Cyber Security: Concept, Theory and Practice

ECE 509

Lecture 9 : Basic Control Hijacking Attacks

Instructor: Salim Hariri
Dept of Electrical and Computer Engineering
University of Arizona

What we covered so far?

Application Security & Resilience

User and Web Applications	Mobile Platforms	Web Protocols
Encryption	Forensic Analysis	Insider Threats

Operating System Security

Basic Control Hijacking	Rootkits, Isolation

Computer Networks and Protocols Security

Computer Networks		Communication Protocols	
Wireless	Wired	IP Based	Non IP Based

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 machine by hijacking application control flow

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

Exploits
Buffer Overflows and Format String Attacks

David Brumley
Carnegie Mellon University

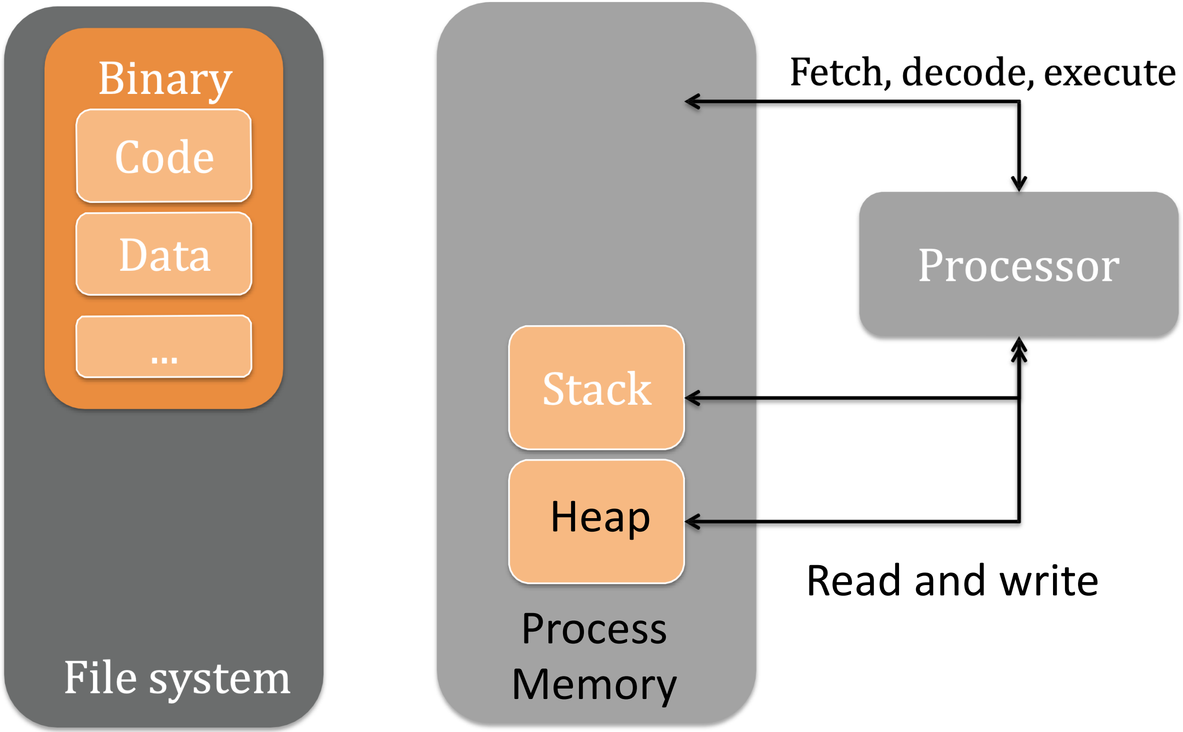
Fact:
Ubuntu Linux
has over
99,000
known bugs

Bugs and Exploits

- A ***bug*** is a place where real execution behavior may ***deviate*** from expected behavior.
- An ***exploit*** is an ***input*** that gives an attacker an advantage

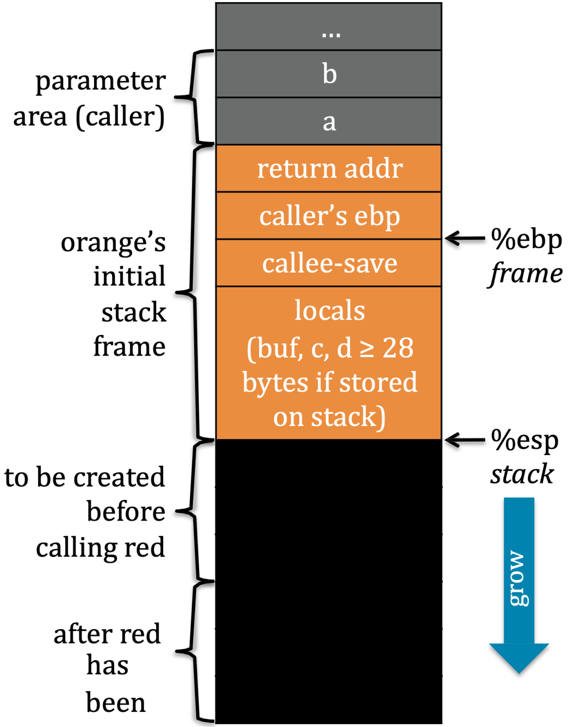
Method	Objective
Control Flow Hijack	Gain control of the instruction pointer `%eip`
Denial of Service	Cause program to crash or stop servicing clients
Information Disclosure	Leak private information, e.g., saved password

Basic Execution



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;
  }
```



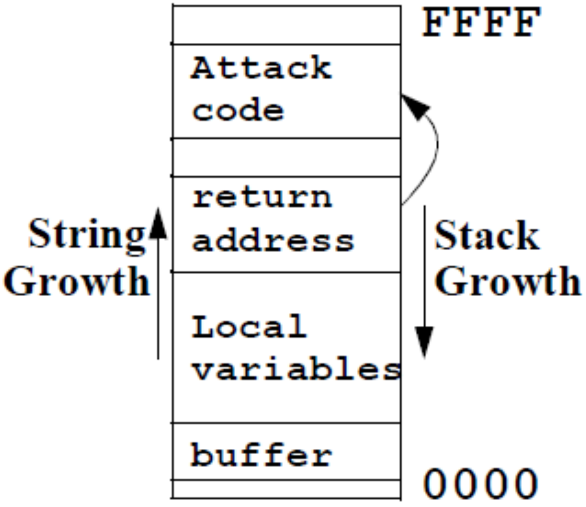


Figure 1: Buffer Overflow Attack Against

Control Flow Hijack:
Always Computation + Control

shellcode (aka payload)	padding	&buf

computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- ...

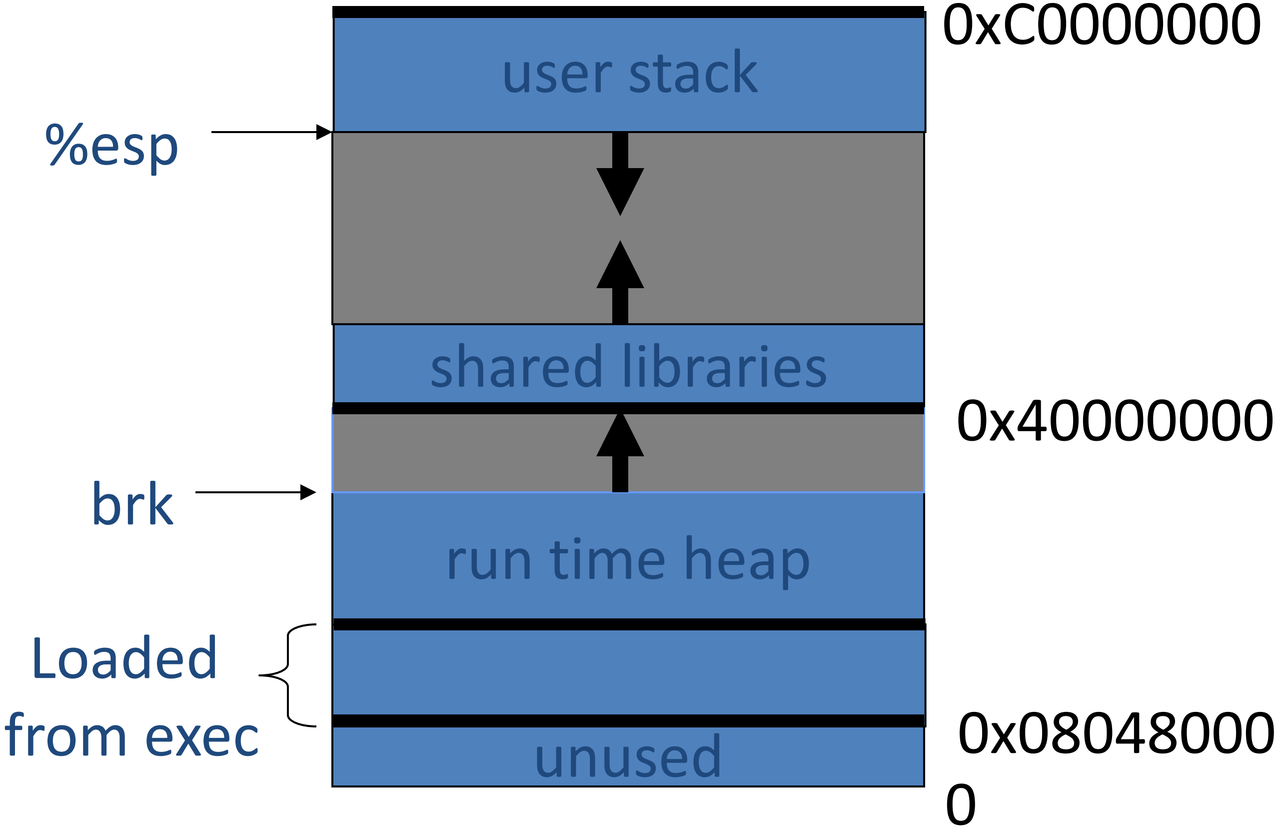
Same principle, different mechanism

What is needed

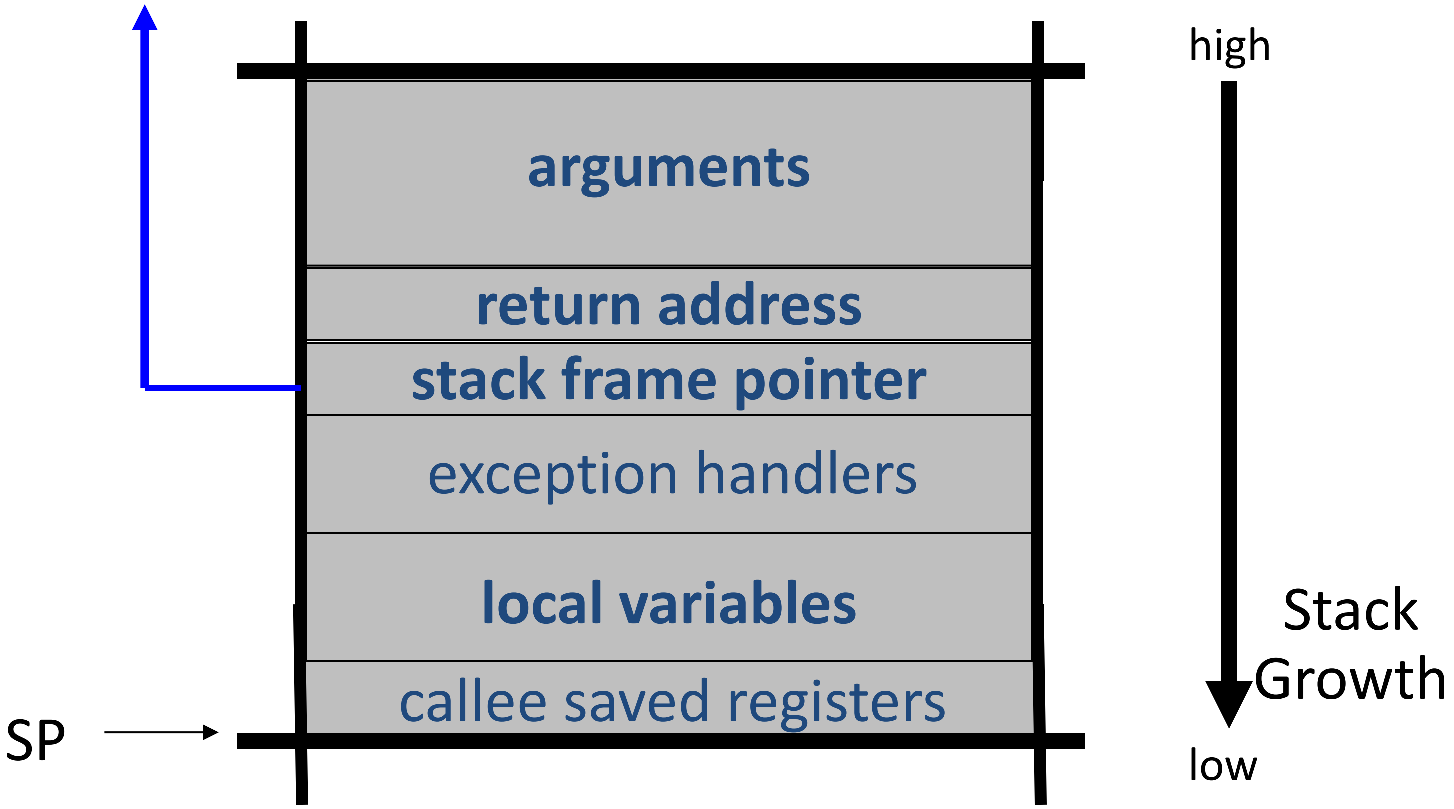
- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

- 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



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

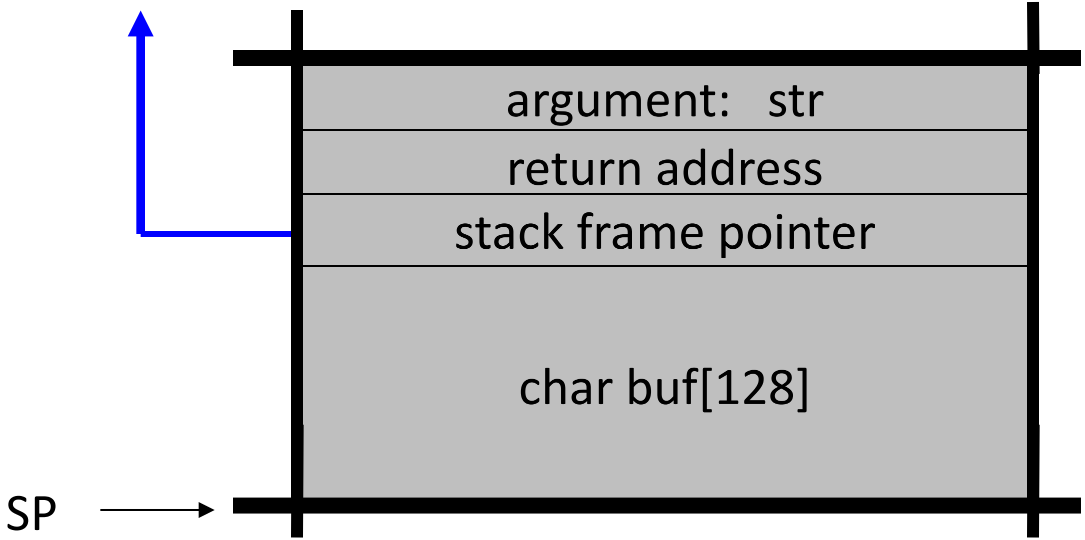
What are buffer overflows?

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:



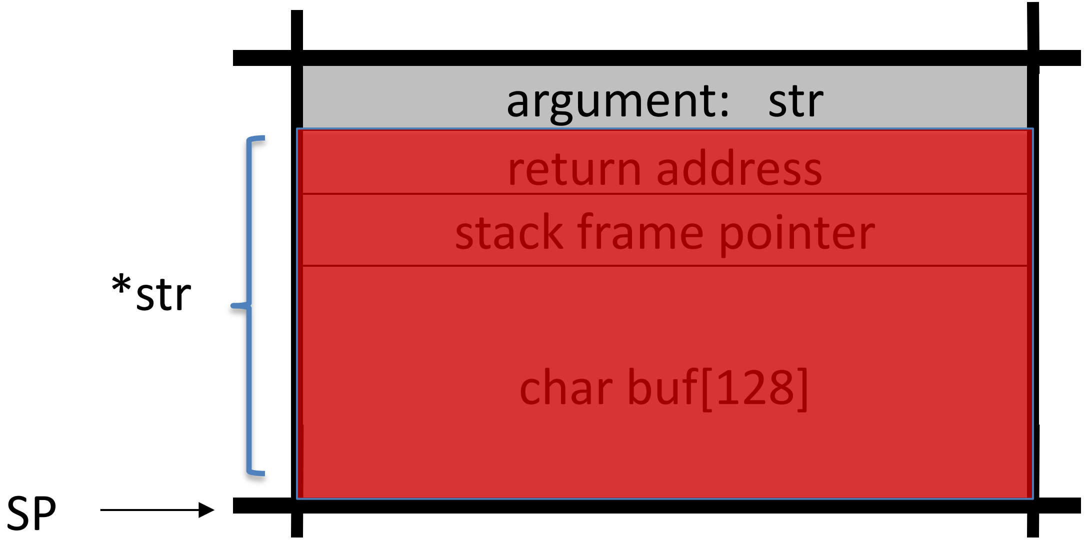
What are buffer overflows?

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

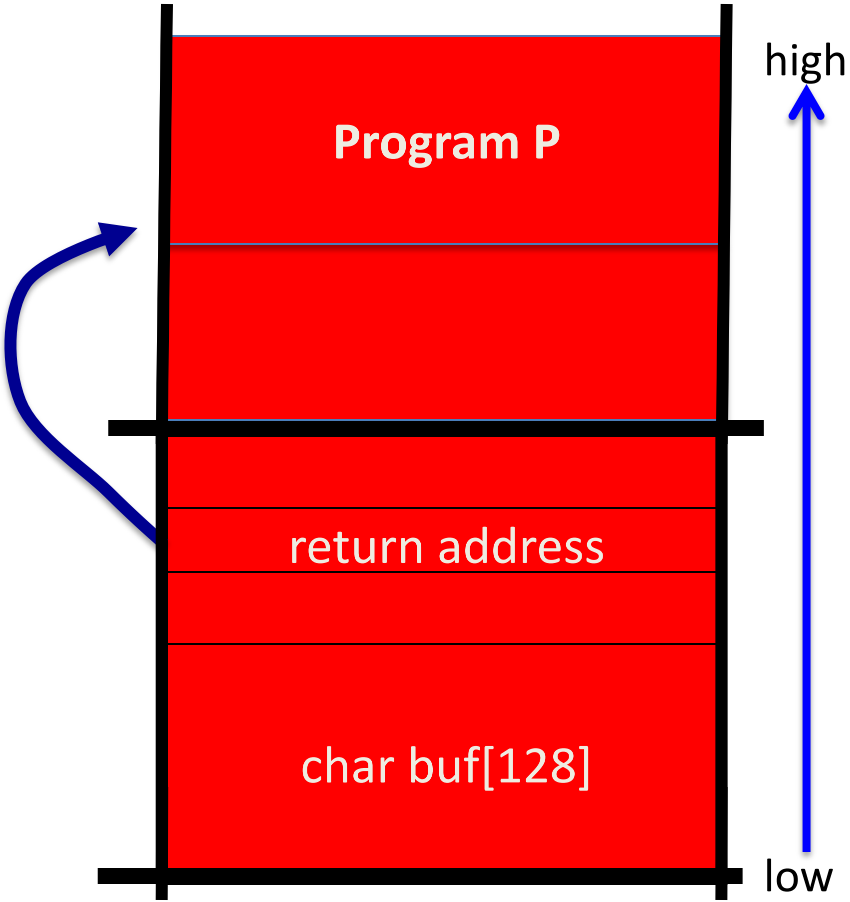
Suppose *str is such that
after strcpy stack looks like:

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*.



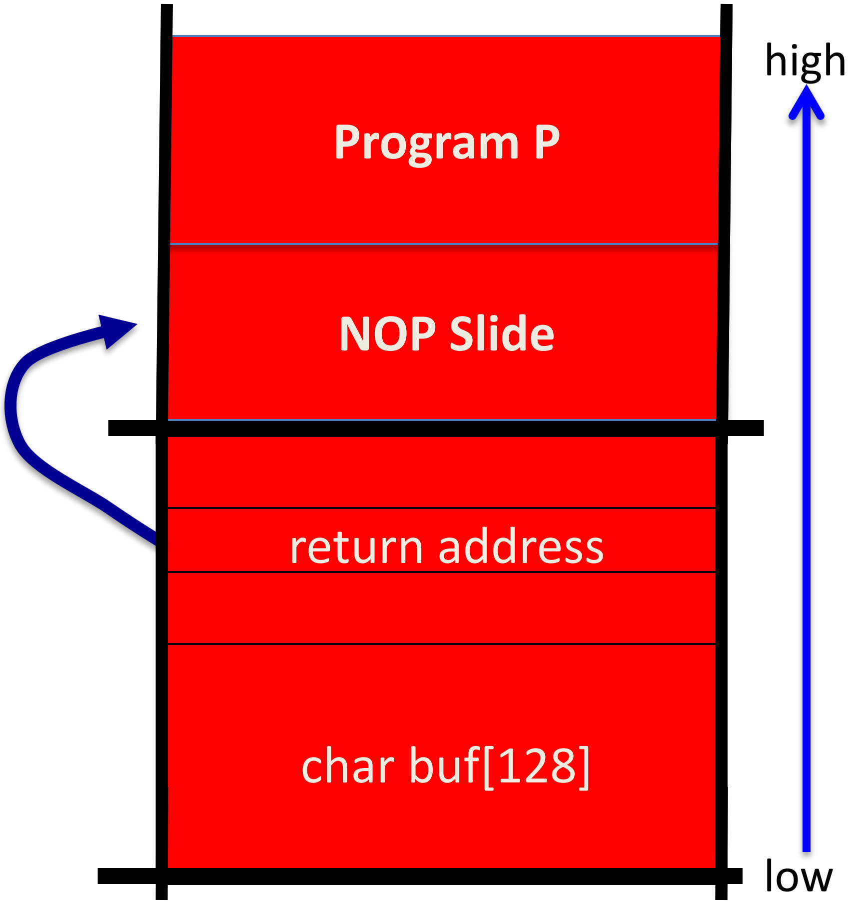
The NOP slide

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



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()
  - (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.

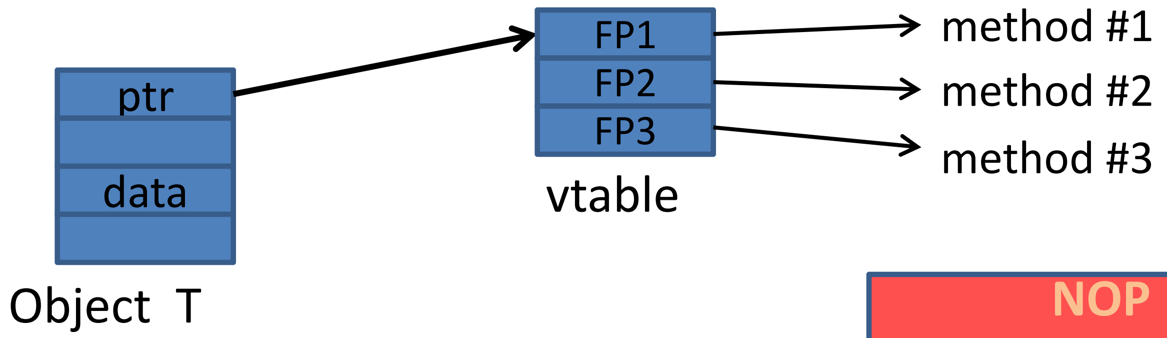
- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)

  - Overflowing buf will override function pointer.

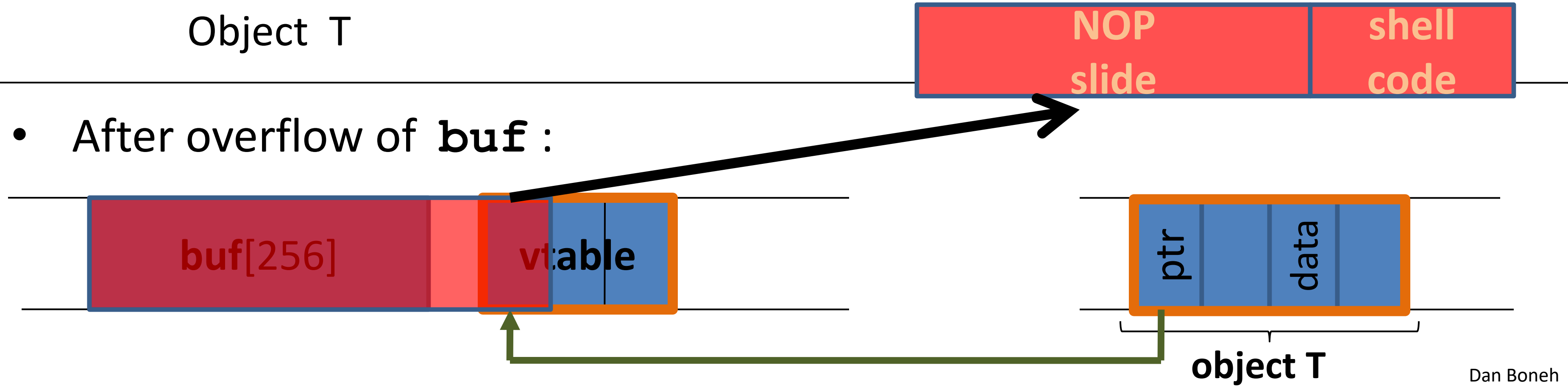
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

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 "$$$$$")
    - Many automated tools exist (called fuzzers – next module)
  - If web server crashes,
    search core dump for "$$$$$" 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**

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)**

$$c = 0x80 + 0x80 = 128 + 128 \quad \Rightarrow \quad c = 0$$

$$s = 0xff80 + 0x80 \quad \Rightarrow \quad s = 0$$

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

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 !!

Format string bugs

How to exploit format string vulnerabilities

a. Viewing memory

b. Overwriting memory

1 Format String

- What is a format string?

```
printf ("The magic number is: %d\n", 1911);
```

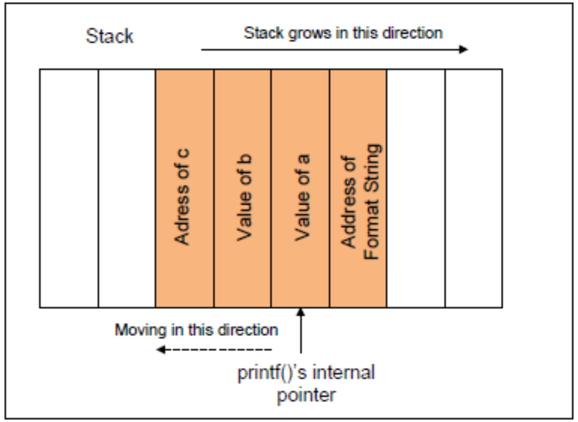
The text to be printed is "The magic number is:", followed by a format parameter '%d', which is replaced with the parameter (1911) in the output. Therefore the output looks like: The magic number is: 1911. In addition to %d, there are several other format parameters, each having different meaning. The following table summarizes these format parameters:

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

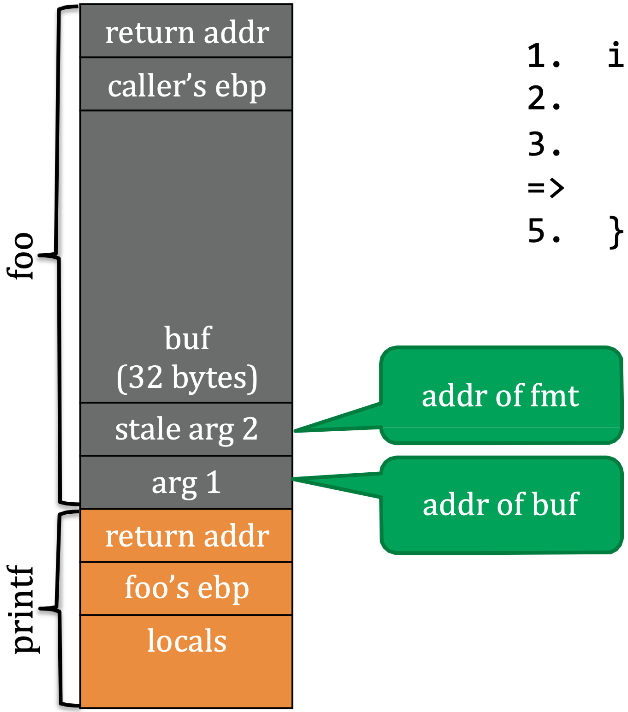
- The stack and its role at format strings

The behavior of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack.

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                                    a, b, &c);
```



Stack Diagram @ printf




```
int foo(char *fmt) {
  char buf[32];
  strcpy(buf, fmt);
  printf(buf);
}
```

Attacks on Format String Vulnerability

- Crashing the program

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

- For each `%s`, `printf()` will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character (i.e., number 0, not character 0) is encountered.
- Since the number fetched by `printf()` might not be an address, the memory pointed by this number might not exist (i.e. no physical memory has been assigned to such an address), and the program will crash.
- It is also possible that the number happens to be a good address, but the address space is protected (e.g. it is reserved for kernel memory). In this case, the program will also crash.

Viewing the stack

```
printf ("%08x %08x %08x %08x %08x\n");
```

- This instructs the printf-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers.
- So a possible output may look like:

```
40012980 080628c4 bffff7a4 00000005 08059c04
```

Viewing memory at any location

We have to supply an address of the memory. However, we cannot change the code; we can only supply the format string.

- If we use `printf(%s)` without specifying a memory address, the target address will be obtained from the stack anyway by the `printf()` function.
- The function maintains an initial stack pointer, so it knows the location of the parameters in the stack.

- Observation: the format string is usually located on the stack. If we can encode the target address in the format string, the target address will be in the stack.

- In the following example, the format string is stored in a buffer, which is located on the stack.

```
int main(int argc, char *argv[])
{
  char user_input[100];
  ... ... /* other variable definitions and statements */

  scanf("%s", user_input); /* getting a string from user */
  printf(user_input); /* Vulnerable place */

  return 0;
}
```

If we can force the printf to obtain the address from the format string (also on the stack), we can control the address.
`printf ("\x10\x01\x48\x08 %x %x %x %x %s");`
`\x10\x01\x48\x08` are the four bytes of the target address. In C language, `\x10` in a string tells the compiler to put a hexadecimal value `0x10` in the current position. The value will take up just one byte. Without using `\x`, if we directly put `"10"` in a string, the ASCII values of the characters `'1'` and `'0'` will be stored. Their ASCII values are 49 and 48, respectively.

- If we can force the printf to obtain the address from the format string (also on the stack), we can control the address.

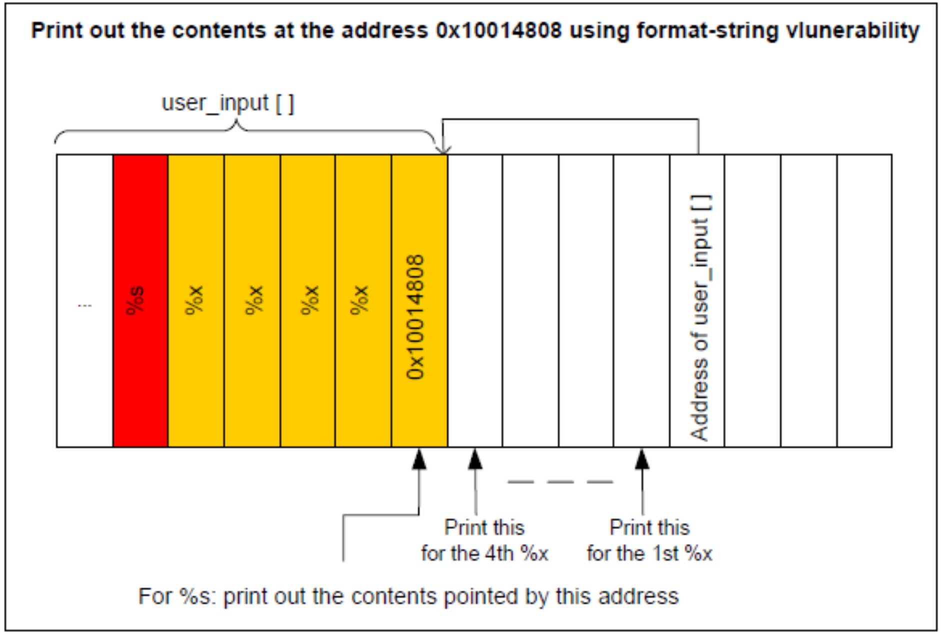
```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```

`\x10\x01\x48\x08` are the four bytes of the target address.

- In C language, `\x10` in a string tells the compiler to put a hexadecimal value `0x10` in the current position. The value will take up just one byte.

  - `%x` causes the stack pointer to move towards the format string.
  - Here is how the attack works if `user_input[]` contains the following format string:

    `"\x10\x01\x48\x08 %x %x %x %x %s".`



Basically, we use four `%x` to move the `printf()`'s pointer towards the address that we stored in the format string. Once we reach the destination, we will give `%s` to `print()`, causing it to print out the contents in the memory address `0x10014808`. The function `printf()` will treat the contents as a string, and print out the string until reaching the end of the string (i.e. 0).

Vulnerable functions

Any function using a format string.

Printing:

- printf, fprintf, sprintf, …
- vprintf, vfprintf, vsprintf, …

Logging:

- syslog, err, warn

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

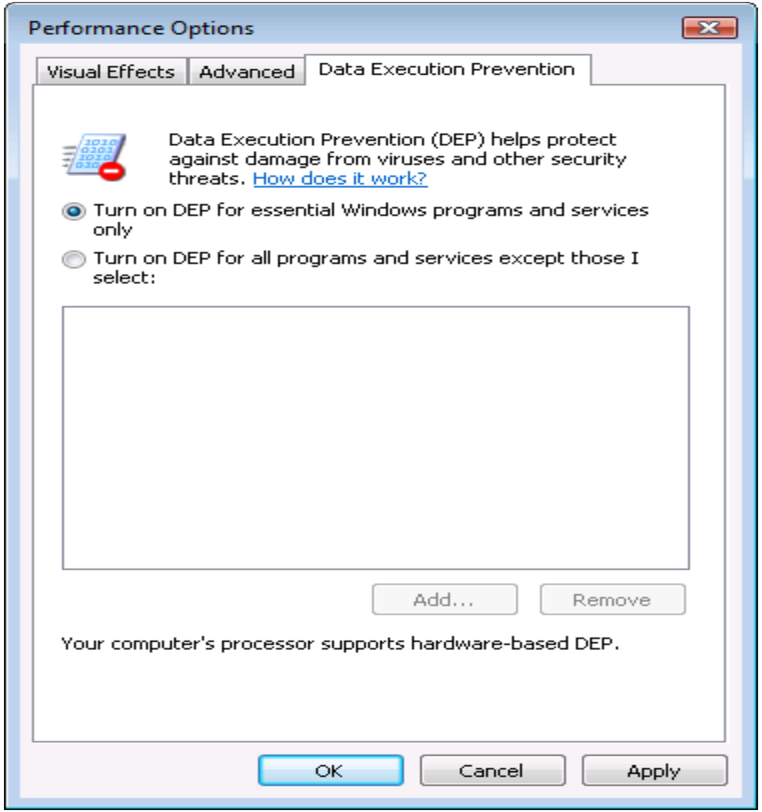
3. Add runtime code to detect overflows exploits
   - Halt process when overflow exploit detected
   - StackGuard, LibSafe, …

Marking memory as non-execute (W^X)

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

- 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

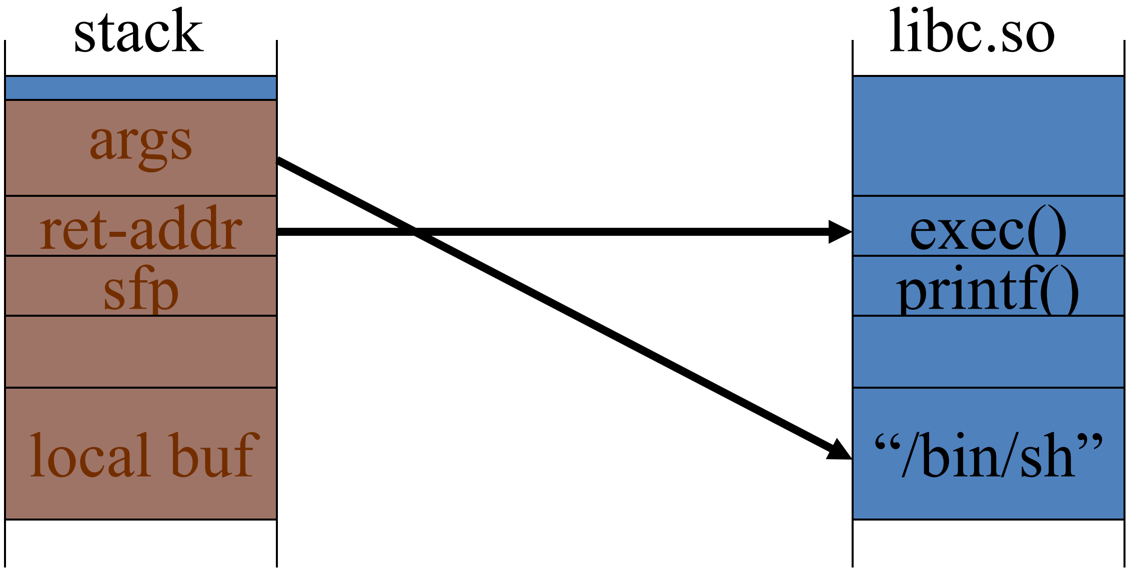
Examples: DEP controls in Windows



DEP terminating a program

Attack: Return Oriented Programming (ROP)

- Control hijacking without executing code



Response: randomization

- **<u>ASLR</u>**: (Address Space Layout Randomization)
  - Map shared libraries to rand location in process memory

    $\Rightarrow$ Attacker cannot jump directly to exec function
  - <u>Deployment</u>: (/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
- <u>Other randomization methods</u>:
  - 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

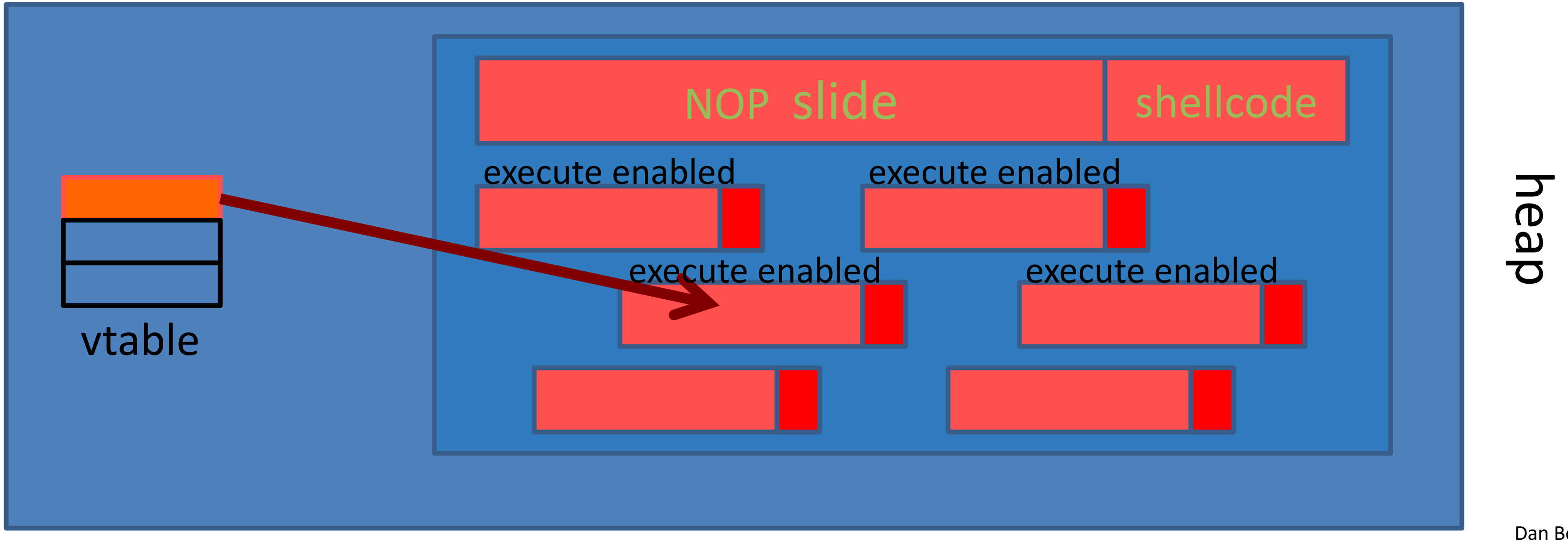
- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

More attacks : JiT spraying

Idea:

1. Force Javascript JiT to fill heap with executable shellcode

2. then point SFP anywhere in spray area



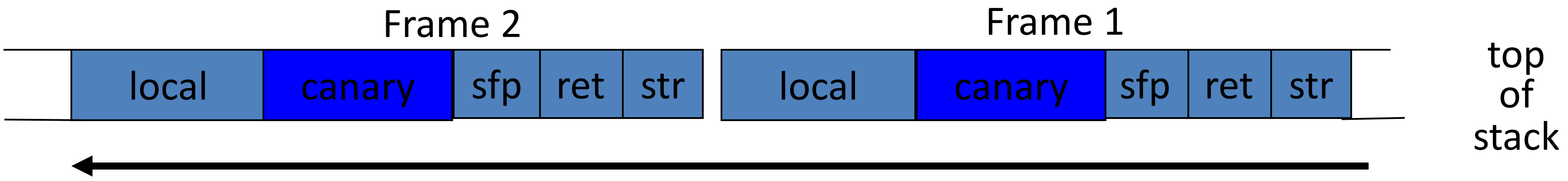
Control Hijacking

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

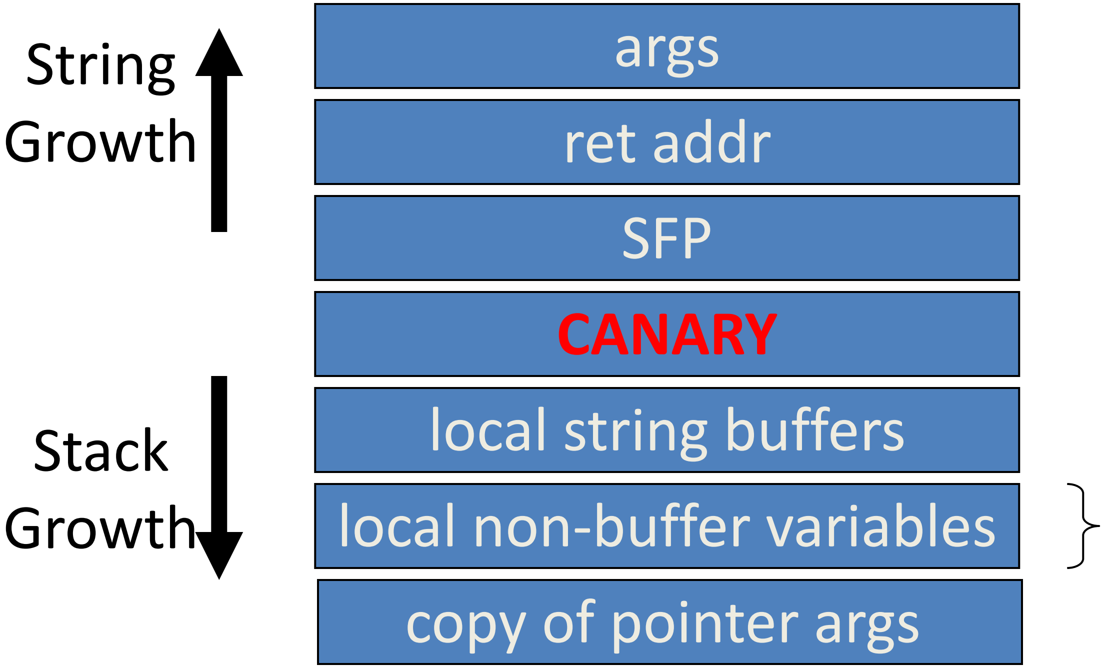
- Minimal performance effects: 8% for Apache

- 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.



Protects pointer args and local pointers from a buffer overflow

pointers, but no arrays

/GS stack frame

String Growth

Stack Growth

args
ret addr
SFP
exception handlers
CANARY
local string buffers
local non-buffer variables
copy of pointer args

Canary protects ret-addr and exception handler frame

pointers, but no arrays

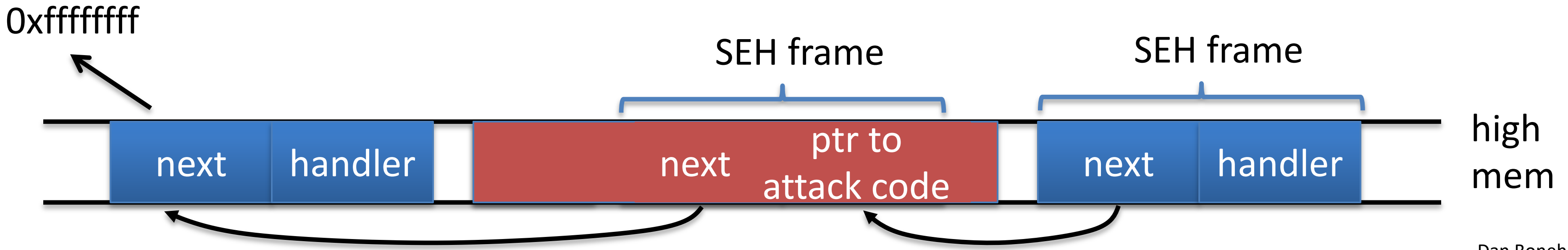
Evading /GS with exception handlers

- 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

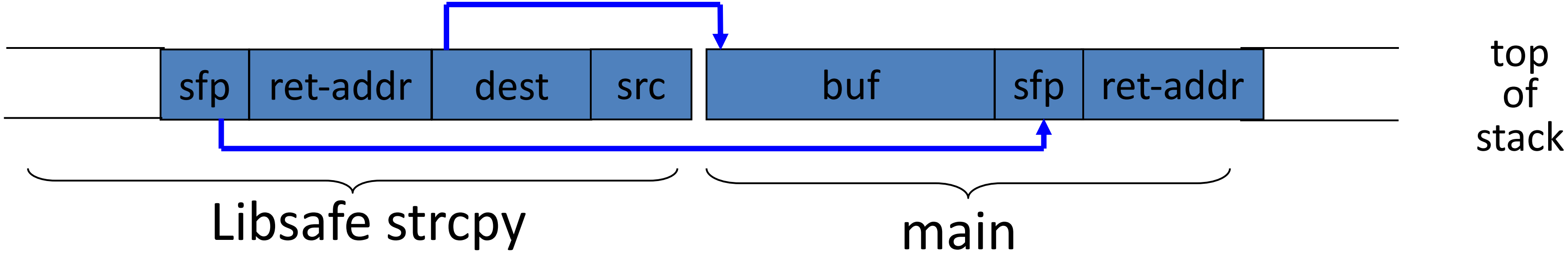
    (also need SAFESEH and SEHOP)

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 …

- **<u>StackShield</u>**
  - 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)

- **<u>Control Flow Integrity</u>** (CFI)
  - A combination of static and dynamic checking
    - Statically determine program control flow
    - Dynamically enforce control flow integrity