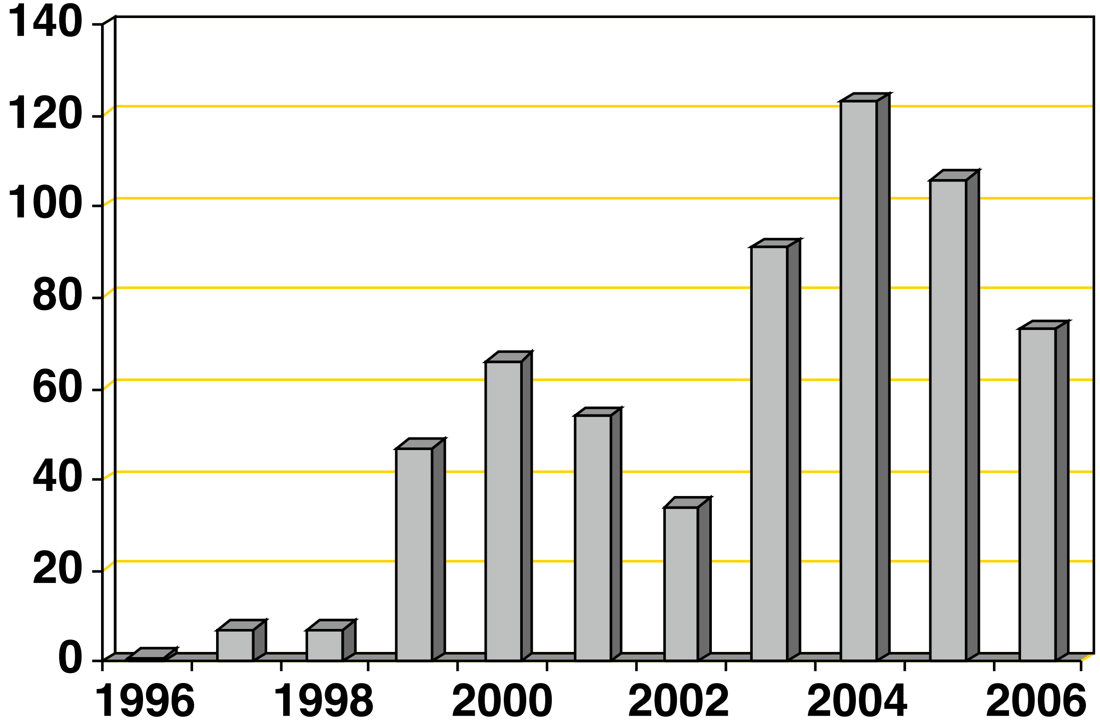
  - Problem: Attacker can make `a` point to `ret-addr' on stack and then overwrite it with input `v`.

Sometimes, you don't really need to "Overflow" the buffer to launch an Overflow Attack



Source: https://xkcd.com/1354

Integer overflow stats



Source: NVD/CVE

Format String Vulnerabilities

```
int func(char *user)  {
  fprintf( stdout, 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” (Writes the number of characters into a pointer)

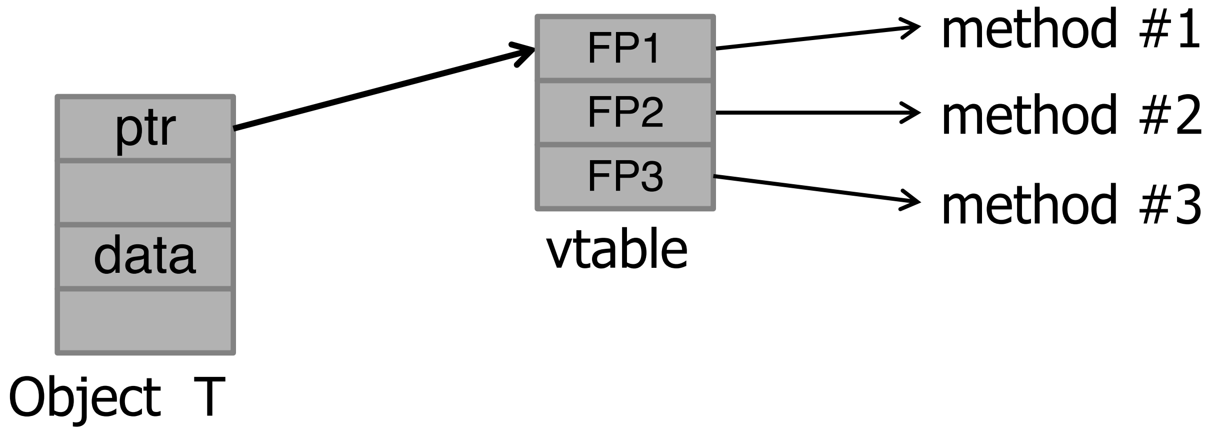
Correct form:

```
int func(char *user)  {
  fprintf( stdout, "%s", user);
}
```

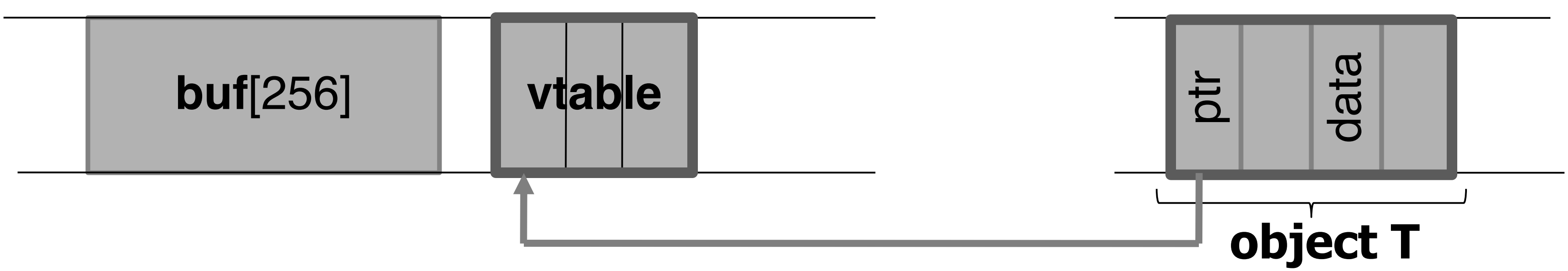
`See:` http://julianor.tripod.com/bc/formatstring-1.2.pdf for details

Heap Overflow

- Buffer can also appear in heap area, like: buff=(char*) malloc(256)
- Heap can be overflowed, just like stacks
- One attack on compiler generated function pointers (e.g. C++ code)

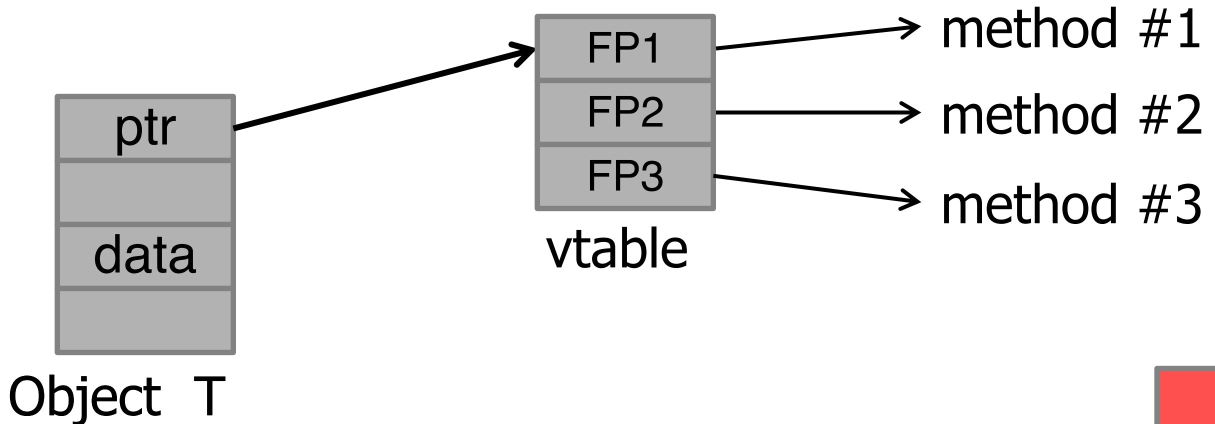


- Suppose vtable is on the heap next to a string object:

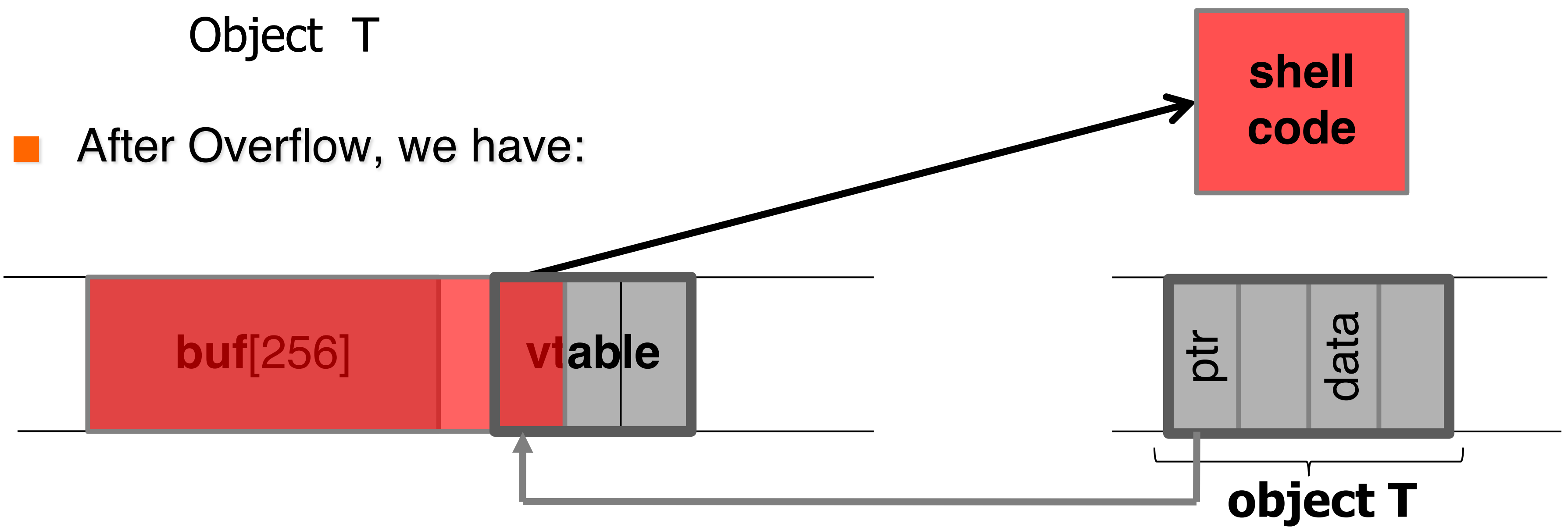


Heap Overflow (cont'd)

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



- After Overflow, we have:

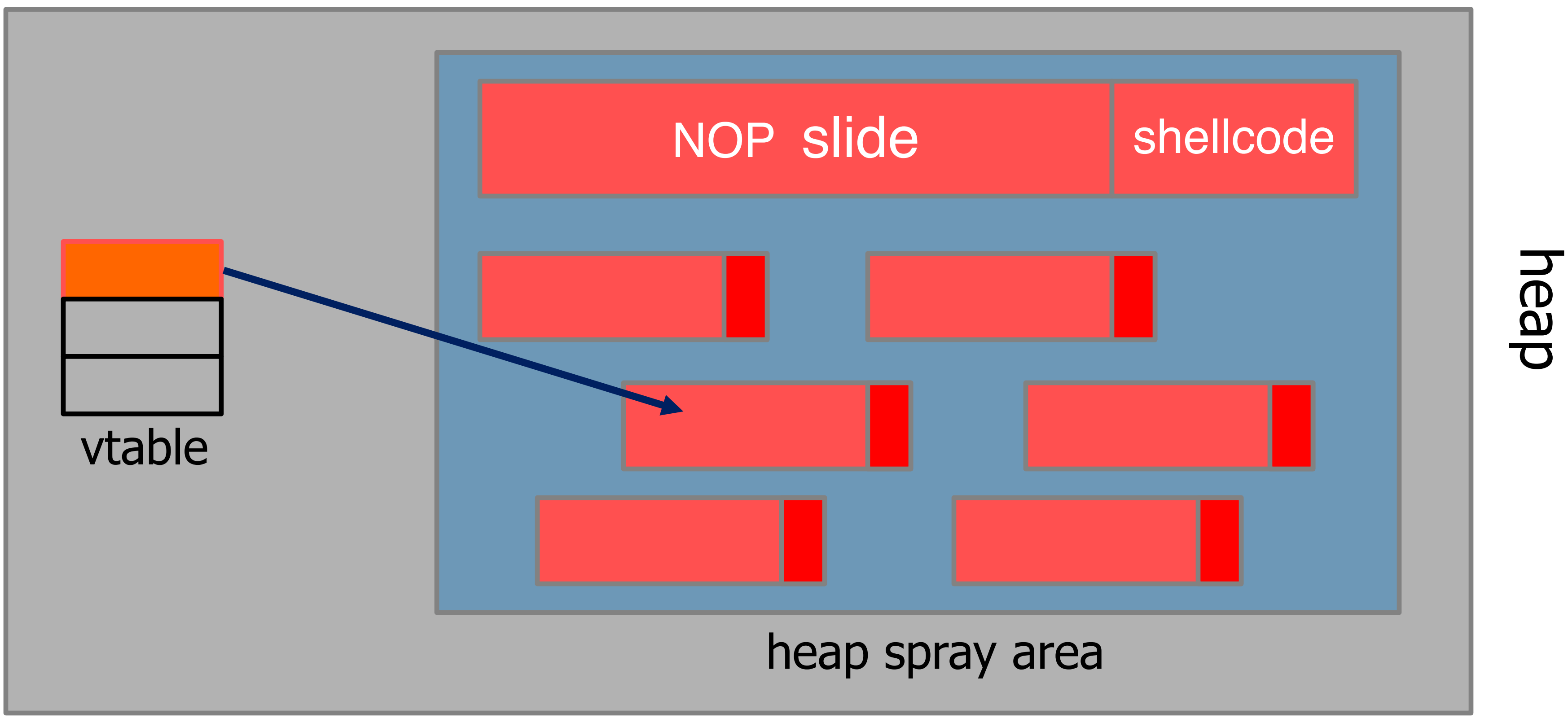


Heap Spraying

[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.

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

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