https://crypto.stanford.edu/cs155

CS155

Computer Security

Control Hijacking

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Control Hijacking

Basic Control Hijacking Attacks

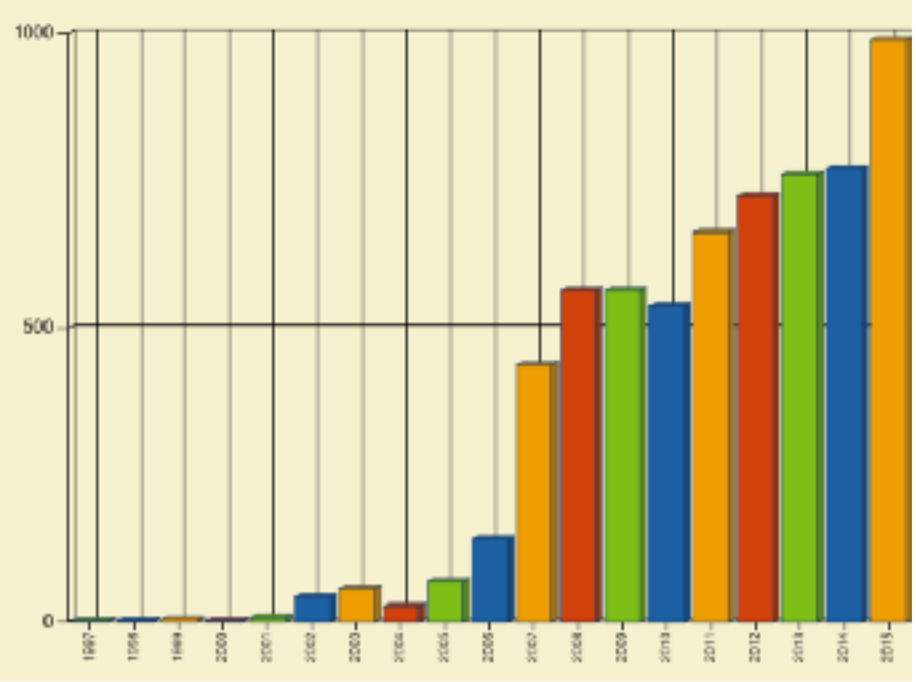
Control hijacking attacks

- 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.



Source: web.nvd.nist.gov

What is needed

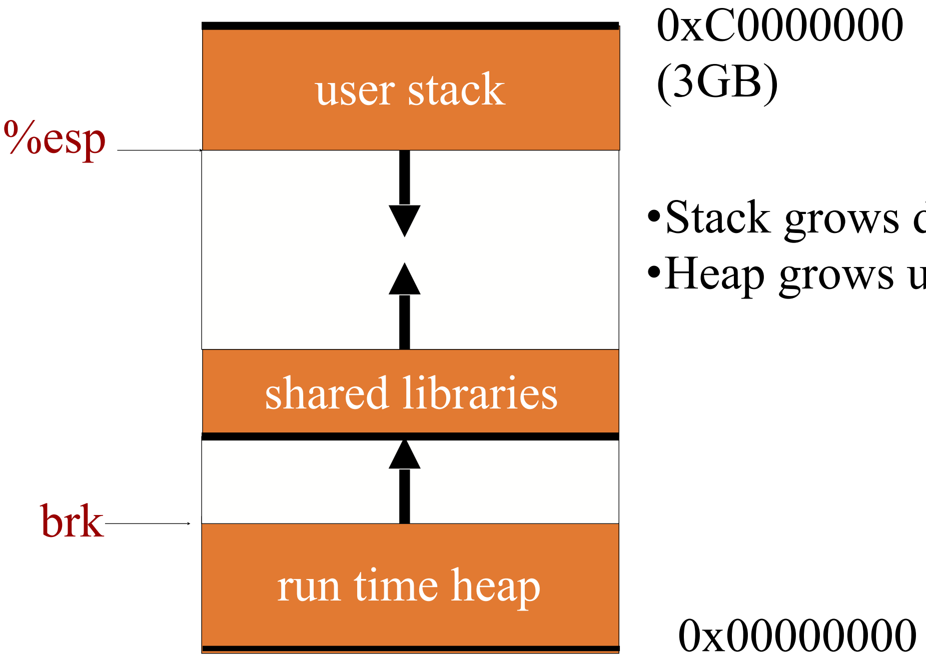
- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

- 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)

Memory Organization

[Brumley]

Memory
Program text
Shared libs
Data
…



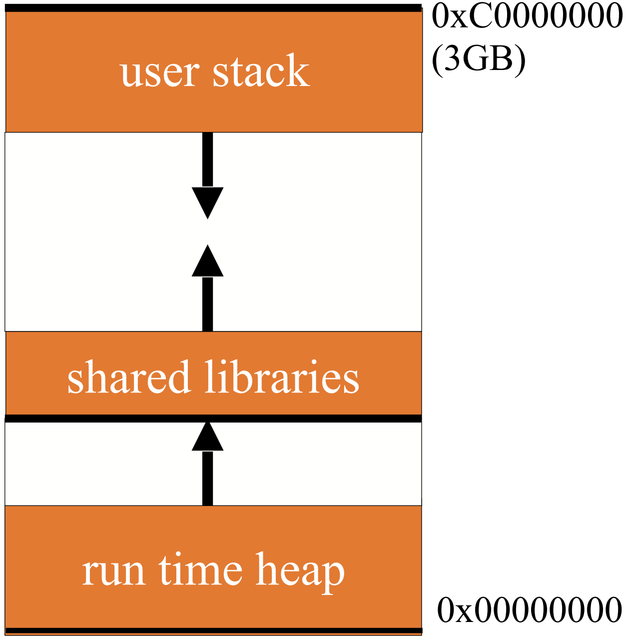
- Stack grows down
- Heap grows up

The Stack grows down towards lower addresses.

[Brumley]

Variables

- On the stack
  - Local variables
  - Lifetime: stack frame
- On the heap
  - Dynamically allocated via new/malloc/etc.
  - Lifetime: until freed



[Brumley]

Procedures

- Procedures are not native to assembly
- Compilers *implement* procedures
  - On the stack
  - Following the call/return stack discipline

[Brumley]

Procedures/Functions

- We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables
- A stack frame provides space for these values
  - Each procedure invocation has its own stack frame
  - Stack discipline is LIFO
    - If procedure A calls B, B's frame must exit before A's

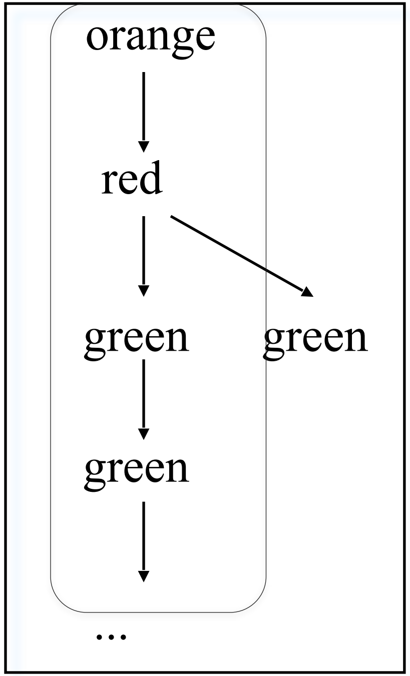
[Brumley]

Function Call Chain

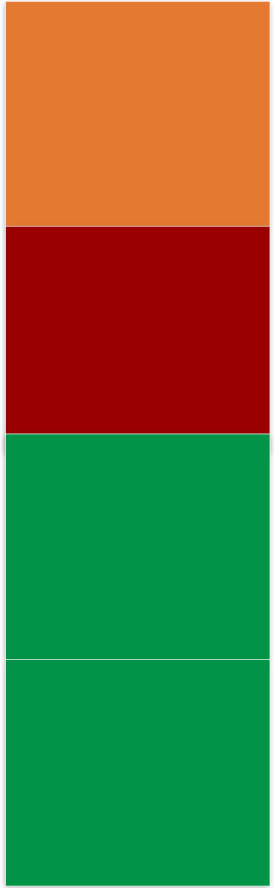
```
orange(…)
{
      …
      red()
      …
}
```

```
red(…)
{
      …
      green()
      …
      green()
}
```

```
green(…)
{
      …
      green()
      …
}
```



[Brumley]

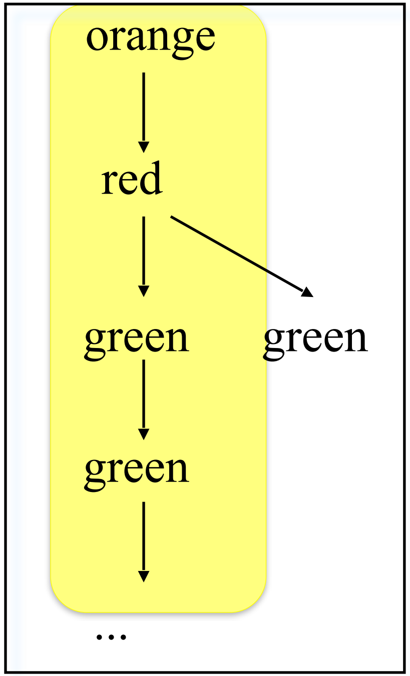


Frame for
- locals
- pushing parameters
- temporary space

Call to red “***pushes***” new frame

When green returns it “***pops***” its frame

Function Call Chain



[Brumley]

On the stack

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```

Need to access arguments

Need space to store local vars (buf, c, and d)

Need space to put arguments for callee

Need a way for callee to return values

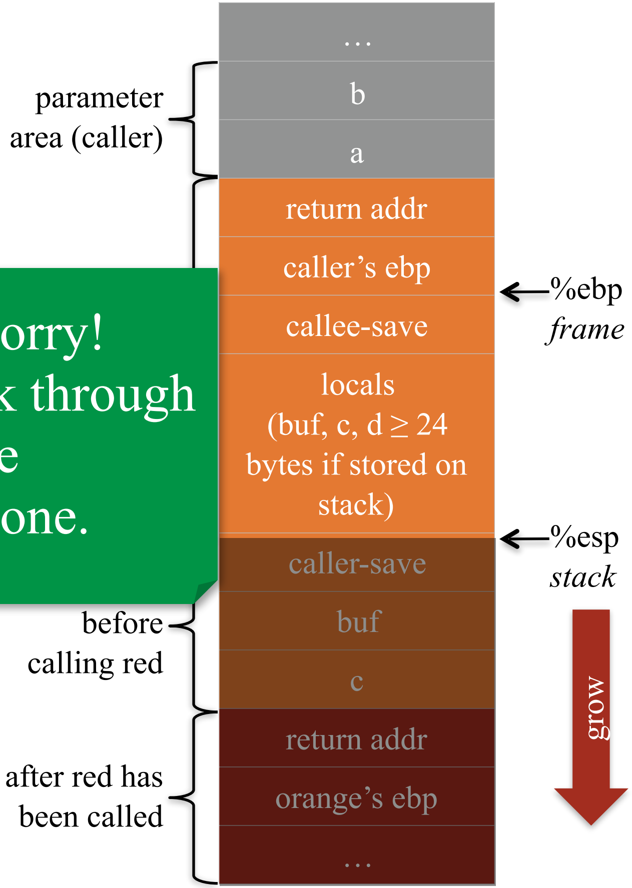
Calling convention determines the above features

[Brumley]

cdecl - the default for Linux & gcc

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```

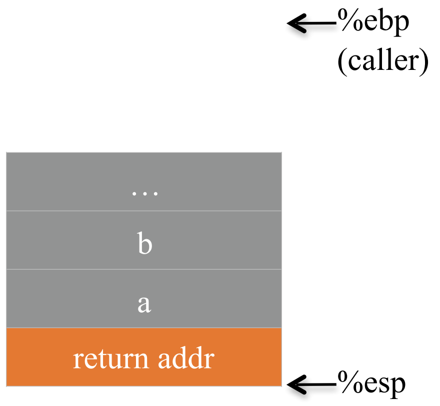
Don't worry!
We will walk through these
one by one.



[Brumley]

When orange attains control,

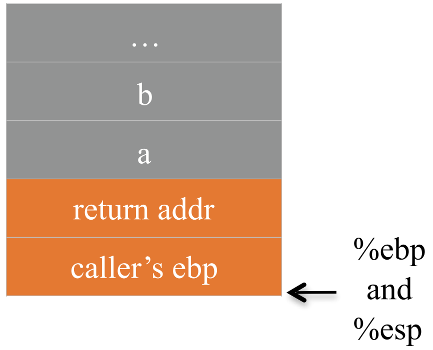
1. return address has already been pushed onto stack by caller



[Brumley]

When orange attains control,

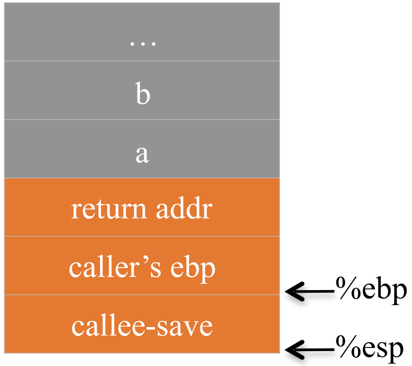
1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8



[Brumley]

When orange attains control,

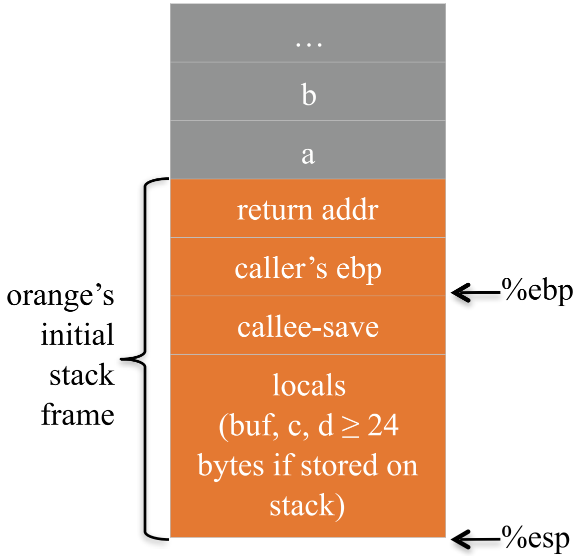
1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic



[Brumley]

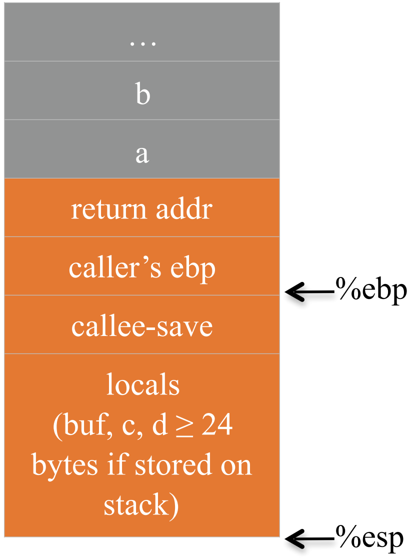
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - "live" variables in registers, which on contention, can be "***spilled***" to stack space



[Brumley]

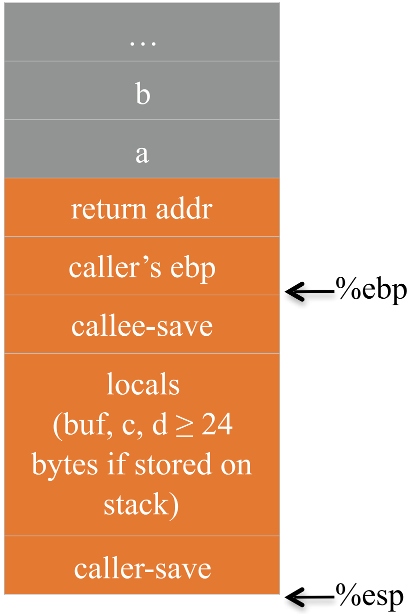
For *caller* orange to call *callee* red,



[Brumley]

For *caller* orange to call *callee* red,

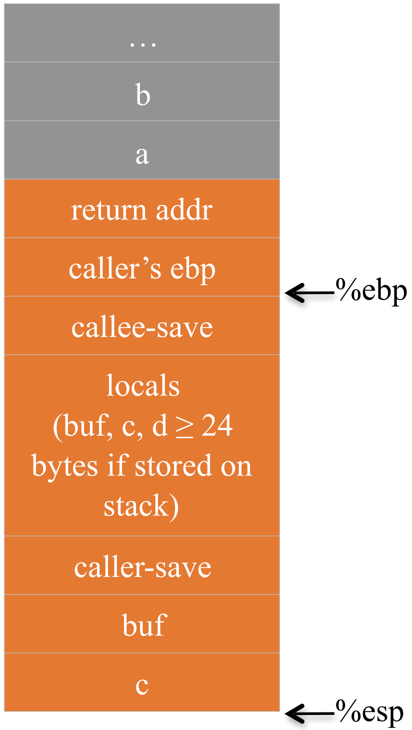
1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx



[Brumley]

For *caller* orange to call *callee* red,

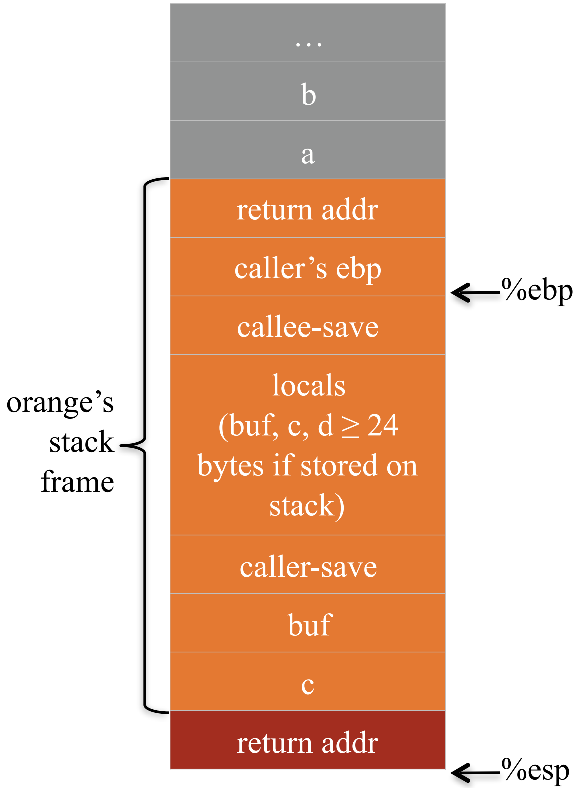
1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack



[Brumley]

For *caller* orange to call *callee* red,

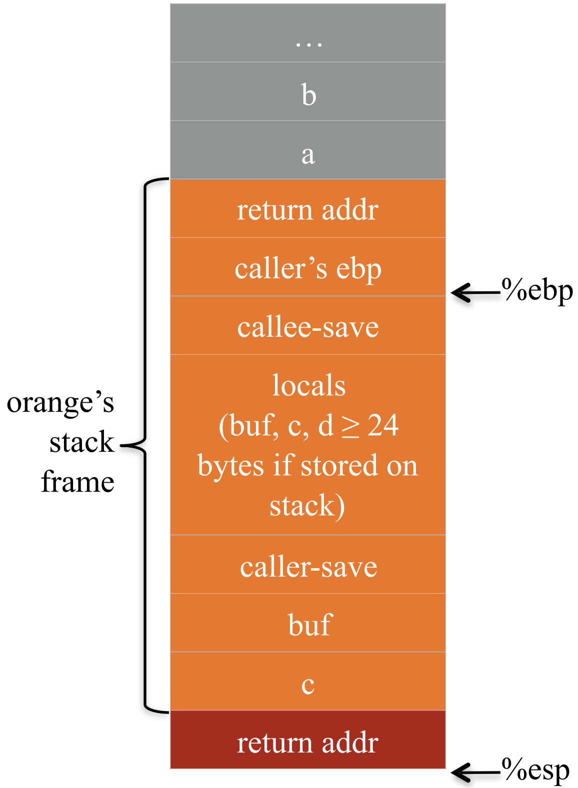
1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns



[Brumley]

For *caller* orange to call *callee* red,

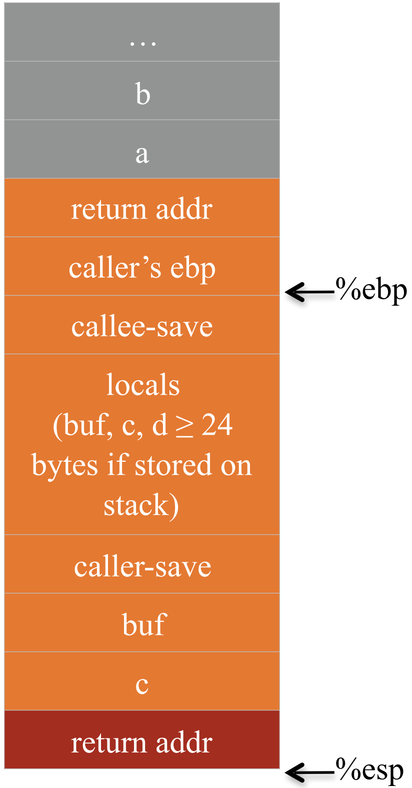
1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns
4. transfer control to red
   - usually happens together with step 3 using `call`



[Brumley]

When red attains control,

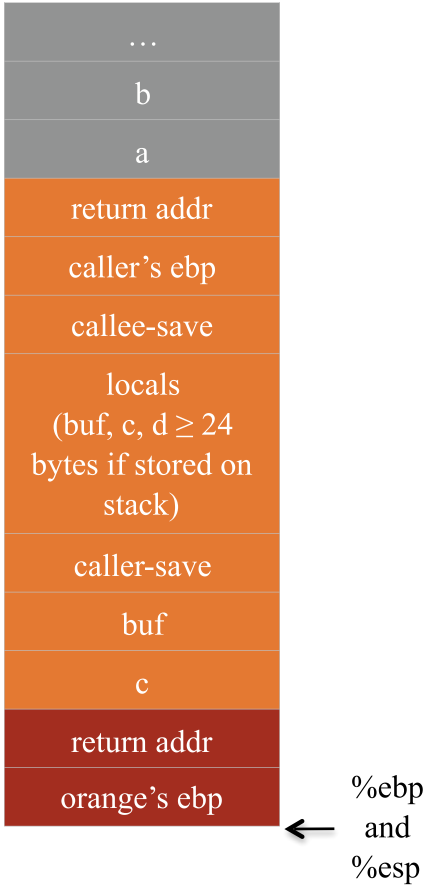
1. return address has already been pushed onto stack by orange



[Brumley]

When red attains control,

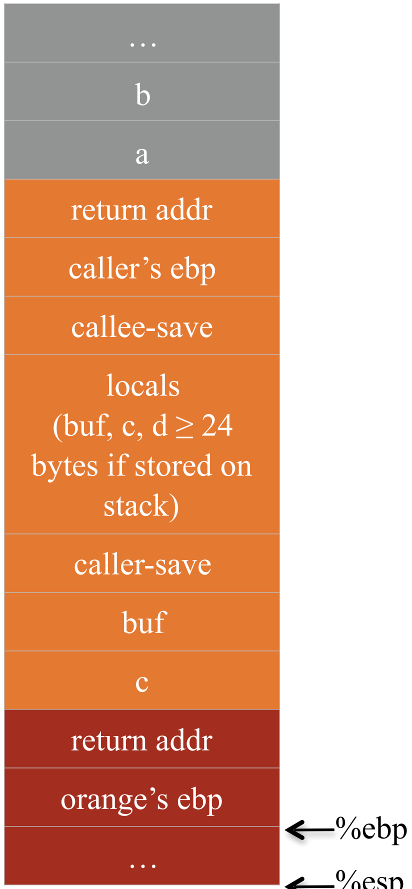
1. return address has already been pushed onto stack by orange
2. own the frame pointer



[Brumley]

When red attains control,

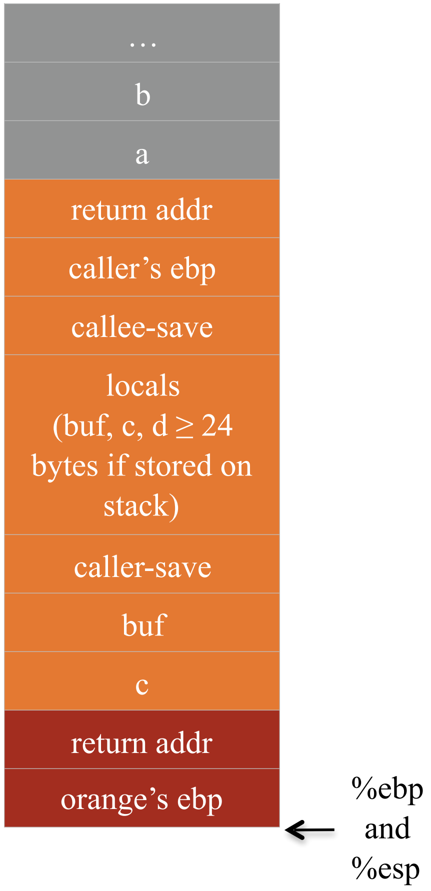
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …



[Brumley]

When red attains control,

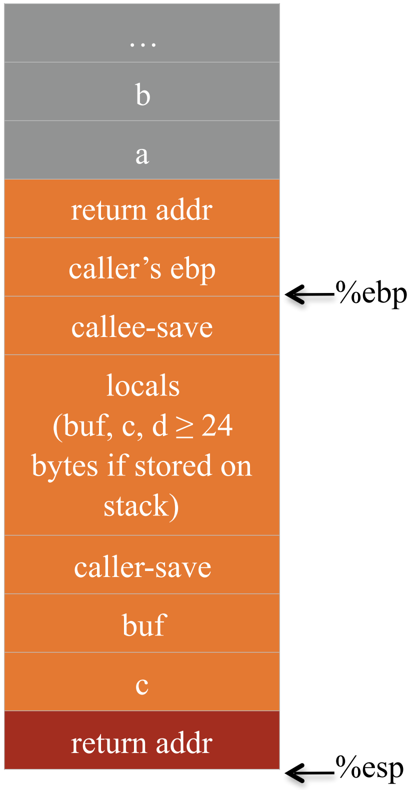
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers



[Brumley]

When red attains control,

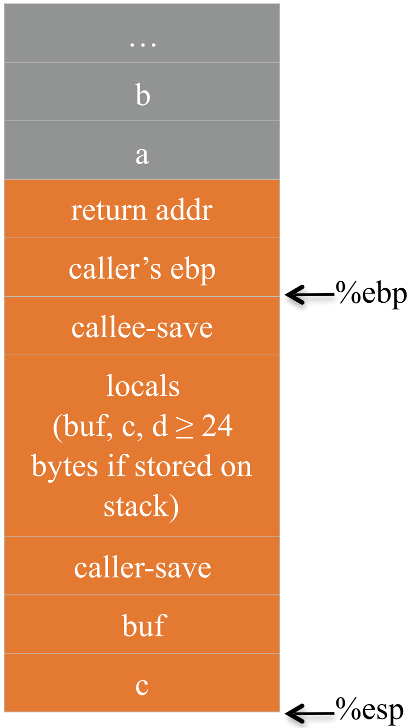
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange's frame pointer
   - `pop %ebp`



[Brumley]

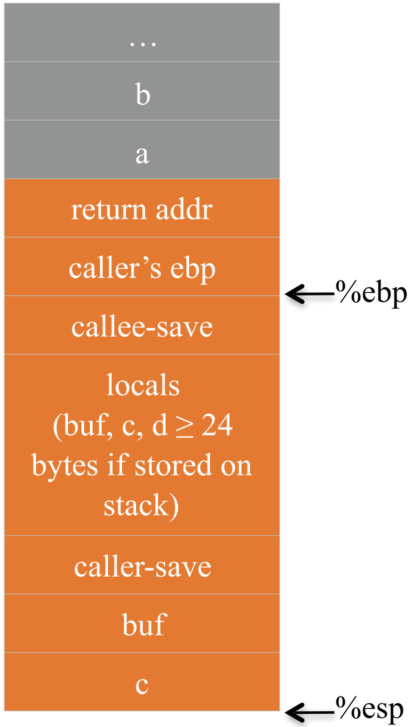
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange's frame pointer
   - `pop %ebp`
8. return control to orange
   - `ret`
   - pops return address from stack and jumps there



[Brumley]

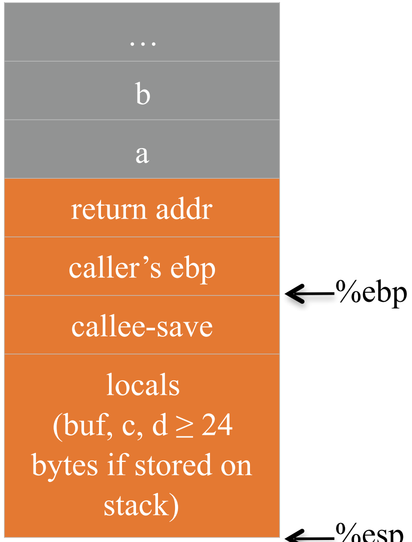
When orange regains control,



[Brumley]

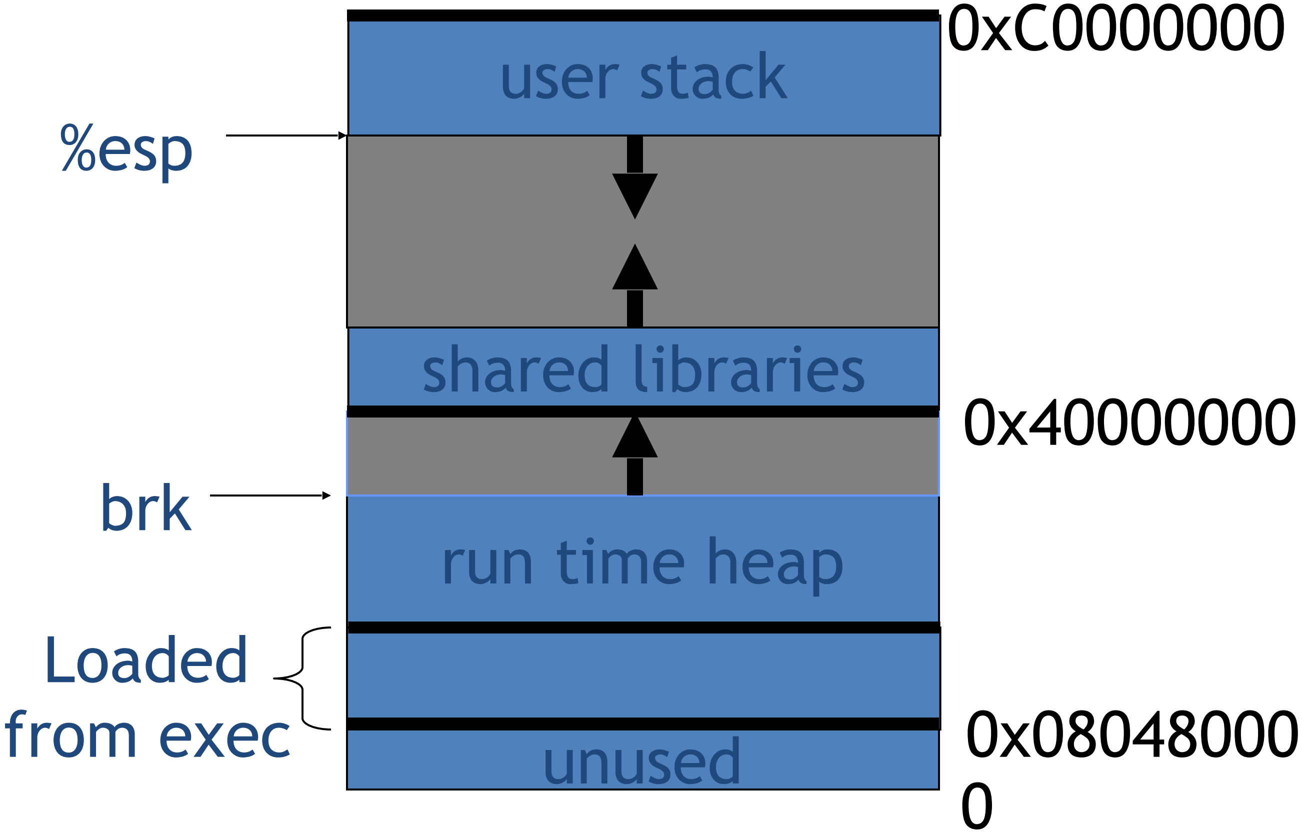
When orange regains control,

1. clean up arguments to red
   - adding to esp
2. restore any caller-save registers
   - pops
3. …



[Brumley]

Linux process memory layout



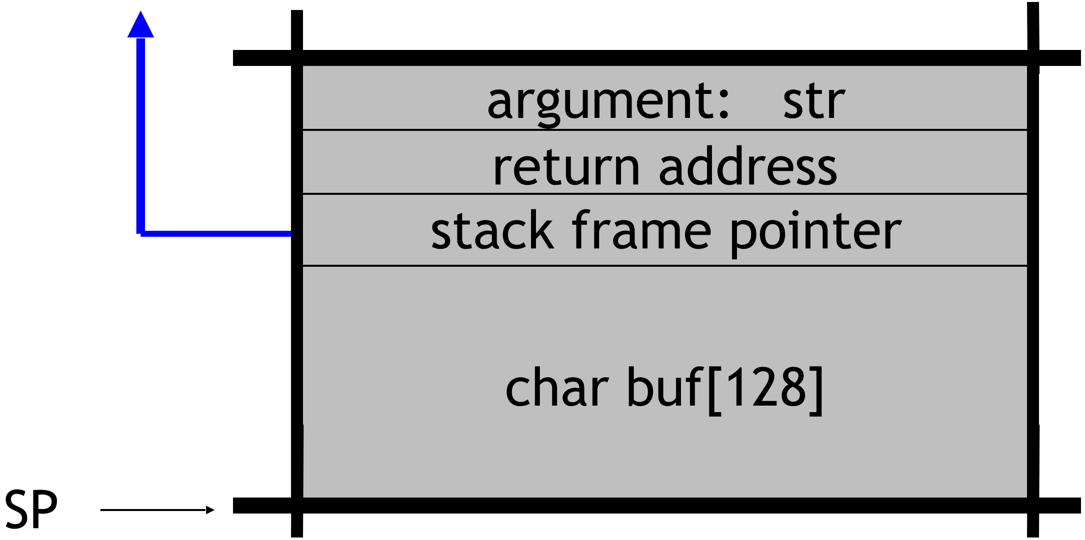
str

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:



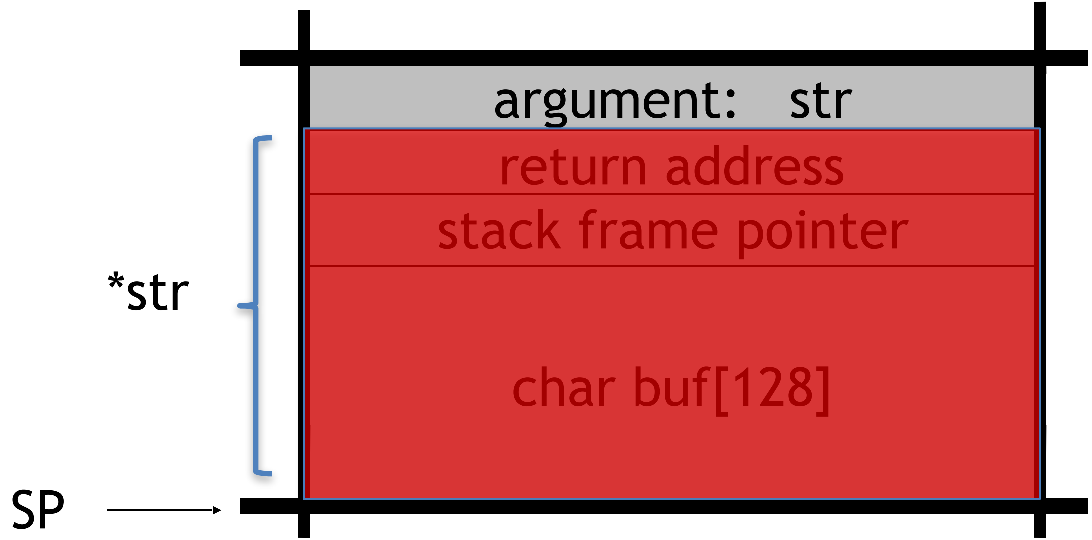
What are buffer overflows?

What if **`*str`** is 136 bytes long?

After **`strcpy`**:

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

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



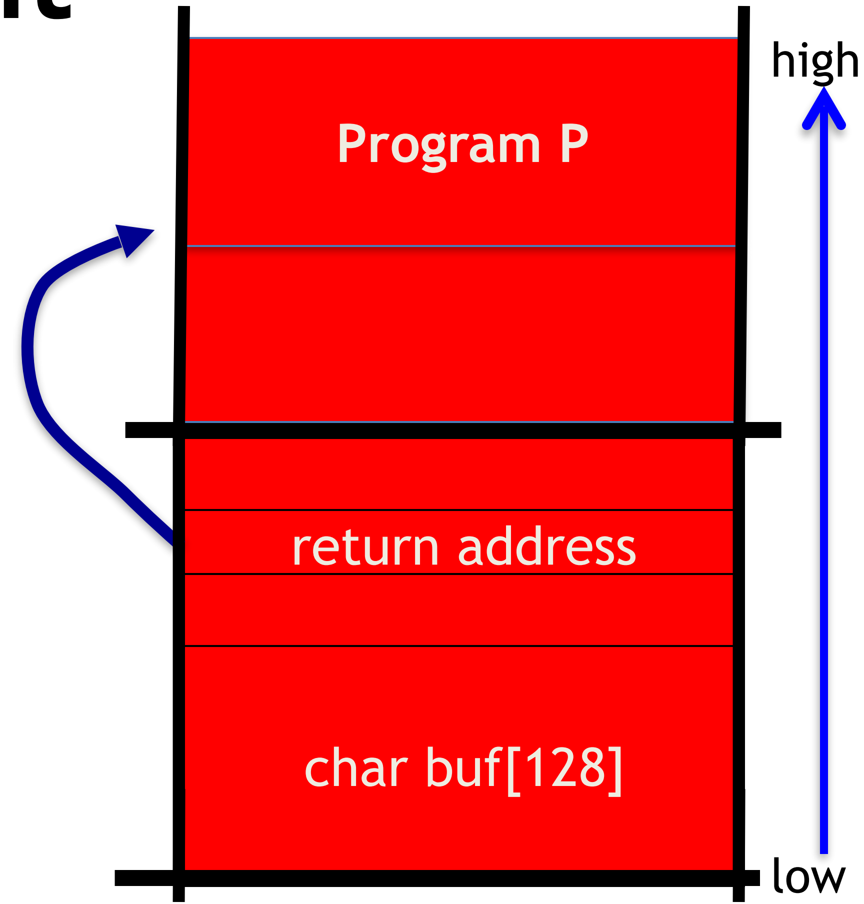
Problem:

no length checking in `strcpy`

Basic stack exploit

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

Program P: exec("/bin/sh")
(exact shell code by Aleph One)



When func() exits, the user gets shell !

Note: attack code P runs *in stack*.

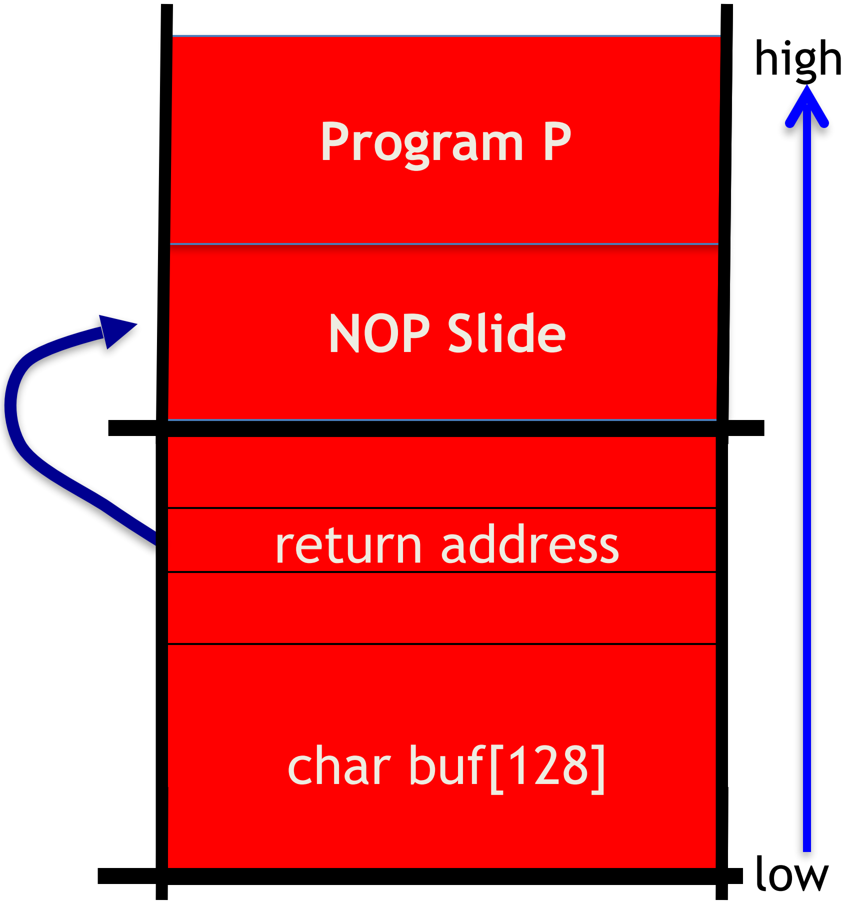
The NOP slide

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



Dan Boneh

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:
  - Overflow in Windows animated cursors (ANI). LoadAniIcon()
  - Past 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.

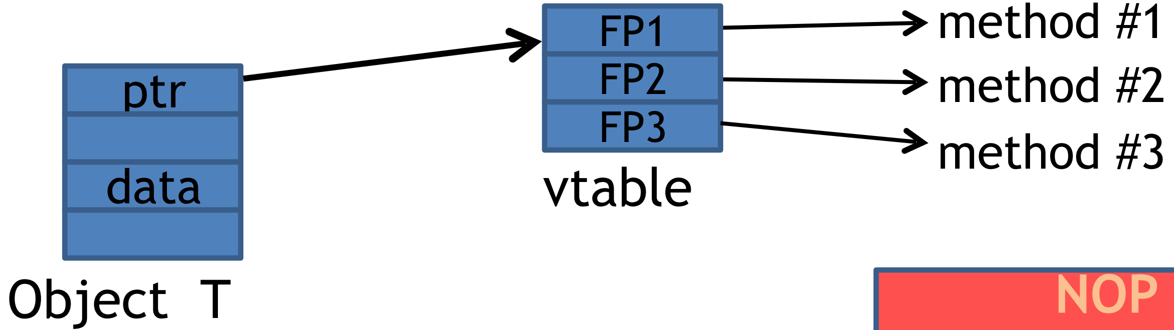
- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)

  - Overflowing buf will override function pointer.

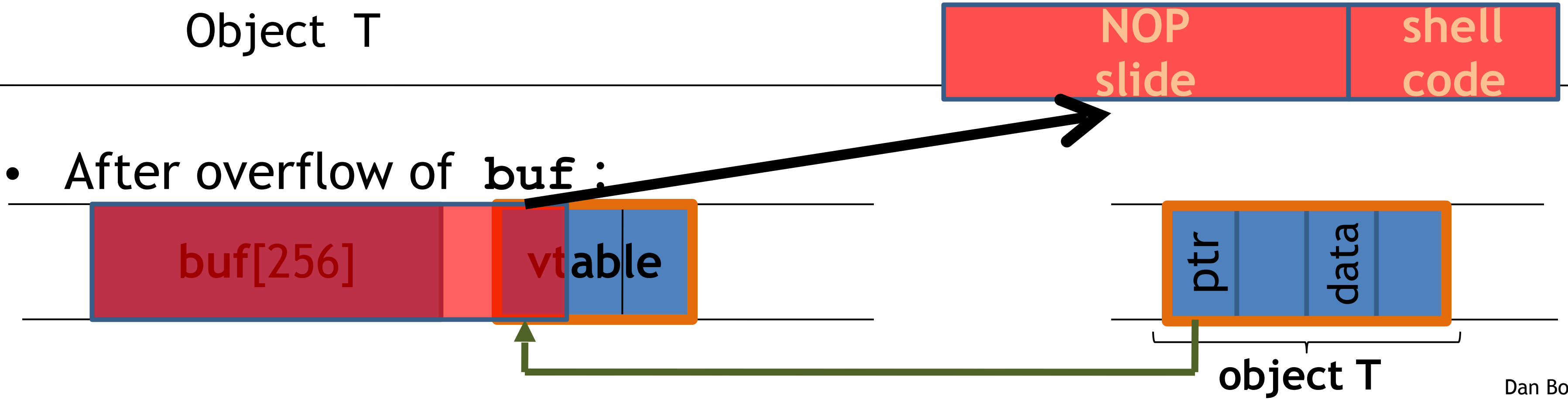
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

Corrupting method pointers

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



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



Dan Boneh

Finding buffer overflows

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with "$$$$$")
    - Many automated tools exist (called fuzzers – next week)
  - If web server crashes,
    search core dump for "$$$$$" 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

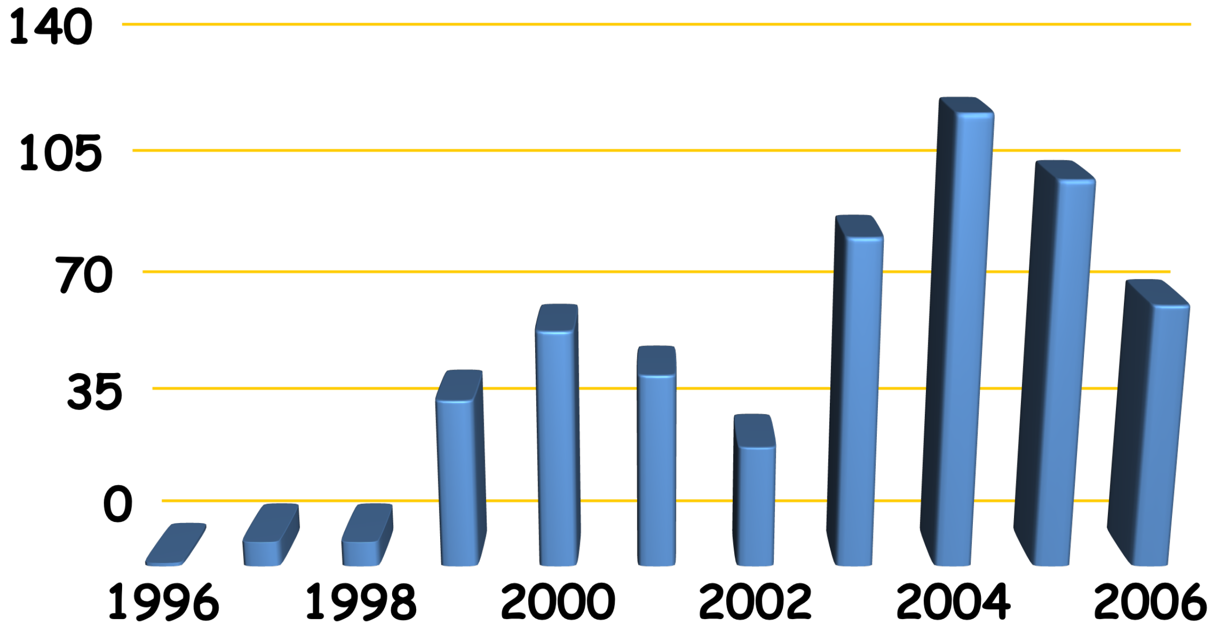
```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      if  (len1 + len2 > 256)  {return -1}          // length check
      memcpy(temp, buf1, len1);                     // cat buffers
      memcpy(temp+len1, buf2, len2);
      do-something(temp);                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

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

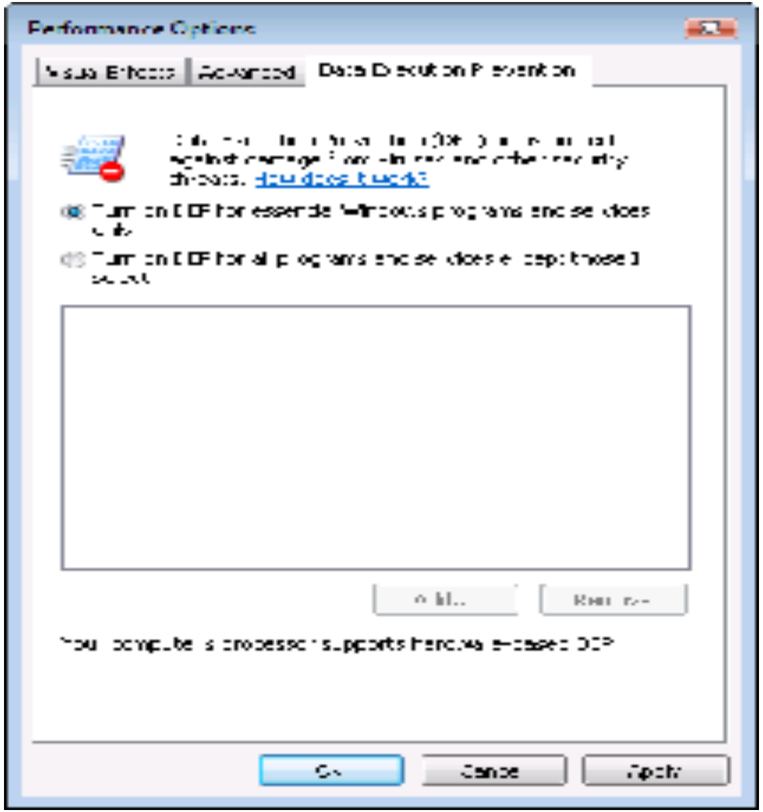
3. Add runtime code to detect overflows exploits
   – Halt process when overflow exploit detected
   – StackGuard, LibSafe, …

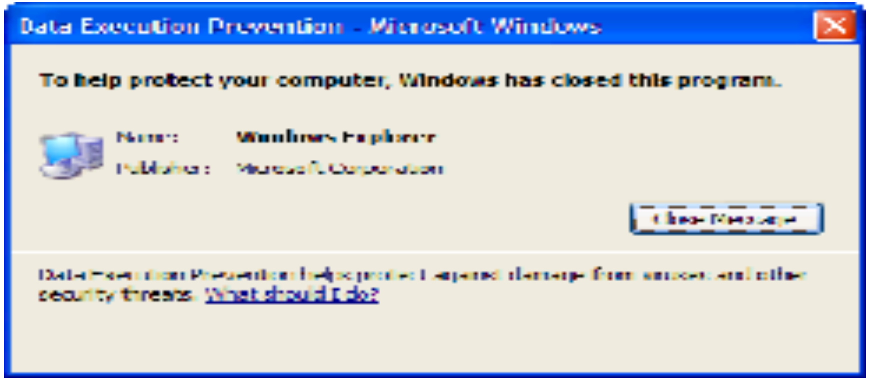
Marking memory as non-execute (DEP)

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

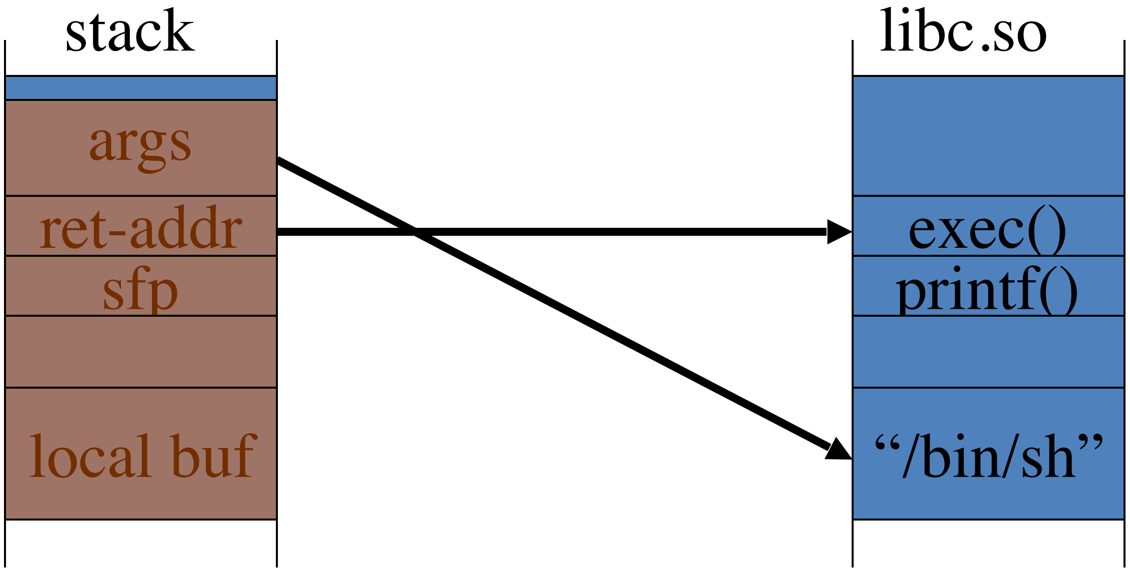




DEP terminating a program

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
    - **Windows 8:** 24 bits of randomness on 64-bit processors
- Other randomization methods:
  - Sys-call randomization: randomize syscall 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

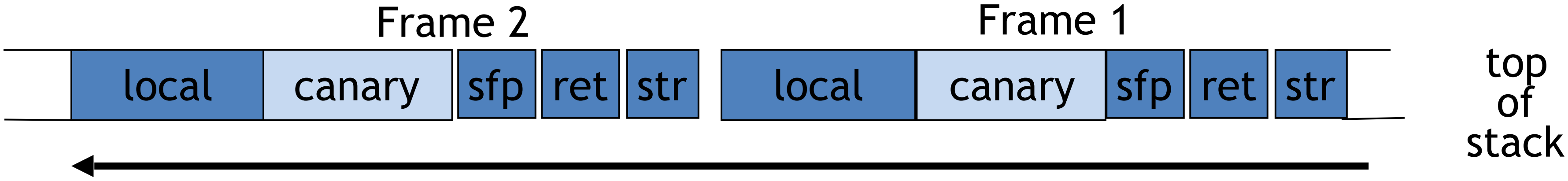
Control Hijacking Defenses

Hardening the executable

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

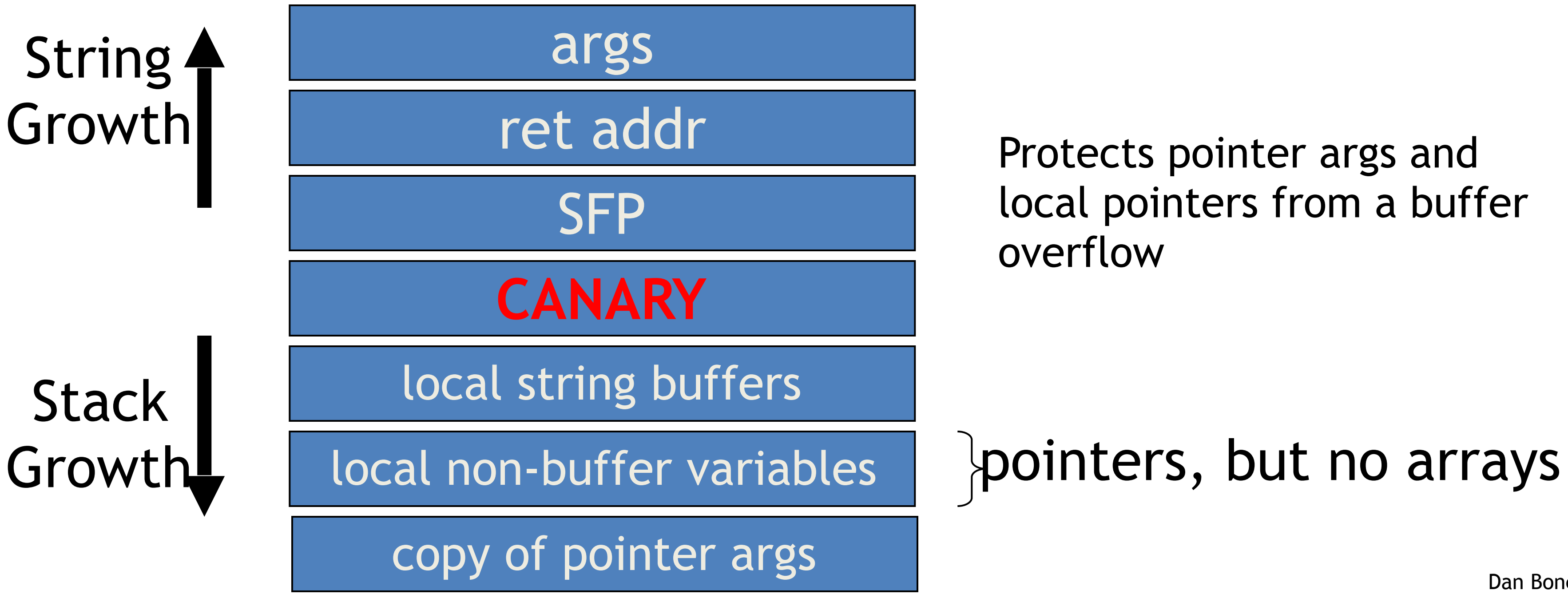
- Minimal performance effects: 8% for Apache

- 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
  - 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:

```
sub   esp, 4      // allocate 4 bytes for cookie
mov   eax, DWORD PTR ___security_cookie
xor   eax, esp      // xor cookie with current esp
mov   DWORD PTR [esp], eax   // save in stack
```

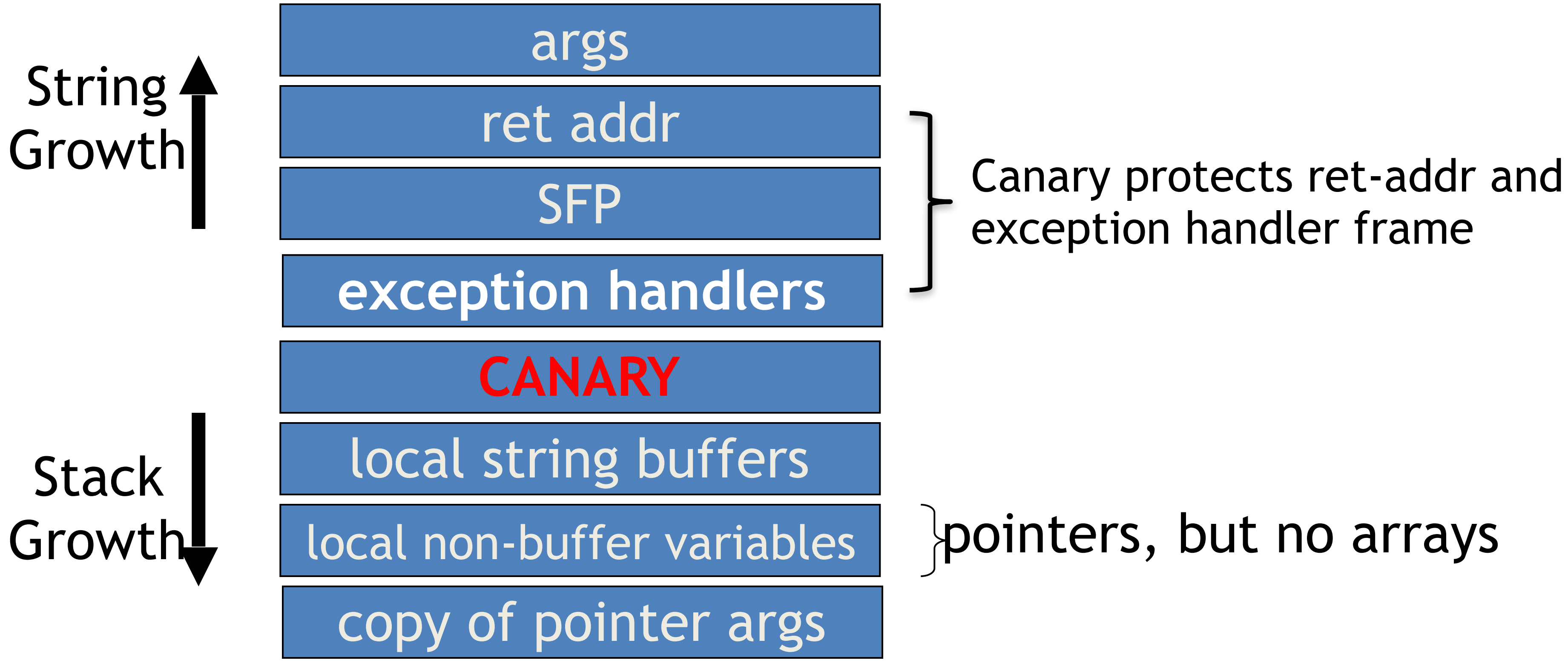
Function epilog:

```
mov   ecx, DWORD PTR  [esp]
xor   ecx, esp
call  @__security_check_cookie@4
add   esp, 4
```

Enhanced /GS in Visual Studio 2010:

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

/GS stack frame



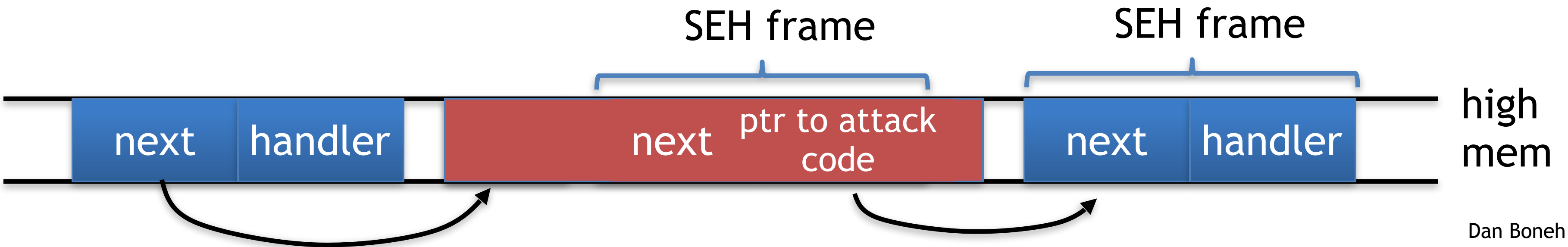
Evading /GS with exception handlers

- 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

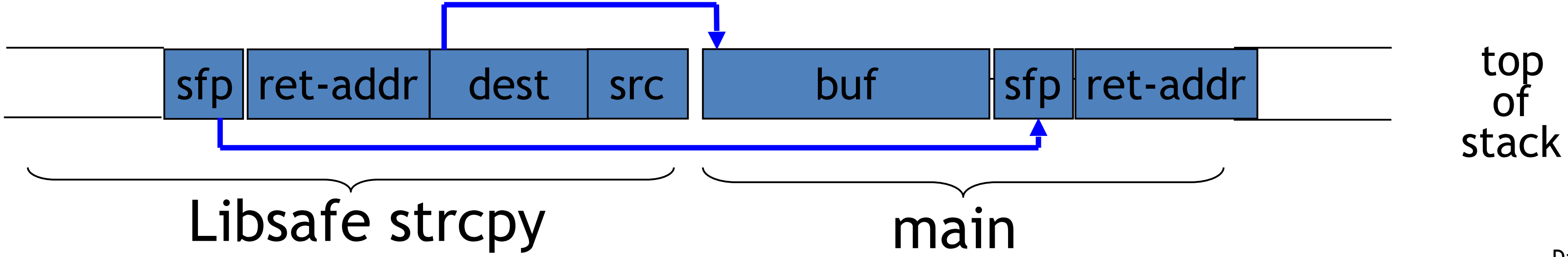
    (also need SAFESEH and SEHOP)

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 Flow Guard (CFG) (Windows 10)

Poor man's version of CFI:

- Protects indirect calls by checking against a bitmask of all valid function entry points in executable

```
rep stosd
mov     esi, [esi]
mov     ecx, esi            ; Target
push    1
call    @_guard_check_icall@4 ; _guard_check_icall(x)
call    esi
add     esp, 4
xor     eax, eax
```

ensures target is the entry point of a function

Control Flow Guard (CFG) (Windows 10)

Poor man's version of CFI:

- P ... of
  a ...

- Does not prevent attacker from causing a jump to a valid <u>wrong</u> function

```
rep
mov
mov
push
call      @_guard_check_icall@4 ; _guard_check_icall(x)
call      esi
add       esp, 4
xor       eax, eax
```

s
of a

Control Hijacking

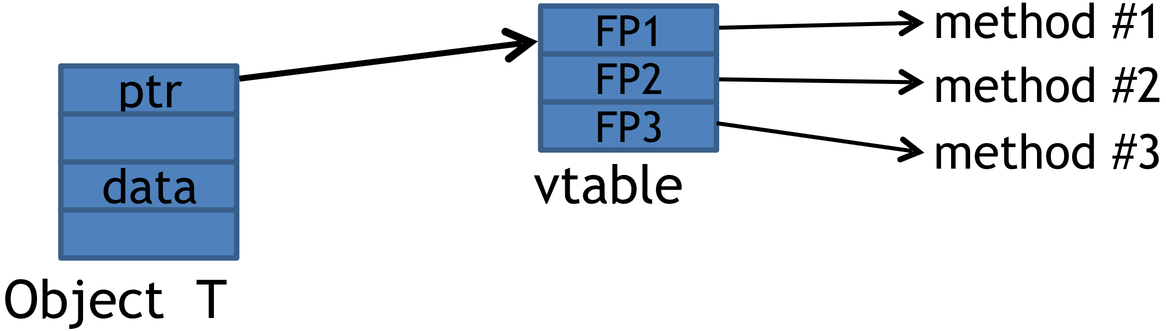
Advanced Hijacking Attacks

Heap Spray Attacks

A reliable method for exploiting heap overflows

Heap-based control hijacking

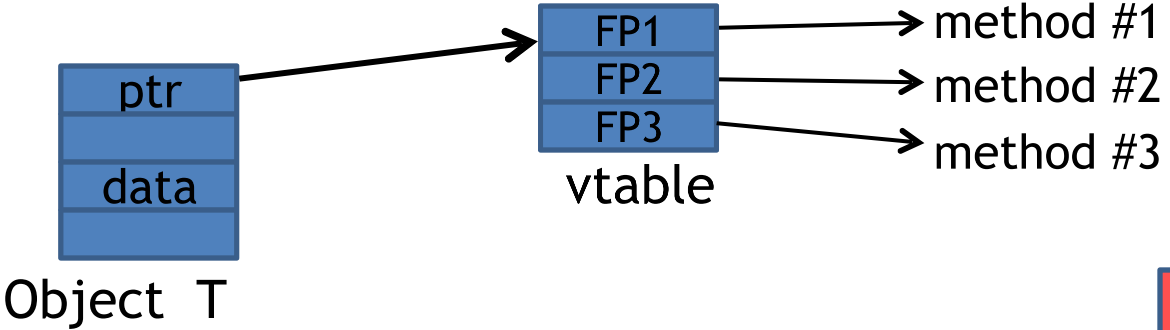
- Compiler generated function pointers (e.g. C++ code)



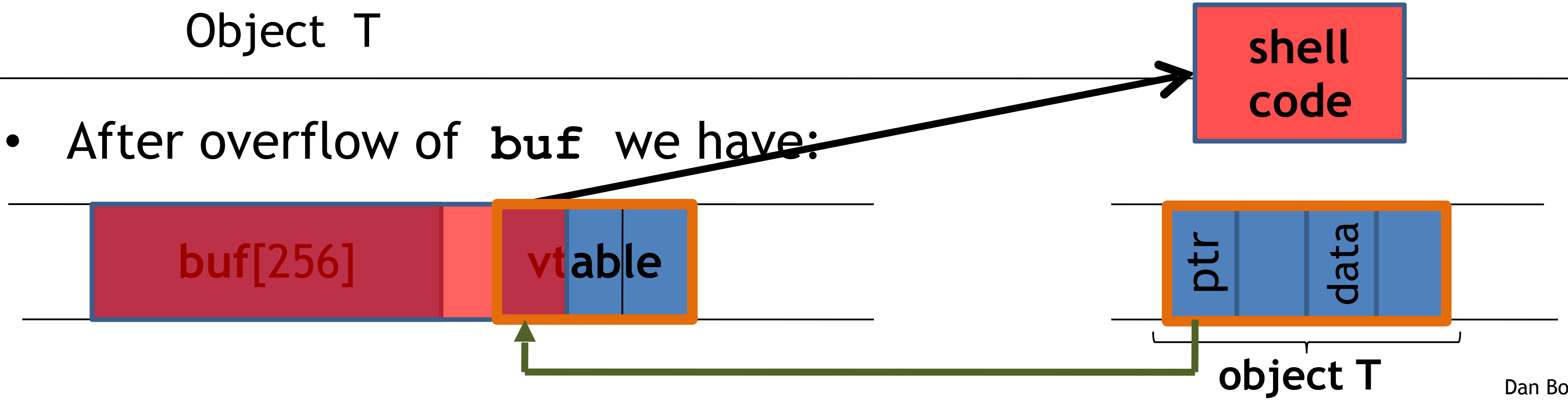
- Suppose vtable is on the heap next to a string object:

Heap-based control hijacking

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



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

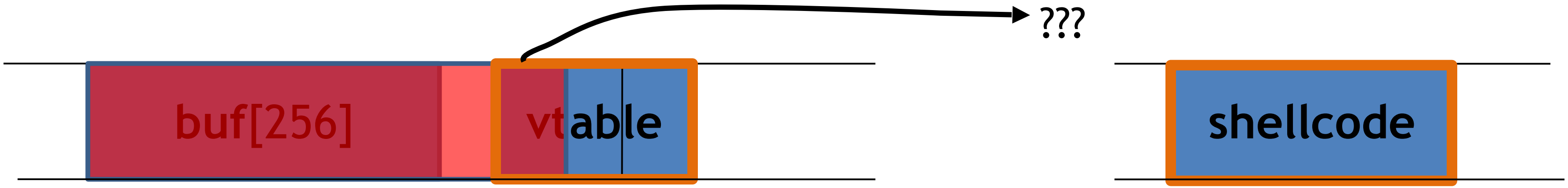


A reliable exploit?

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

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

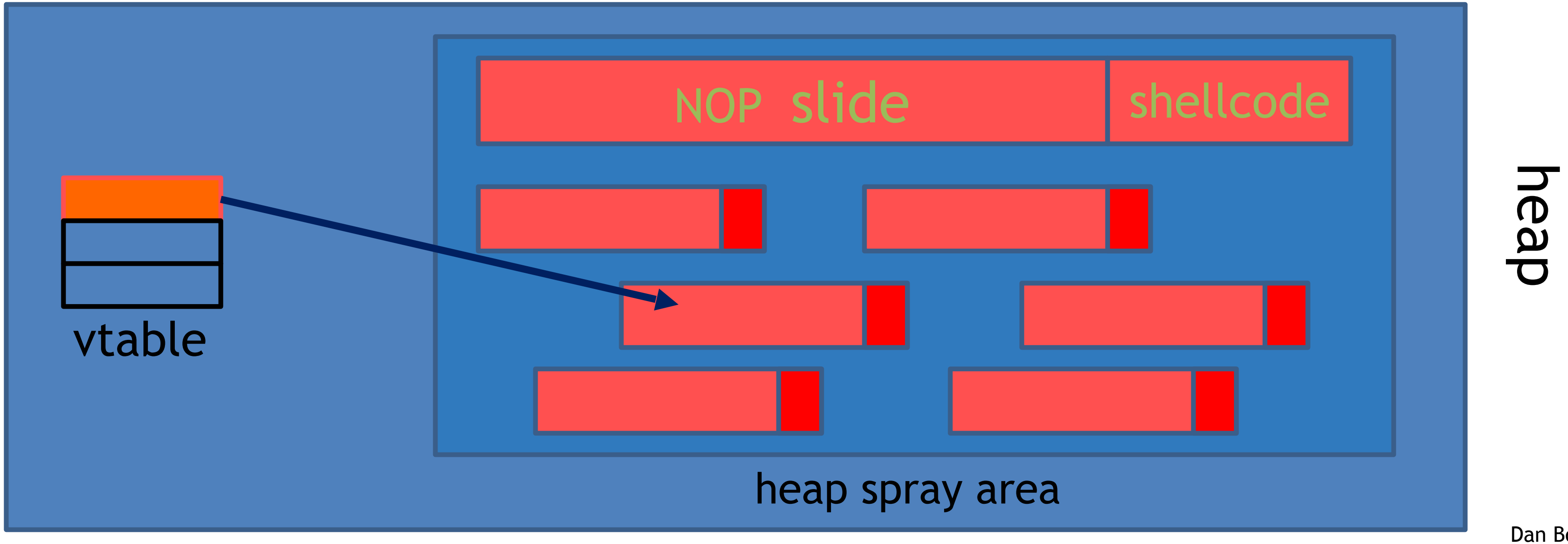
Problem: attacker does not know where browser places **shellcode** on the heap



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.

Many heap spray exploits

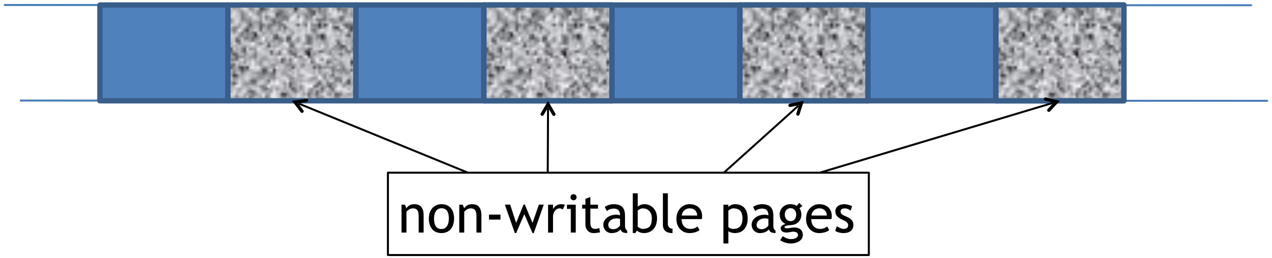
[RLZ'08]

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

Acknowledgments/References

- Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

- [DanBoneh] *CS 155: Computer Security, Dan Boneh, Stanford University, 2015.*
- [Brumley] CS1848: Introduction to Computer Security, Carnegie Mellon University, 2016.