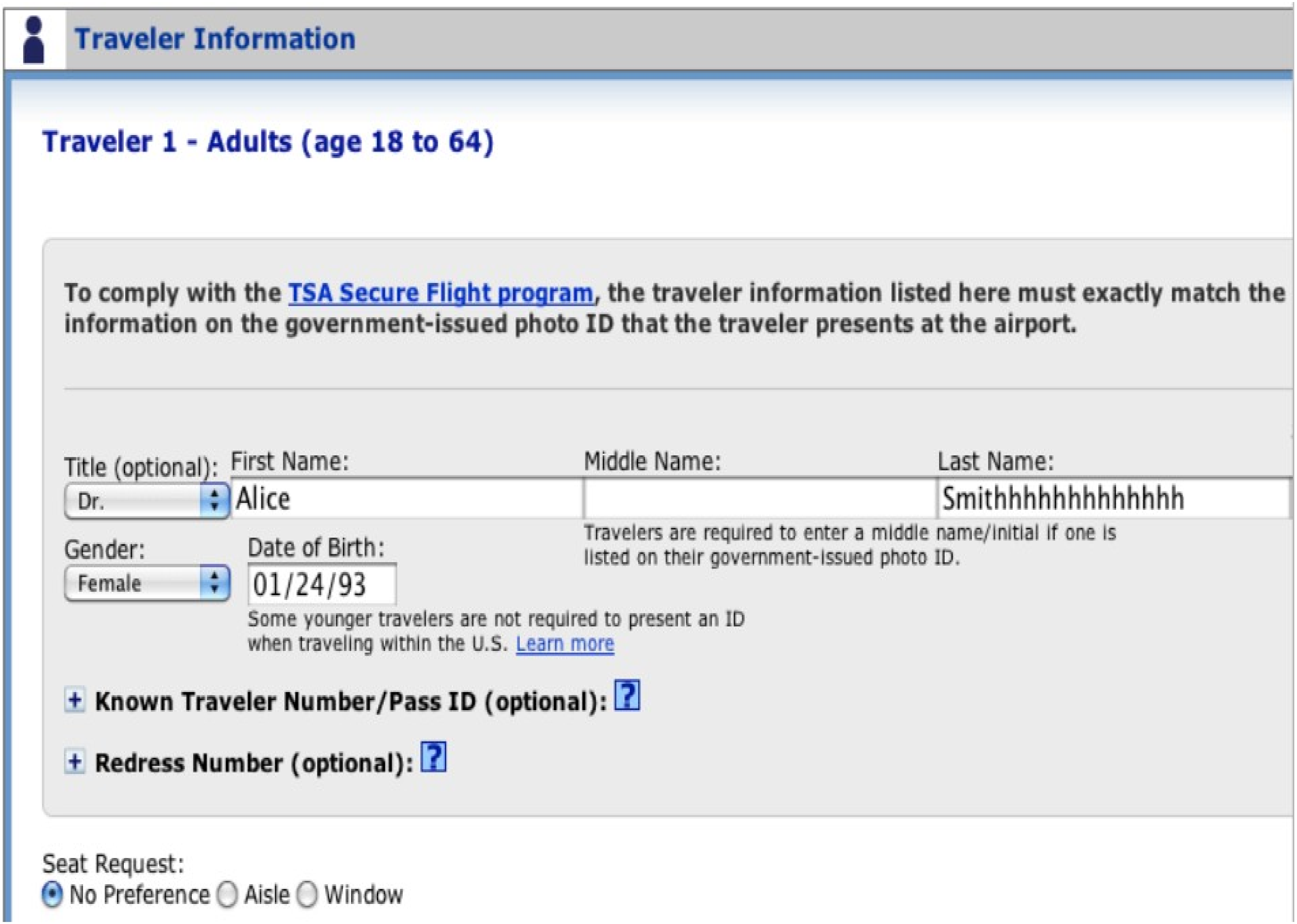
Software Security (II): Other types of software vulnerabilities

Dawn Song

#293 HRE-THR 850 1930
ALICE SMITH
COACH

SPECIAL INSTRUX:
NONE



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.

First Name: Alice

Middle Name:

Last Name: Smithhhhhhhhhhhhh

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

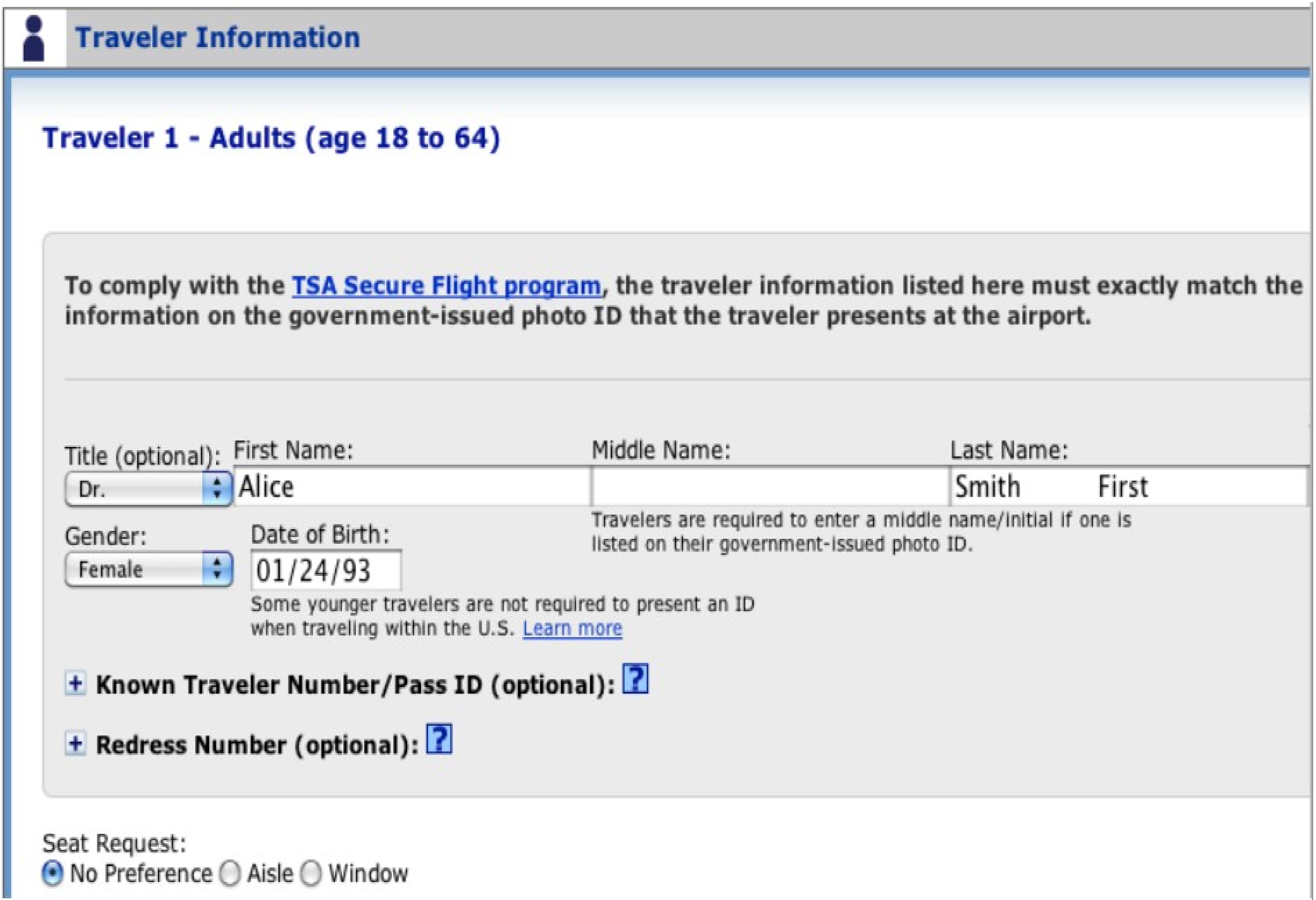
+ Known Traveler Number/Pass ID (optional): ?

+ Redress Number (optional): ?

Seat Request:
● No Preference ○ Aisle ○ Window

```
#293 HRE-THR 850 1930
ALICE SMITHHHHHHHHHHHH
HHACH

SPECIAL INSTRUX: NONE
```



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.
First Name: Alice
Middle Name:
Last Name: Smith First

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female
Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

Seat Request:
● No Preference ○ Aisle ○ Window

```
#293 HRE-THR 850 1930
ALICE SMITH
FIRST

SPECIAL INSTRUX:
NONE
```

Example #1

```
void vulnerable() {
    char name[20];
    …
    gets(name);
    …
}
```

Example #2

```
void vulnerable() {
    char instrux[80] = “none”;
    char name[20];
     …
     gets(name);
     …
}
```

Example #3

```
void vulnerable() {
    char cmd[80];
    char line[512];
     ...
    strncpy(cmd,”/usr/bin/finger”, 80);
      gets(line);
      ...
      execv(cmd, ...);
}
```

Example #4

```
void vulnerable() {
    int (*fnptr)();
   char buf[80];
   …
    gets(buf);
    …
}
```

Example #5

```
void vulnerable() {
     int seatinfirstclass = 0;
    char name[20];
    ...
    gets(name);
    ...
}
```

Example #6

```
void vulnerable() {
      int authenticated = 0;
    char name[20];
    …
     gets(name);
     …
}
```

Common Coding Errors

- Input validation vulnerabilities
- Memory management vulnerabilities
- TOCTTOU vulnerability (later)

Input validation vulnerabilities

- Program requires certain assumptions on inputs to run properly
- Without correct checking for inputs
  - Program gets exploited
- Example:
  - Buffer overflow
  - Format string

Example I

Example I

```
unsigned int size;
Data **datalist;

size = GetUntrustedSizeValue();
datalist = (data **)malloc(size * sizeof(Data *));
for(int i=0; i<size; i++) {
   datalist[i] = InitData();
}
datalist[size] = NULL;
...
```

Example II

Example II

```
char buf[80];
void vulnerable() {
  int len = read_int_from_network();
  char *p = read_string_from_network();
  if (len > sizeof buf) {
    error(“length too large, nice try!”);
    return;
  }
  memcpy(buf, p, len);
}
```

- What's wrong with this code?
- Hint – `memcpy()` prototype:
  - `void *memcpy(void *dest, const void *src, size_t n);`
- Definition of `size_t`: `typedef unsigned int size_t;`
- Do you see it now?

Implicit Casting Bug

- Attacker provides a negative value for `len`
  - `if` won't notice anything wrong
  - Execute `memcpy()` with negative third arg
  - Third arg is implicitly cast to an `unsigned int`, and becomes a very large positive int
  - `memcpy()` copies huge amount of memory into `buf`, yielding a buffer overrun!
- A signed/unsigned or an implicit casting bug
  - Very nasty – hard to spot
- C compiler doesn't warn about type mismatch between `signed int` and `unsigned int`
  - Silently inserts an implicit cast

Example III (Integer Overflow)

Example III

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
...
```

- What's wrong with this code?
  - No buffer overrun problems (5 spare bytes)
  - No sign problems (all ints are unsigned)
- But, `len+5` can overflow if `len` is too large
  - If `len = 0xFFFFFFFF`, then `len+5` is 4
  - Allocate 4-byte buffer then read a lot more than 4 bytes into it: classic buffer overrun!
- Know programming language's semantics well to avoid pitfalls

Example IV

Example IV

```
char* ptr = (char*) malloc(SIZE);
if (err) {
  abrt = 1;
  free(ptr);
}
...
if (abrt) {
  logError(“operation aborted before commit”, ptr);
}
```

- Use-after-free
- Corrupt memory

http://cwe.mitre.org

Example V

Example V

```
char* ptr = (char*) malloc(SIZE);
if (err) {
  abrt = 1;
  free(ptr);
}
...
free(ptr);
```

- Double-free error
- Corrupts memory-management data structure

Example VI: Format string problem

Example VI

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

Format Functions

- Used to convert simple C data types to a string representation
- Variable number of arguments
- Including format string
- Example
  - printf(“%s number %d”, “block”, 2)
  - Output: “block number 2”

Format String Parameters

Parameter	Output	Passed as
%d	Decimal (int)	Value
%u	Unsigned decimal (unsigned int)	Value
%x	Hexadecimal (unsigned int)	Value
%s	String ((const) (unsigned) char *)	Reference
%n	# bytes written so far, (* int)	Reference

Example VI: Format string problem

Example VI

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

- Problem: what if   *user = “%s%s%s%s%s%s %s” ??
  - %s displays memory
  - Likely to read from an illegal address
  - If not, program will print memory contents.

Correct form: `fprintf( stdout, “%s”, user);`

Stack and Format Strings

- Function behavior is controlled by the format string
- Retrieves parameters from stack as requested: “%”
- Example:

printf r %d has no address, number %d has: %08x\n”, I, a, &a)

stack top
…
<&a>
<a>
<i>
A
…
stack bottom

A	Address of the format string
I	Value of variable I
a	Value of variable a
&a	Address of variable a

View Stack

- printf(“%08x. %08x. %08x. %08x\n”)
  - 40012983.0806ba43.bfffff4a.0802738b

- display 4 values from stack

Read Arbitrary Memory

- char input[] = “\x10\x01\x48\x08_%08x. %08x. %08x. %08x|%s|”;

  printf(input)

  - Will display memory from 0x08480110

- Uses reads to move stack pointer into format string

- %s will read at 0x08480110 till it reaches null byte

Writing to arbitrary memory

- printf(“hello %n”, &temp)
  - writes ‘6’ into temp.

- printf(“%08x.%08x.%08x.%08x.%n”)

Vulnerable functions

Any function using a format string.

Printing:

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

Logging:

- syslog, err, warn

An Exploit Example

```
syslog(“Reading username:”);
read_socket(username);
syslog(username);
```

⇓

```
Welcome to InsecureCorp. Please login.
Login: EvilUser%s%s…%400n…%n
root@server> _
```

Why The Bug Exists

- C language has poor support for variable-argument functions
  - Callee doesn’t know the number of actual args
- No run-time checking for consistency between format string and other args
- Programmer error

Real-world Vulnerability Samples

- 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

⋮

What are software vulnerabilities?

- Flaws in software
- Break certain assumptions important for security
  - E.g., what assumptions are broken in buffer overflow?

Why does software have vulnerabilities?

- Programmers are humans!
  - Humans make mistakes!
- Programmers are not security-aware

- Programming languages are not designed well for security

What can you do?

- Programmers are humans!
  - Humans make mistakes!
  - Use tools! (next lecture)
- Programmers were not security aware
  - Learn about different common classes of coding errors
- Programming languages are not designed well for security
  - Pick better languages

Software Security (III): Defenses against Memory-Safety Exploits

Preventing hijacking attacks

Fix bugs:

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

Allow overflow, but prevent code execution

Add runtime code to detect overflows exploits:

- Halt process when overflow exploit detected
- StackGuard, Libsafe

Control-hijacking Attack Space

Defenses/Mitigations	Code Injection	Arc Injection
Stack		
Heap		
Exception Handlers		

Defense I: 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)
- <u>Deployment</u>:
  - Linux (via PaX project); OpenBSD
  - Windows: since XP SP2 (DEP)
    - Boot.ini : **/noexecute=OptIn** or **AlwaysOn**
    - Visual Studio: **/NXCompat[:NO]**

Effectiveness and Limitations

- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against exploits using return-oriented programming

Defer

ations

	Code Injection	Arc Injection
Stack	**Non-Execute (NX)***	
Heap	**Non-Execute (NX)***	
Exception Handlers	**Non-Execute (NX)***	
Handlers		

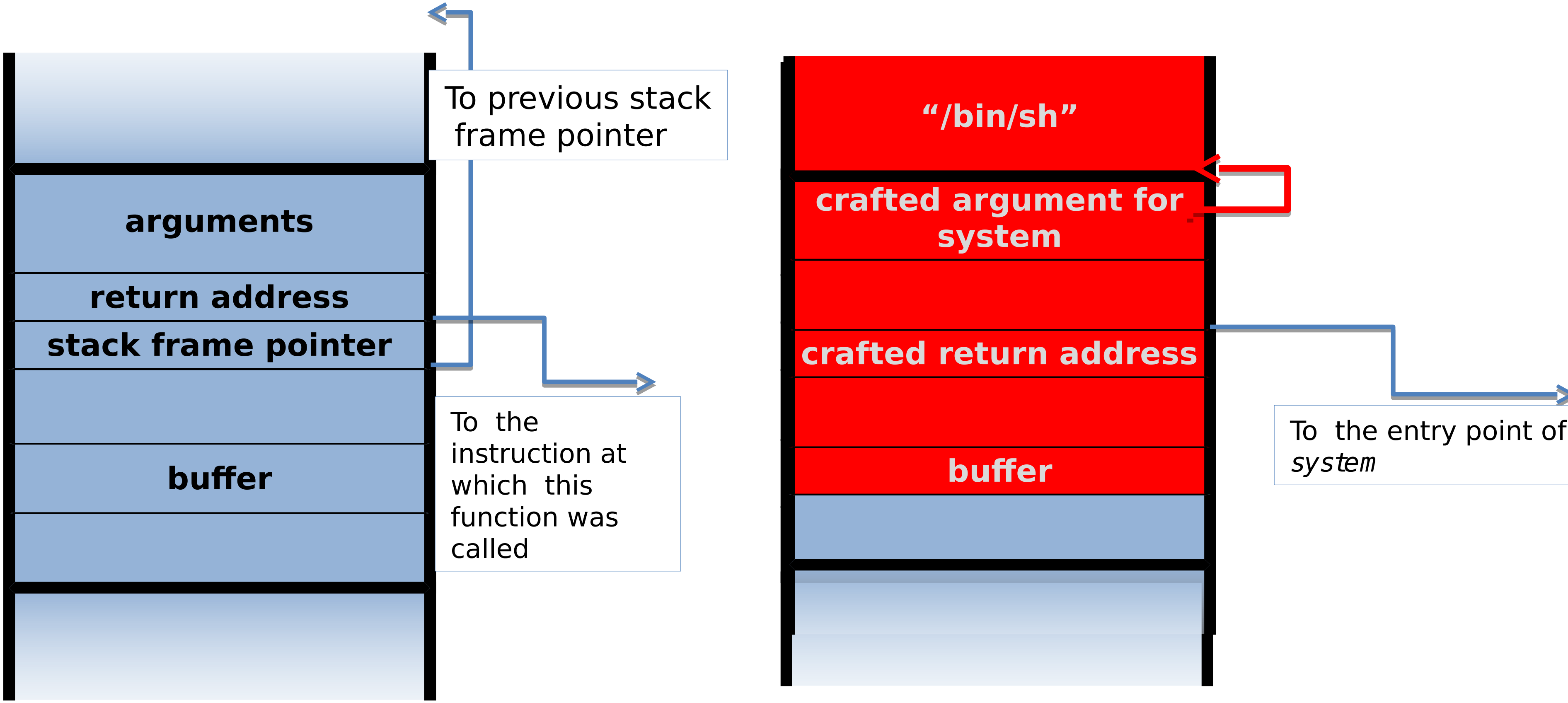
* When Applicable

Return-Oriented Programming (ROP)

- ret2lib exploits
  - Reuse existing functions, no code injection required

Ret-2-lib Exploit

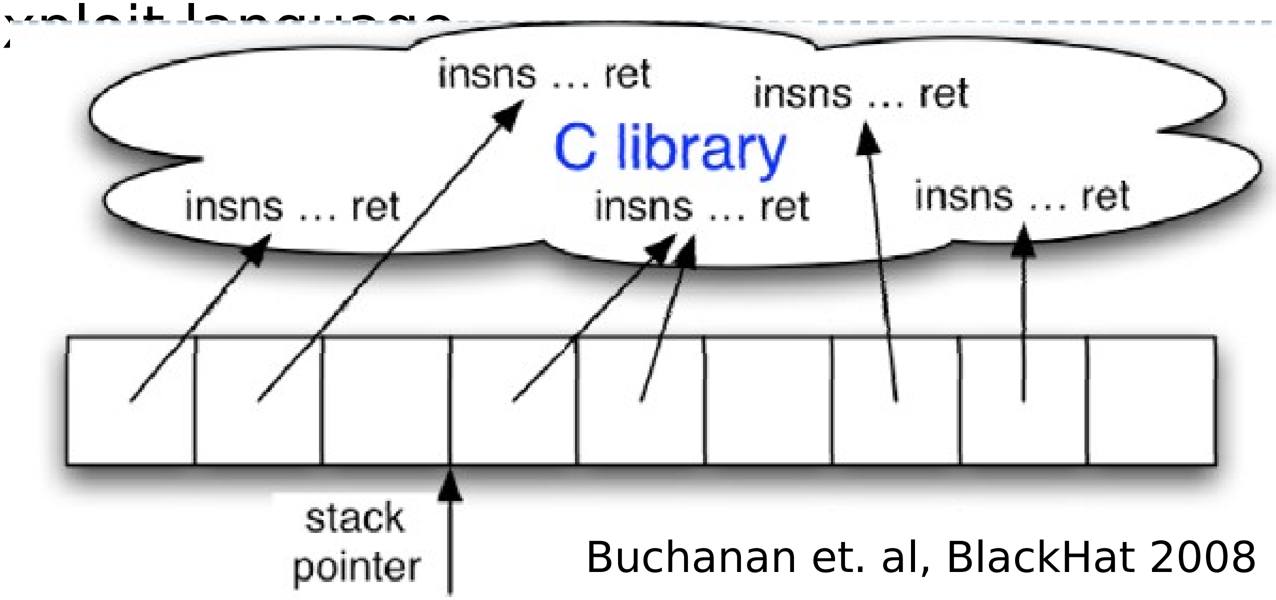
So suppose we want to spawn a shell by exploiting a buffer overflow vulnerability: **Shell Code:** system("/bin/sh")



When the function exits, it returns to the entry of the libc function *system* .
With the crafted argument, the user gets a shell !!!

Return-Oriented Programming (ROP)

- ret2lib exploits
  - Reuse existing functions, no code injection required
- Return-oriented programming
  - Reuses existing code chunks (called *gadgets*)
  - The gadgets could provide a Turing-complete exploit language



Buchanan et. al, BlackHat 2008

Defense II: Address Randomization

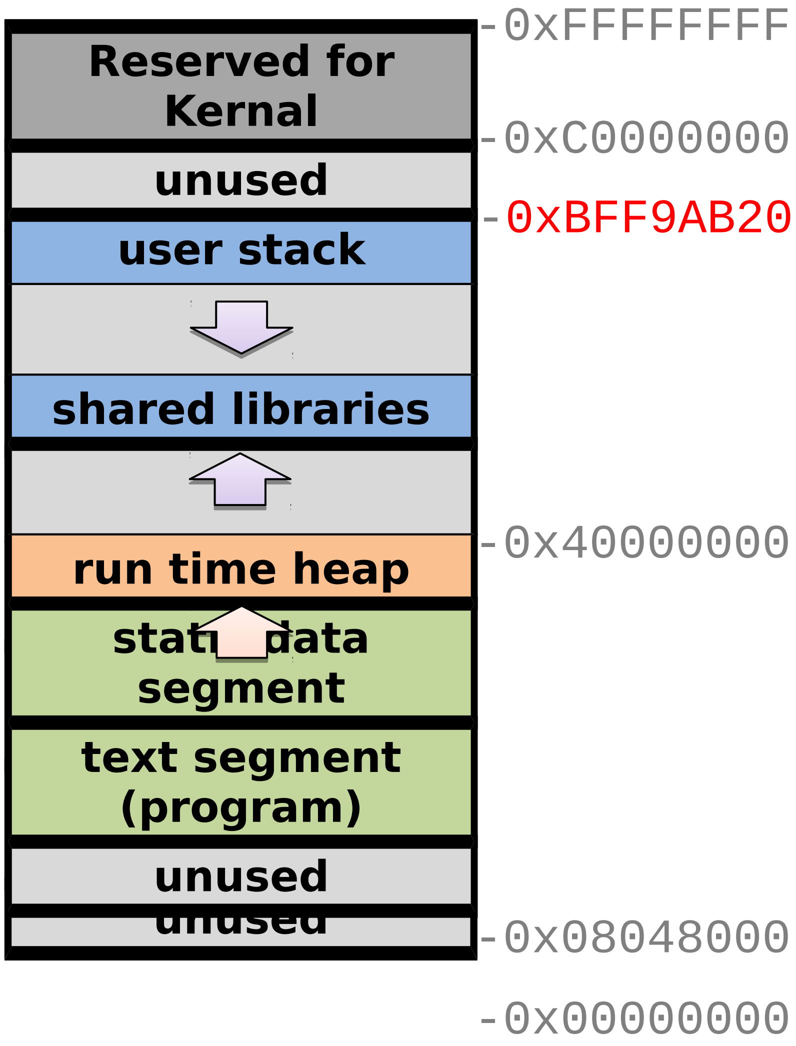
ASLR: (Address Space Layout Randomization)

- Start stack at a random location
- Start heap at a random locatioin
- Map shared libraries to rand location in process memory

  ⇒ 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 ⇒ 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
- Instruction Set Randomization (ISR)



Reserved for Kernal

unused

user stack

shared libraries

run time heap

static data segment

text segment (program)

unused

unused

0xFFFFFFFF

0xC0000000

0xBFF9AB20

0x40000000

0x08048000

0x00000000

Effectiveness and Limitations

- Limitations
  - Randomness is limited
  - Some vulnerabilities can allow secret to be leaked

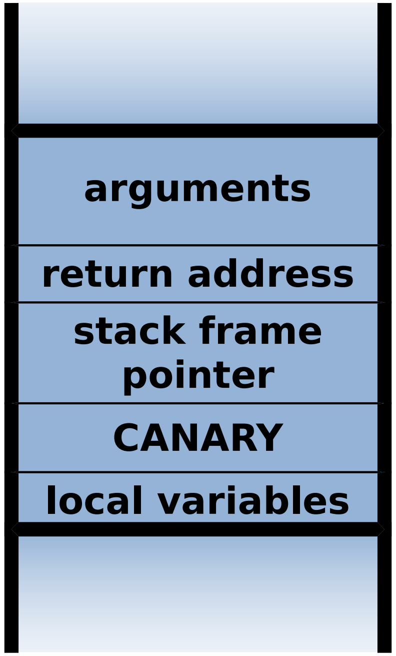
Defenses/Mitigations	Code Injection	Arc Injection
Stack	Non-Execute (NX)* **ASLR**	**ASLR**
Heap	Non-Execute (NX)* **ASLR**	**ASLR**
Exception Handlers	Non-Execute (NX)* **ASLR**	**ASLR**

* When Applicable

Defense III: 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 exploit successfully, 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.

- Low performance effects: 8% for Apache.

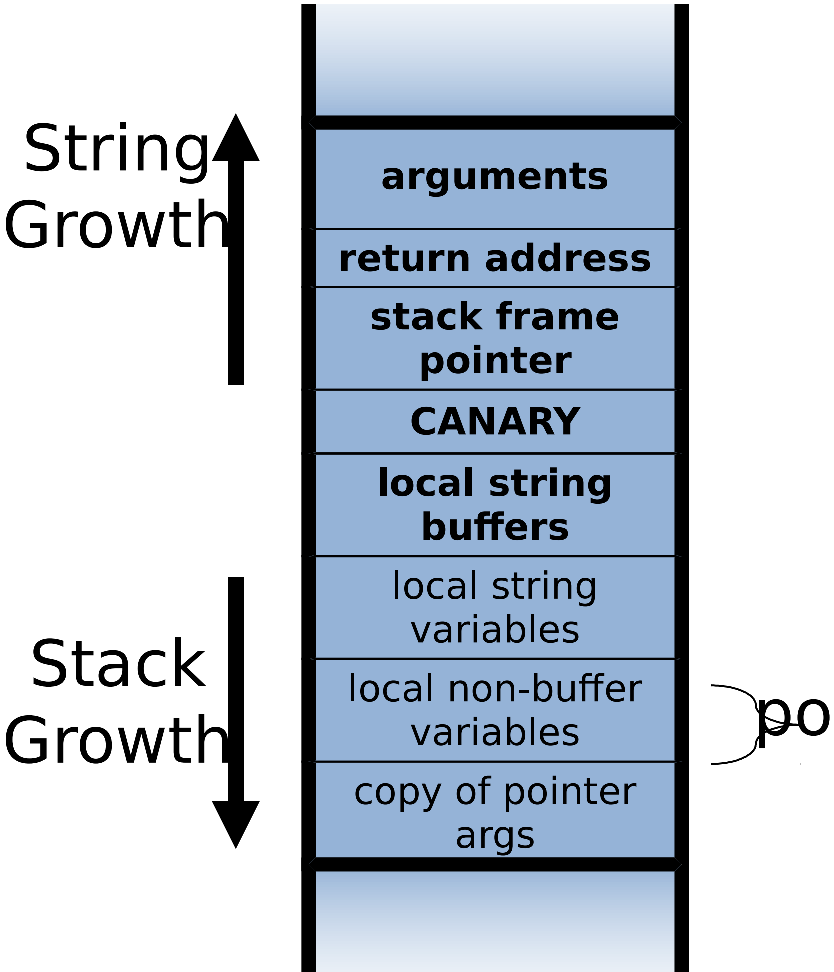
- Note: Canaries don't provide full proof protection.
  - Some stack smashing attacks leave canaries unchanged

- 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.



pointers, but no arrays

Protects pointer args and local pointers from a buffer overflow

MS Visual Studio /GS [since 2003]

Compiler /GS option:

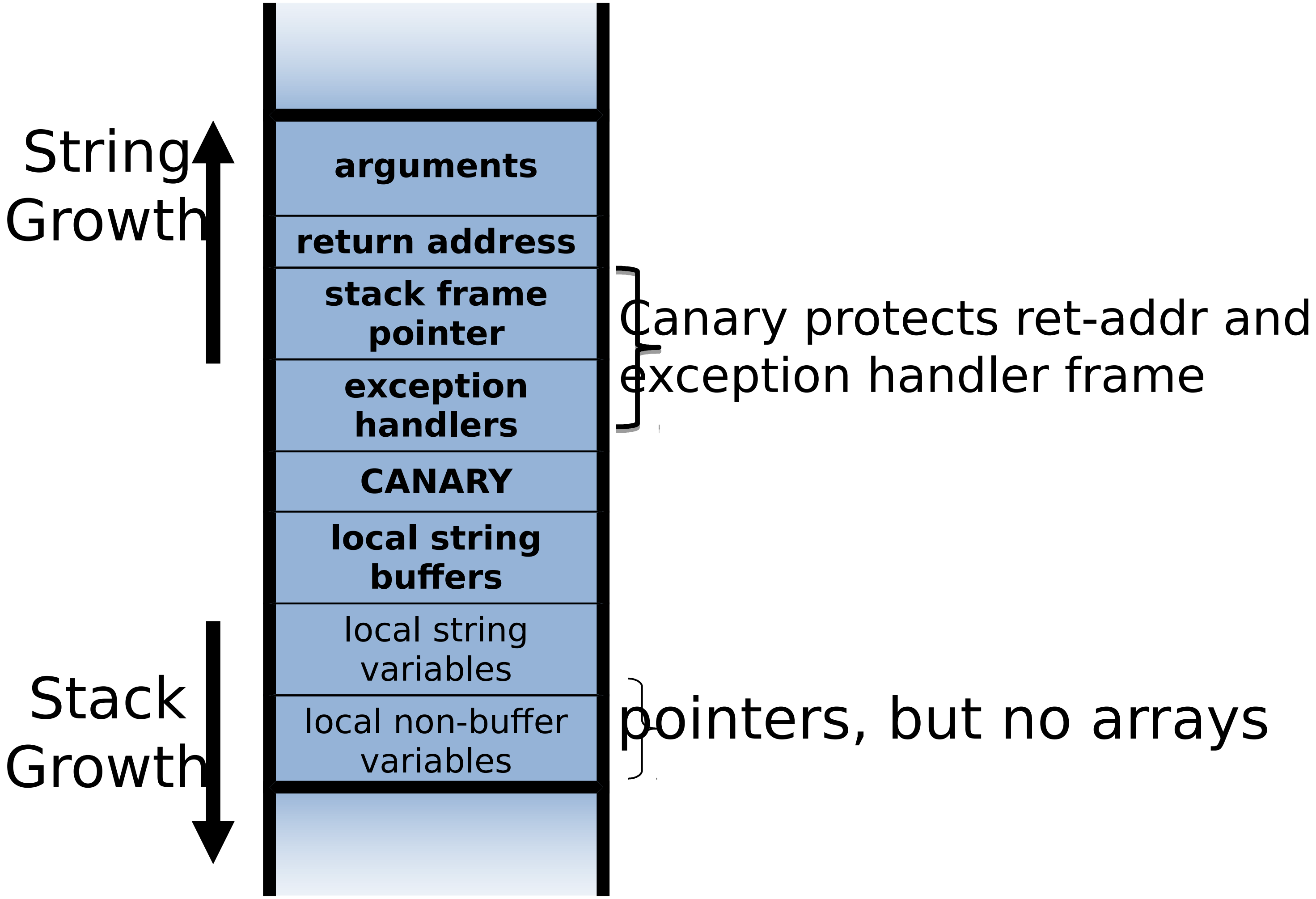
- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call **_exit(3)**

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

Enhanced /GS in Visual Studio 2010:

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

/GS stack frame



String Growth

arguments

return address

stack frame pointer

exception handlers

CANARY

local string buffers

local string variables

local non-buffer variables

Stack Growth

Canary protects ret-addr and exception handler frame

pointers, but no arrays

Effectiveness and Limitations

- Limitation:
  - Evasion with exception handler

* When Applicable

Defenses/Mitigations	Code Injection	Arc Injection
Stack	Non-Execute (NX)* ASLR **StackGuard(Canaries)** **ProPolice** **/GS**	ASLR **StackGuard(Canaries)** **ProPolice** **/GS**
Heap	Non-Execute (NX)* ASLR **PointGuard**	ASLR **PointGuard**
Exception Handlers	Non-Execute (NX)* ASLR	ASLR

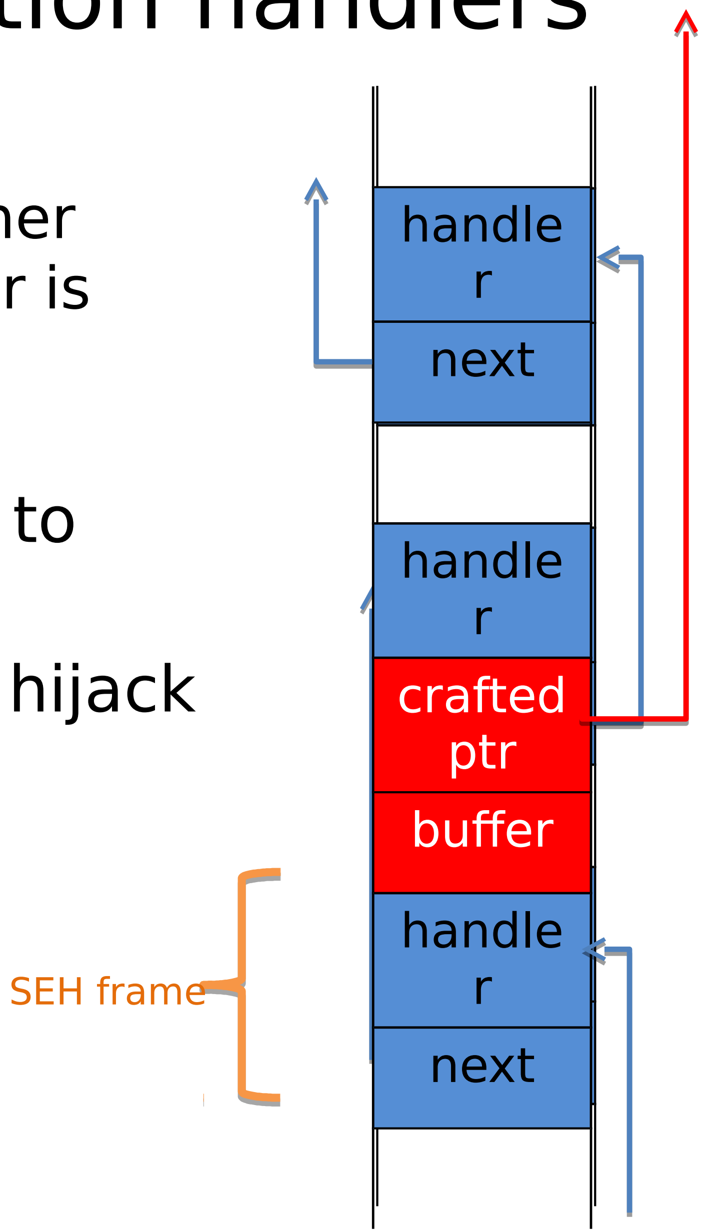
Evading /GS with exception handlers

- 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



Defense III: 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.

Effectiveness and Limitations

- Limitations:
  - Require recompilation

* When Applicable

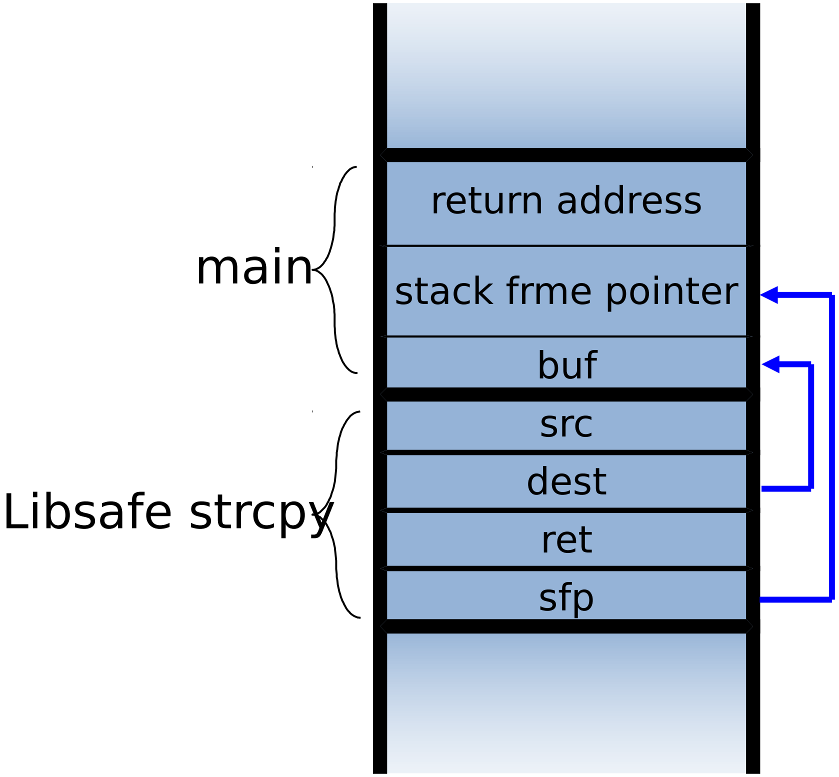
Defenses/Mitigations	Code Injection	Arc Injection
Stack	Non-Execute (NX)* ASLR StacKGuard(Canaries) ProPolice /GS	ASLR StacKGuard(Canaries) ProPolice /GS
Heap	Non-Execute (NX)* ASLR PointGuard	ASLR PointGuard
Exception Handlers	Non-Execute (NX)* ASLR **SAFESEH and SEHOP**	ASLR **SAFESEH and SEHOP**

Defense IV: Libsafe

- 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



Effectiveness and Limitations

- Limitations:
  - Limited protection

* When Applicable

Defenses/Mitigations	Code Injection	Arc Injection
Stack	Non-Execute (NX)* ASLR StacKGuard(Canaries) ProPolice /GS **libsafe**	ASLR StacKGuard(Canaries) ProPolice /GS **libsafe**
Heap	Non-Execute (NX)* ASLR PointGuard	ASLR PointGuard
Exception Handlers	Non-Execute (NX)* ASLR SAFESEH and SEHOP	ASLR SAFESEH and SEHOP

Other Defenses

- **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

Effectiveness and Limitations

- Many different kinds of attacks. Not one silver bullet defense.

* When Applicable

Defenses/Mitigations	Code Injection	Arc Injection
Stack	Non-Execute (NX)* ASLR StacKGuard(Canaries) ProPolice /GS **li**bsafe **StackShield**	ASLR StacKGuard(Canaries) ProPolice /GS libsafe **StackShield**
Heap	Non-Execute (NX)* ASLR PointGuard	ASLR PointGuard
Exception Handlers	Non-Execute (NX)* ASLR SAFESEH and SEHOP	ASLR SAFESEH and SEHOP