Control Hijacking

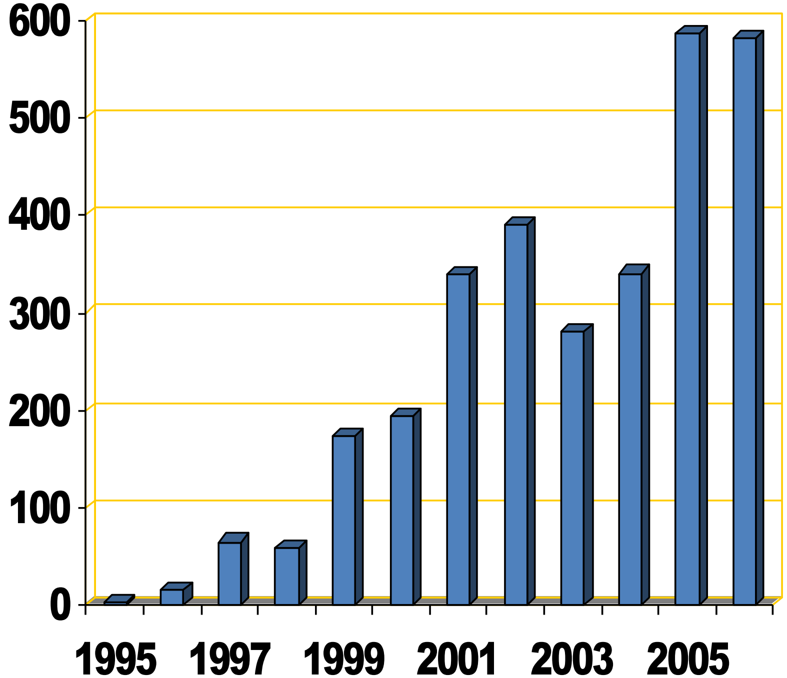
Basic Control Hijacking Attacks

Control hijacking attacks

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

2005-2007: ≈ 10%

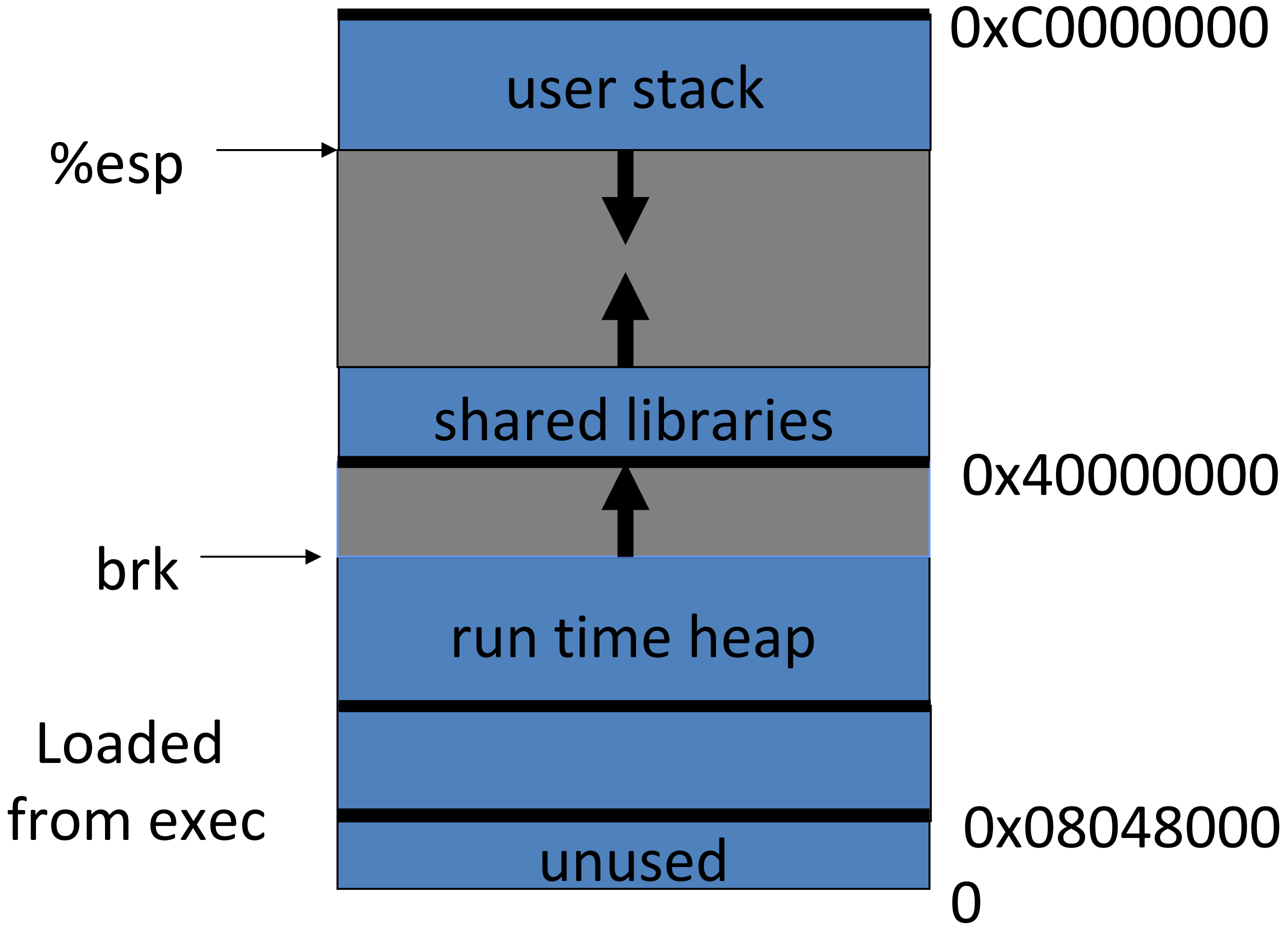
Source: NVD/CVE

What is needed

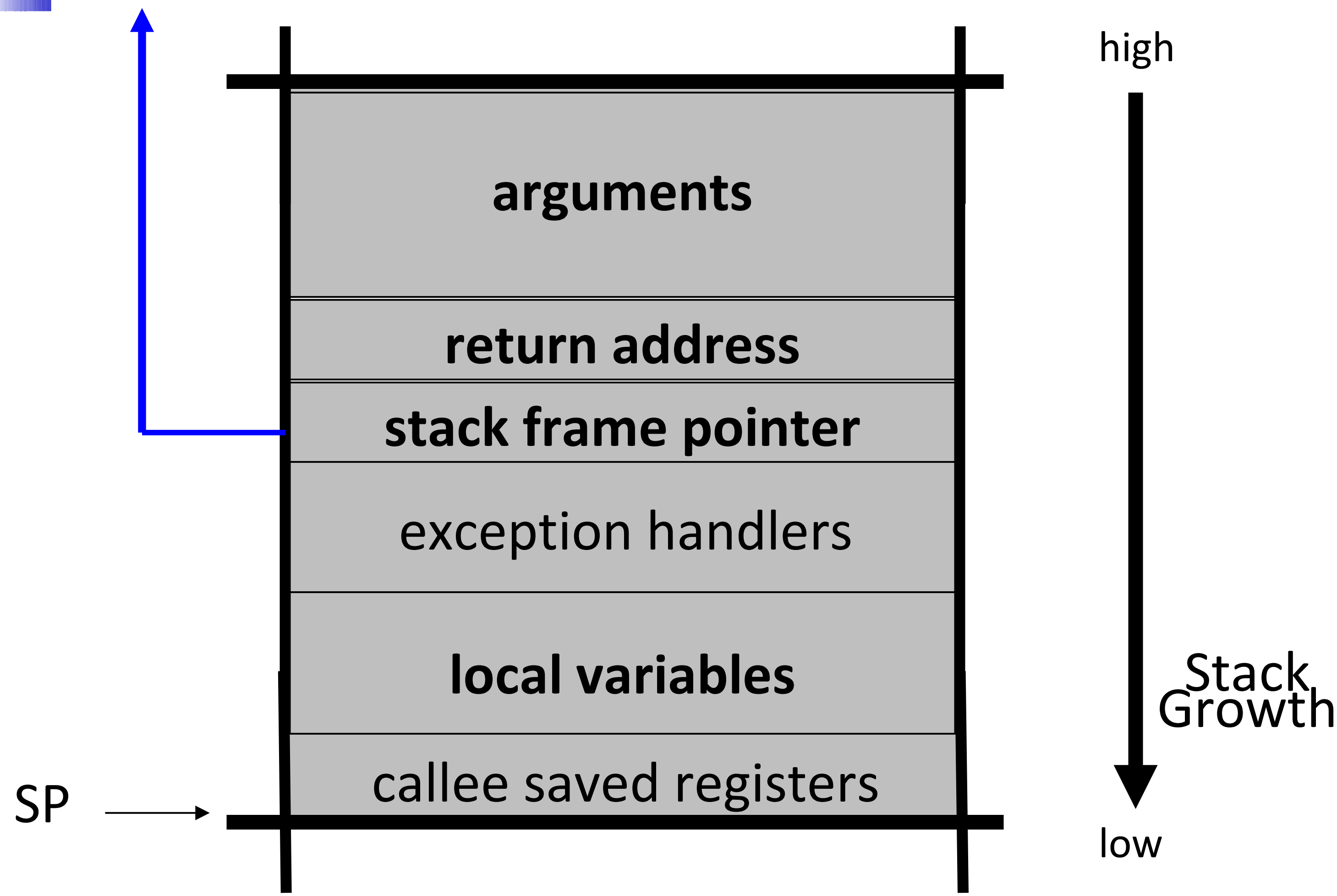
- Understanding C functions, the stack, and the heap.
- Know how system calls are made

- The exec() system call
- Attacker needs to know 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

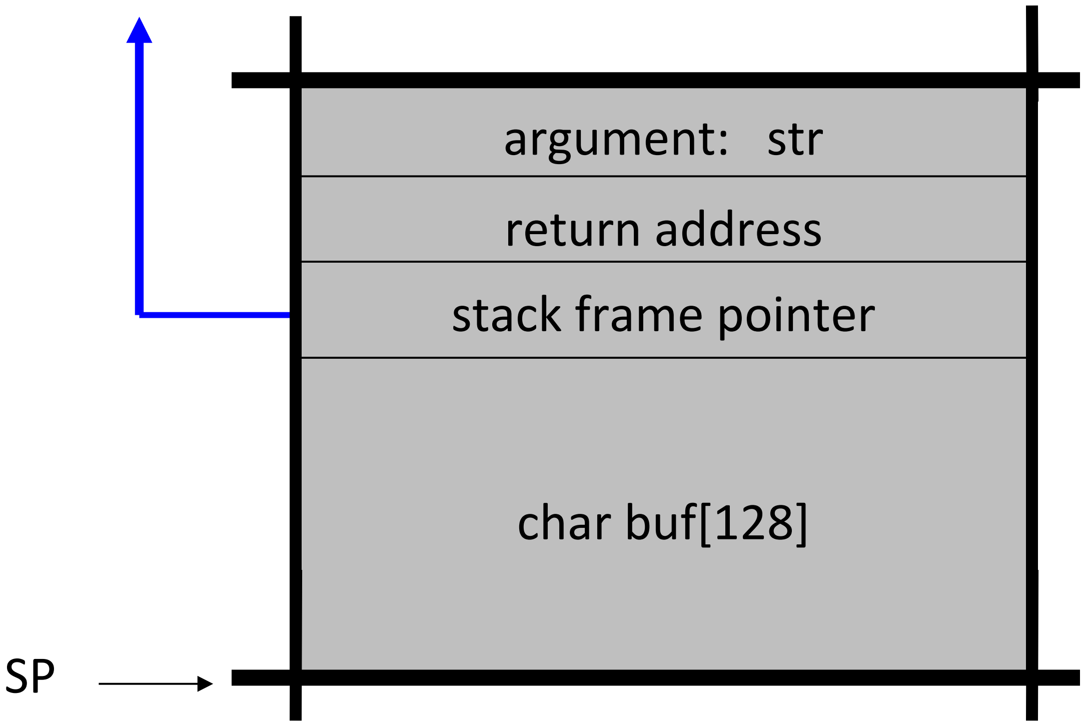

high
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
SP
Stack
Growth
low

What are buffer overflows?

Suppose a web server contains a function:

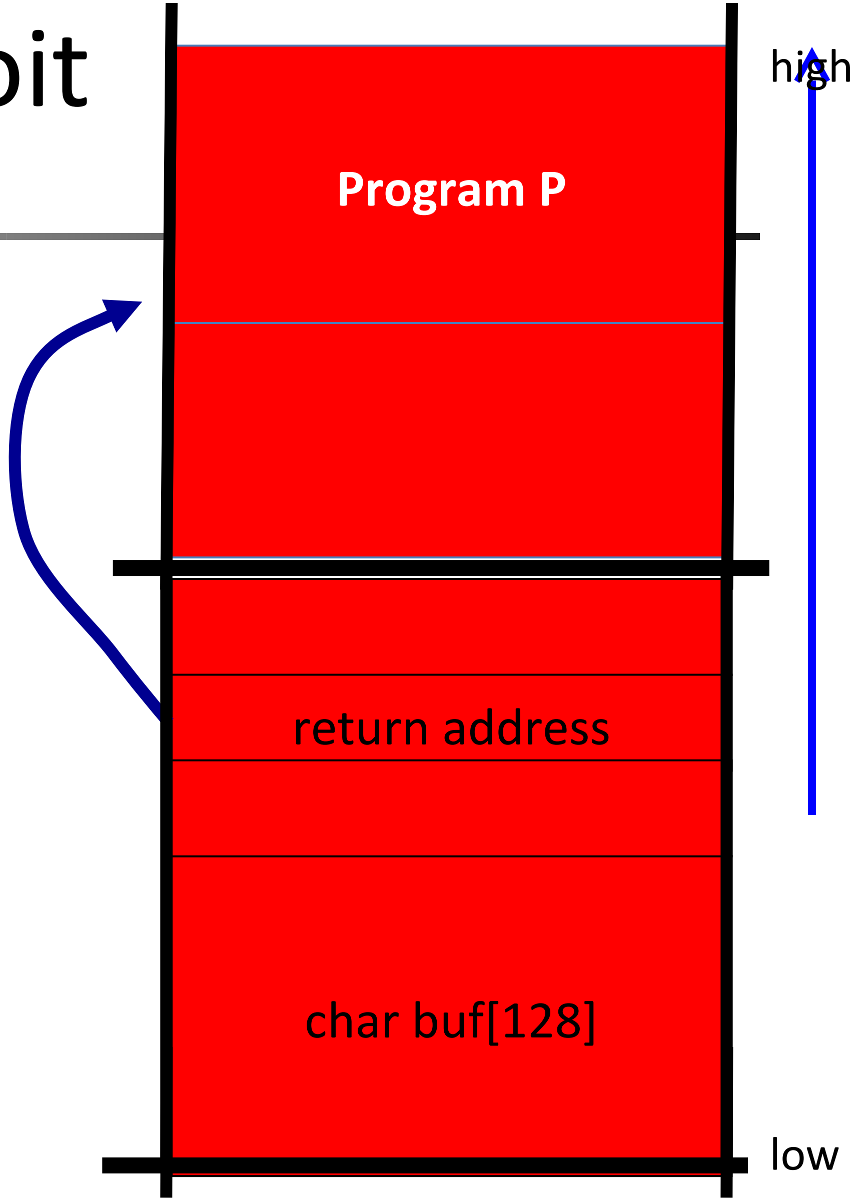
```
void func(char *str) {
    char buf[128];

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

When func() is called stack looks like:



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*.



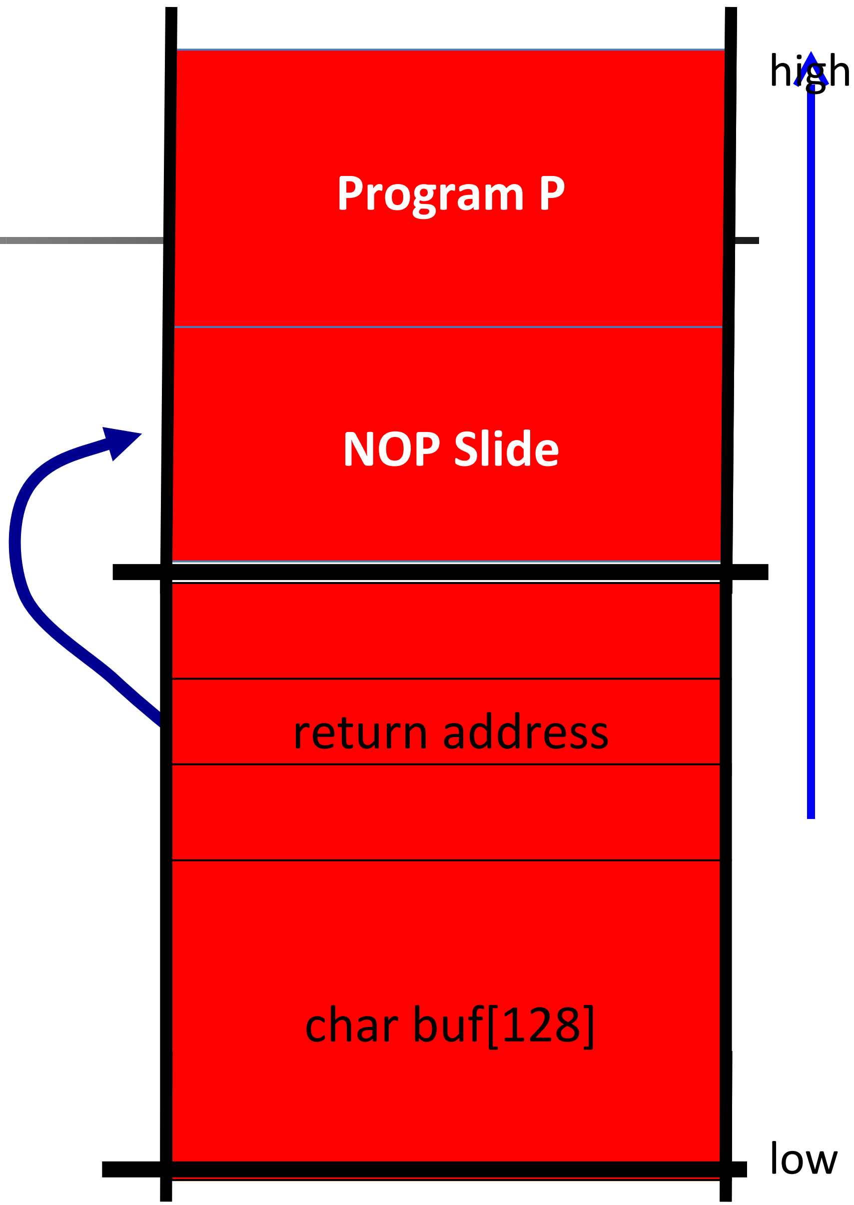
The NOP slide

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.
- Sample <u>remote</u> stack smashing overflows:
  - (2007) Windows animated cursors (ANI), LoadAniIcon()
  - (2005) Symantec Virus Detection

Many unsafe libc functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

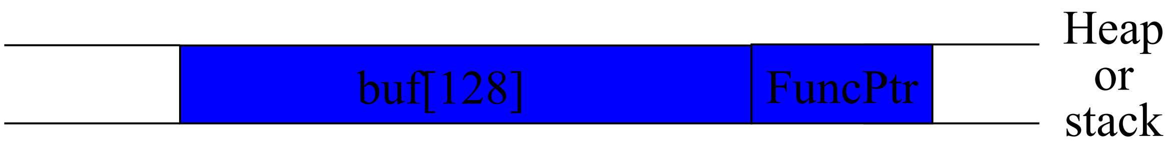
scanf (const char *format, …)   and many more.

- "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 an exception handler address in stack frame.



- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)
  - Overflowing buf will override function pointer.

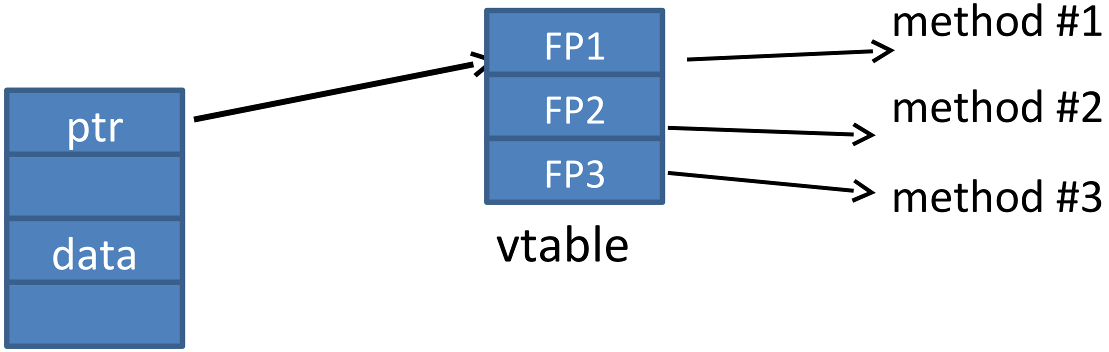
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

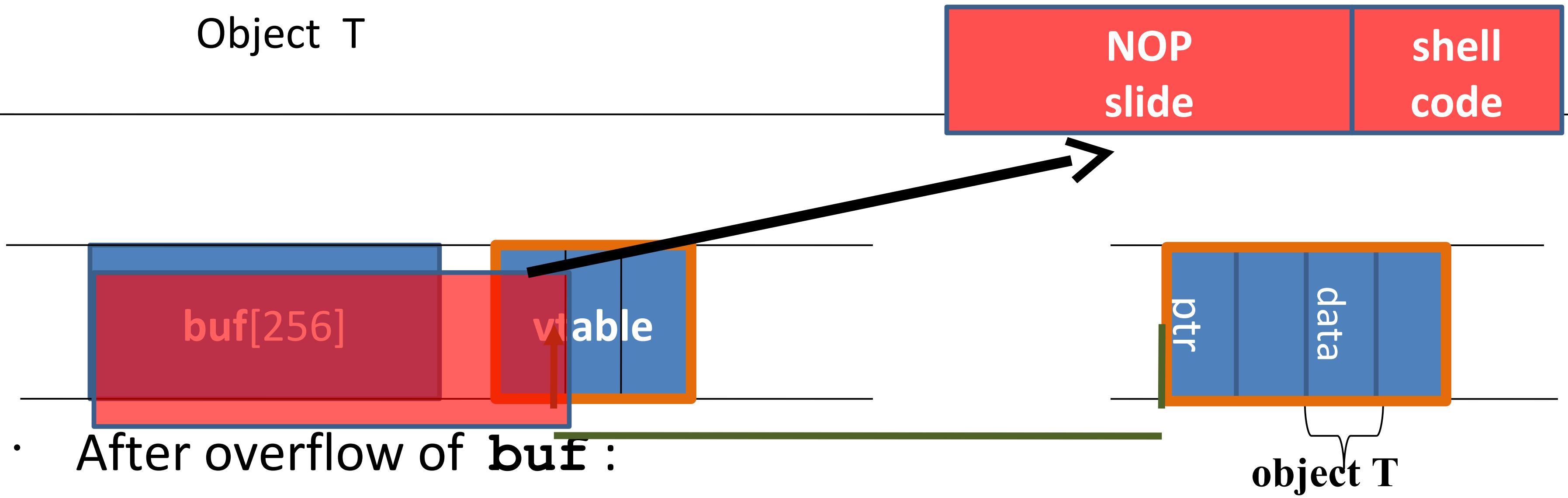
SEH attack

- It executes arbitrary code by abusing the 32-bit Windows exception dispatching facilities
- A stack-overflow overwrites an exception registration record (ERR) on a thread's stack.
- An ERR includes a next pointer and an exception handler function pointer. The next pointer links to the next record in the list of registered exception handlers. The exception handler function pointer is used when an exception occurs.
- After an exception registration record has been overwritten, an exception must be raised so that the exception dispatcher will attempt to handle it.

Corrupting method pointers

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





- After overflow of **buf** :

Finding buffer overflows

- To find overflow in a web server:
  - Run 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)

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**

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 = 0xffffff80 + 0x80 $\Rightarrow$ m = 0

Can this be exploited?

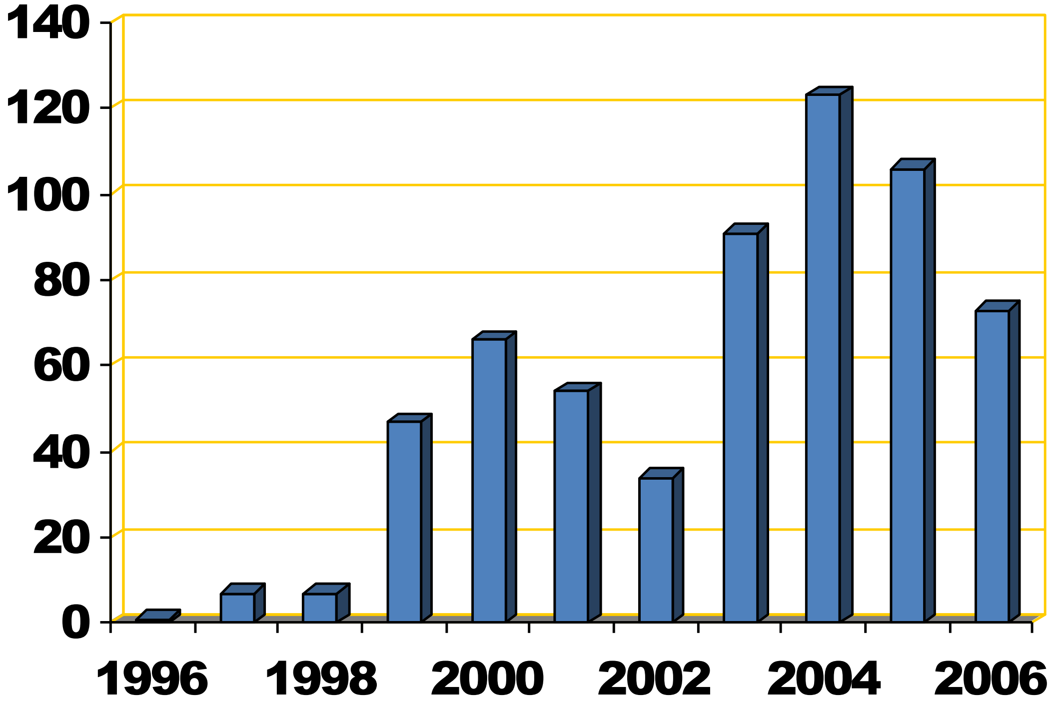
An example

```
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
}
```

If **len1 = 0x80,** **len2 = 0xffffff80** ⇒ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer overflow exploit stats



Source: NVD/CVE

Format string bugs

Format string problem

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

```
int fprintf(FILE *stream, char *formato,
argomenti ...);
```

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);`

Format string problem

Se si passa a una funzione che stampa una stringa a schermo (printf del C) una stringa che in realtà contiene una serie di parametri di specifica dell'input (tipicamente %s e %x per esaminare il contenuto della memoria e %n per sovrascriverne parti , in particolare dello stack) si permette l'avvio di un attacco di tipo stack overflow e return to libc.

Per proteggersi da questo attacco, quando si vuole stampare una stringa s usando la printf() o una qualsiasi funzione C che accetti un numero illimitato di identificatori di formato, bisogna scrivere la funzione printf("%s", s) e non printf(s)

History

- First exploit discovered in June 2000.
- Examples:
  - wu-ftpd 2.* : remote root
  - Linux rpc.statd: remote root
  - IRIX telnetd: remote root
  - BSD chpass: local root

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) - ‘6’ into temp.
  - printf(“%08x.%08x.%08x.%08x.%n”)

Control Hijacking

Platform Defenses

=

Contromisure

Preventing hijacking attacks

a) Fix bugs:
- Audit software
  - Automated tools: Coverity, Prefast/Prefix.
- Rewrite software in a type safe languange (Java, ML)
  - Difficult for existing (legacy) code …

b) Concede overflow, but prevent code execution

c) Add runtime code to detect overflows exploits
- Halt process when overflow exploit detected
- StackGuard, LibSafe, …

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);

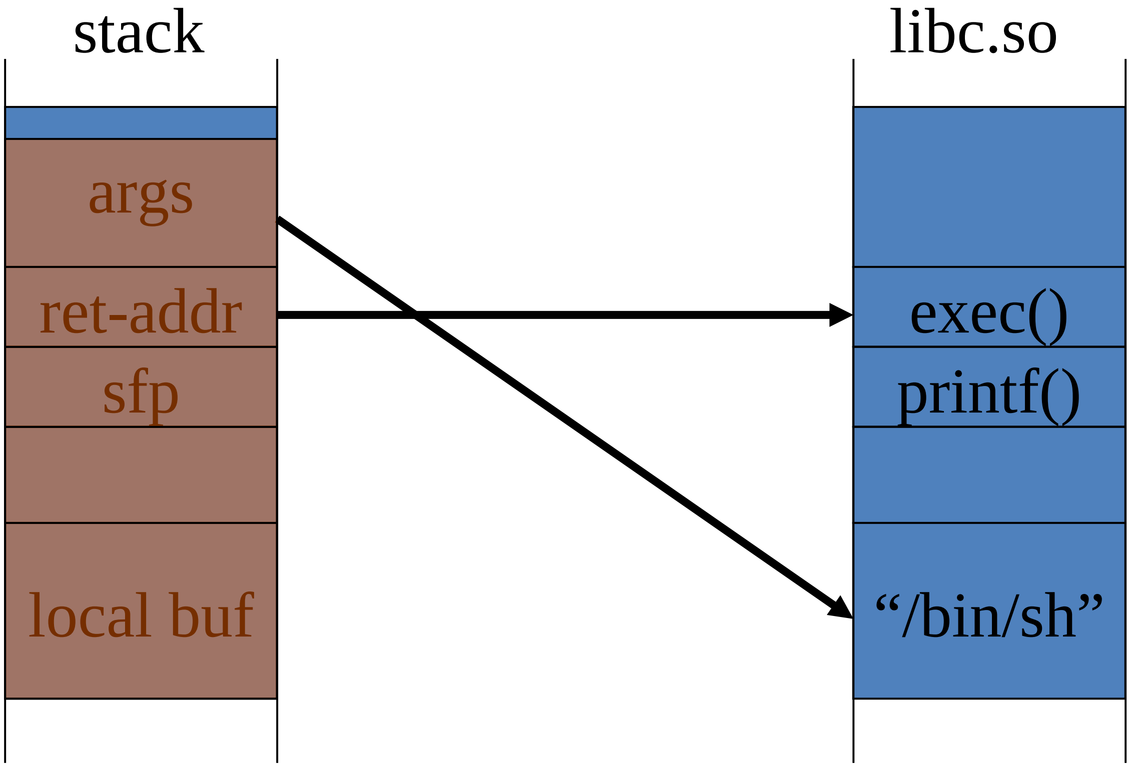
- OpenBSDWindows: since XP SP2 (DEP)
- Visual Studio: **/NXCompat[:NO]**

Limitations:

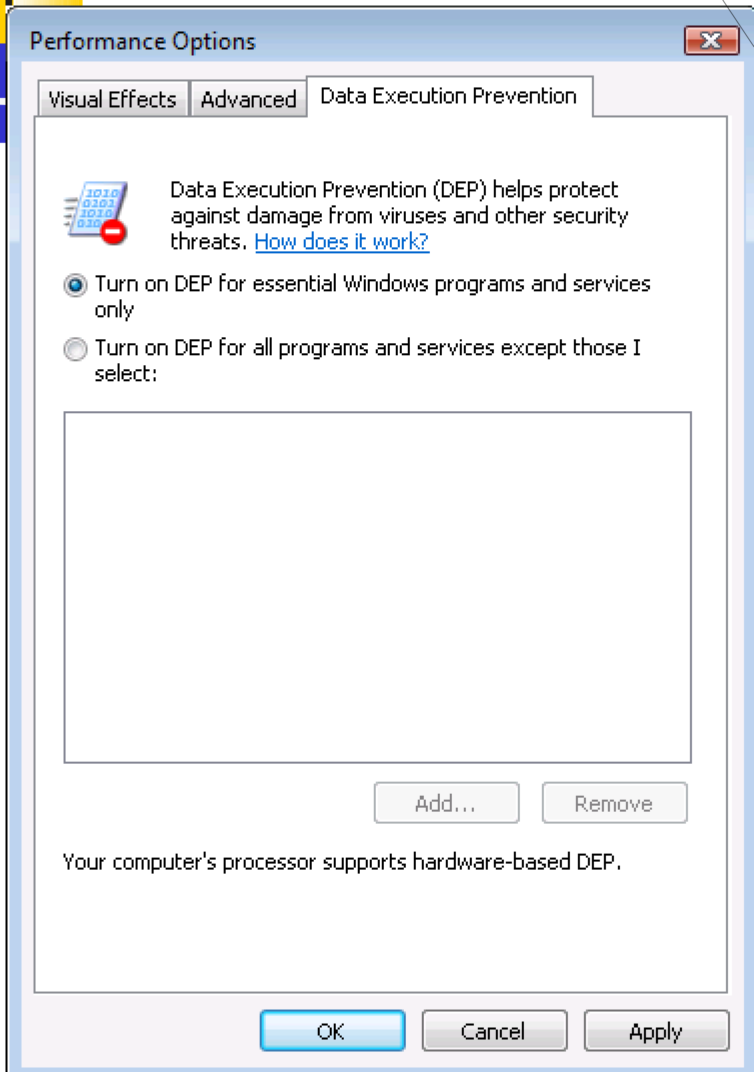
- Some apps need executable heap (e.g. JITs).
- Does not defend against `**Return Oriented Programming**'

Attack: Return Oriented Programming (ROP)

- Control hijacking without executing code



Examples: DEP controls in Windows



DEP terminating a program

Response: ASLR = Address space layout randomization

- Shared libraries to random location in process memory

$\Rightarrow$ 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 $\Rightarrow$ 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

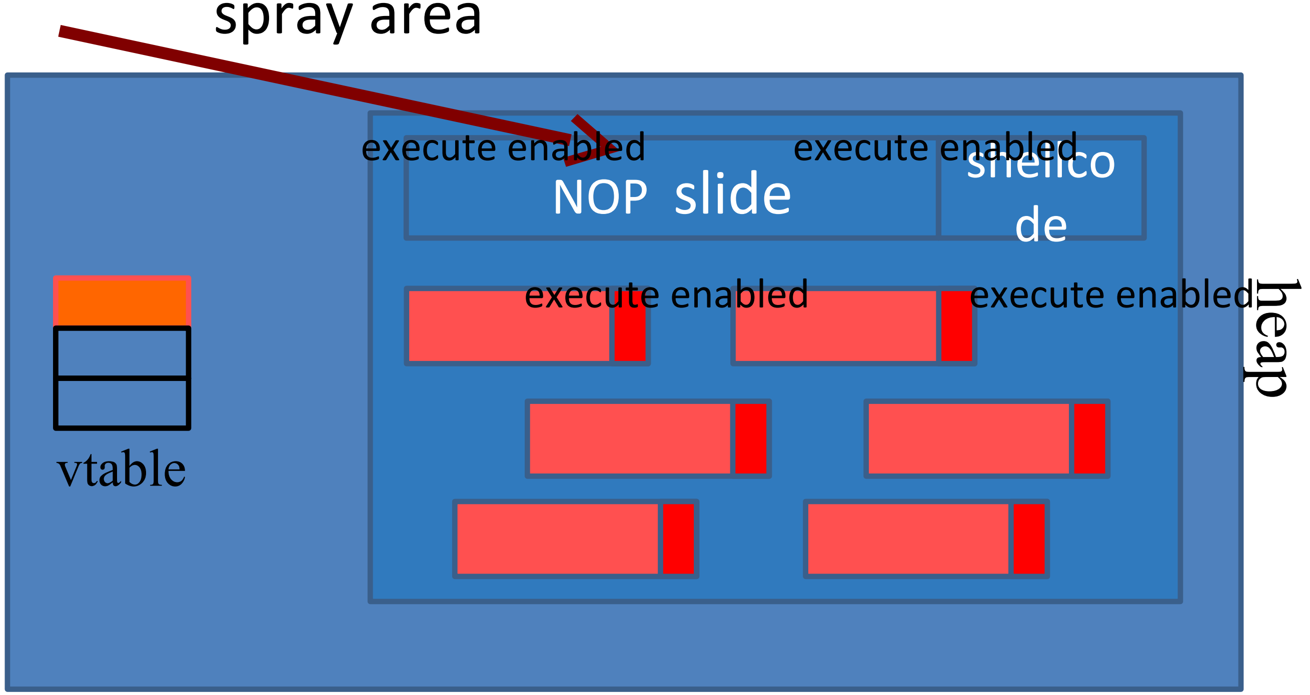
- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

More attacks : JiT spraying

Idea:

1. Force Javascript JiT to fill heap with executable shellcode

2. then point Saved Frame Pointer anywhere in spray area



More attacks : JiT spraying

Most modern interpreters implement a Just-In-Time (JIT) compiler to transform the parsed input or bytecode into machine code for faster execution.

JIT spraying is the process of coercing the JIT engine to write many executable pages with embedded shellcode.
This shellcode will entered through the middle of a normal JIT instruction.

For example, a Javascript statement such as "var x = 0x41414141 + 0x42424242;" might be compiled to contain two 4 byte constants in the executable image
("mov eax, 0x41414141; mov ecx, 0x42424242; add eax, ecx").
By starting execution in the middle of these constants, a completely different instruction stream is revealed.

Control Hijacking

Run-time Defenses

StackGuard

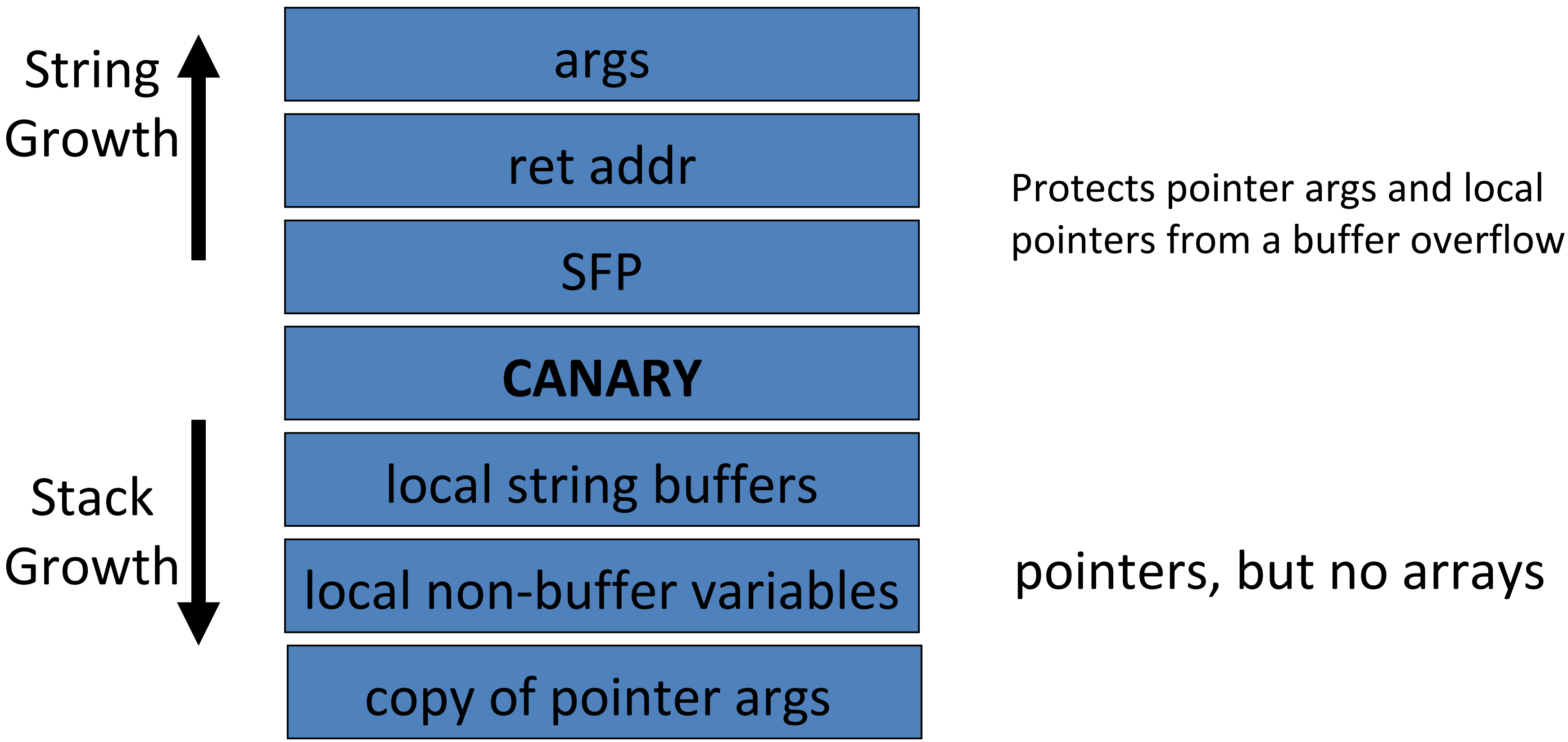
- Minimal performance effects: 8% for Apache.
- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
- Note: Canaries don't provide full proof protection.
  - Some attacks leave canaries unchanged
- Heap protection: PointGuard.
  - Protects pointers and buffers by encryption
  - Less effective, more noticeable performance effects

Heap protection: PointGuard.

- Protects pointers and buffers by encryption
- Key generated when the program starts
- Never shared so it is secure
- Less effective, more noticeable performance effects

StackGuard enhancements: ProPolice IBM

Rearrange stack layout to prevent ptr overflow.



ProPolice IBM

- reorder local variables to place buffers after pointers to avoid the corruption of pointers
- copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers
- omission of instrumentation code from some functions to decrease the performance overhead.

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:

```
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
```

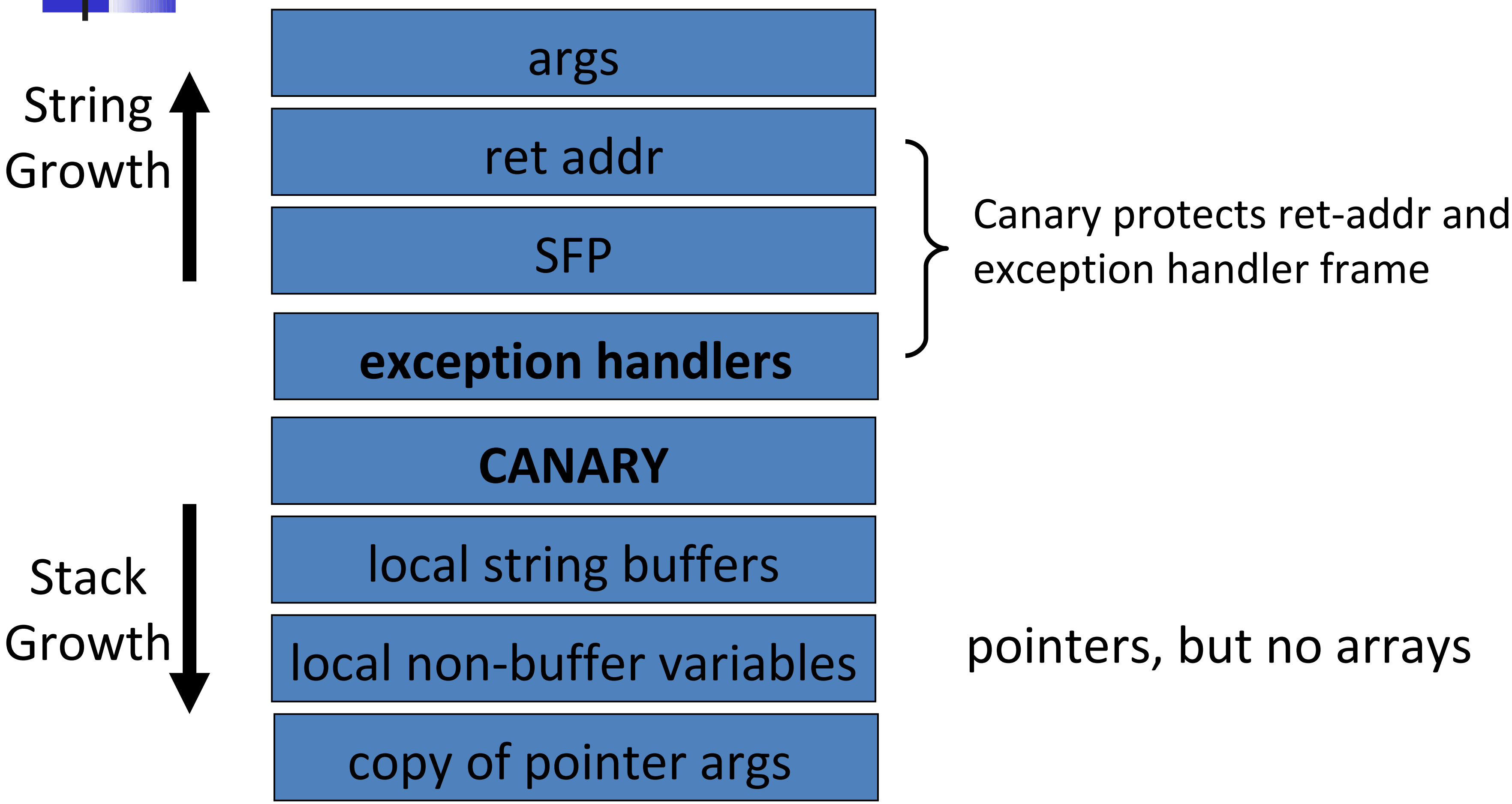
Function epilog:

```
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



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 attack