Software Security (II): Buffer-overflow Defenses

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)
- Deployment:
  - 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 `**return-to-libc**' exploits

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

* When Applicable

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

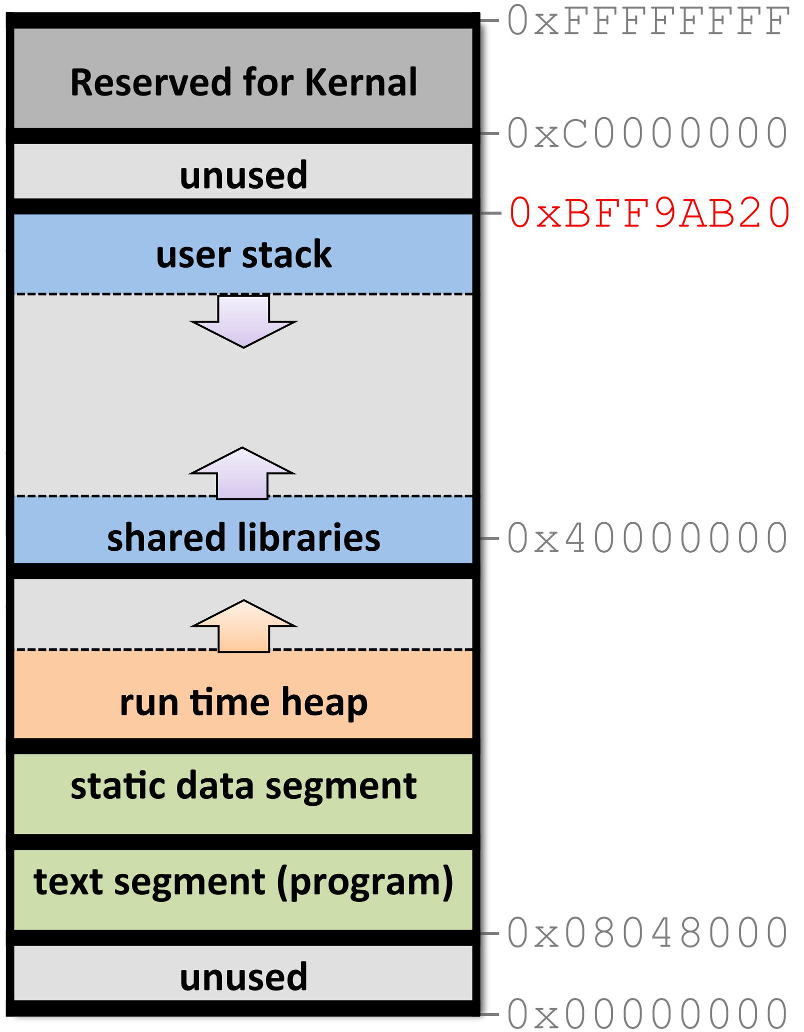
– Deployment: (/DynamicBase)
  - **Windows** Vista: 8 bits of randomness for DLLs
    – aligned to 64K page in a 16MB region ⇒ 256 choices
  - **Linux** (via PaX): 16 bits of randomness for libraries

– More effective on 64-bit architectures

Other randomization methods:

– Sys-call randomization: randomize sys-call id's

– Instruction Set Randomization (ISR)



Effectiveness and Limitations

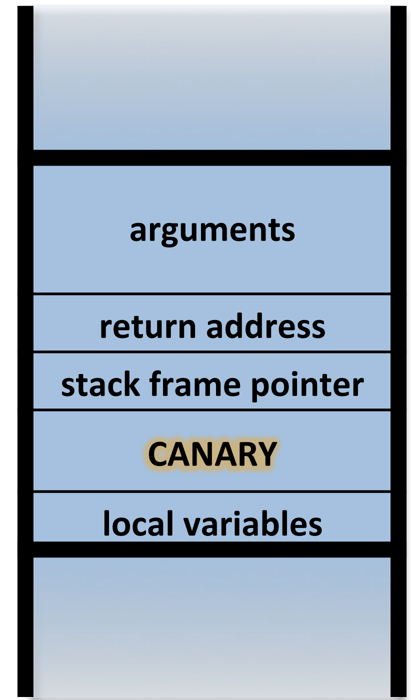
- Limitations
  - Randomness is limited
  - Some vulnerabilities can allow secret to be leaked

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.

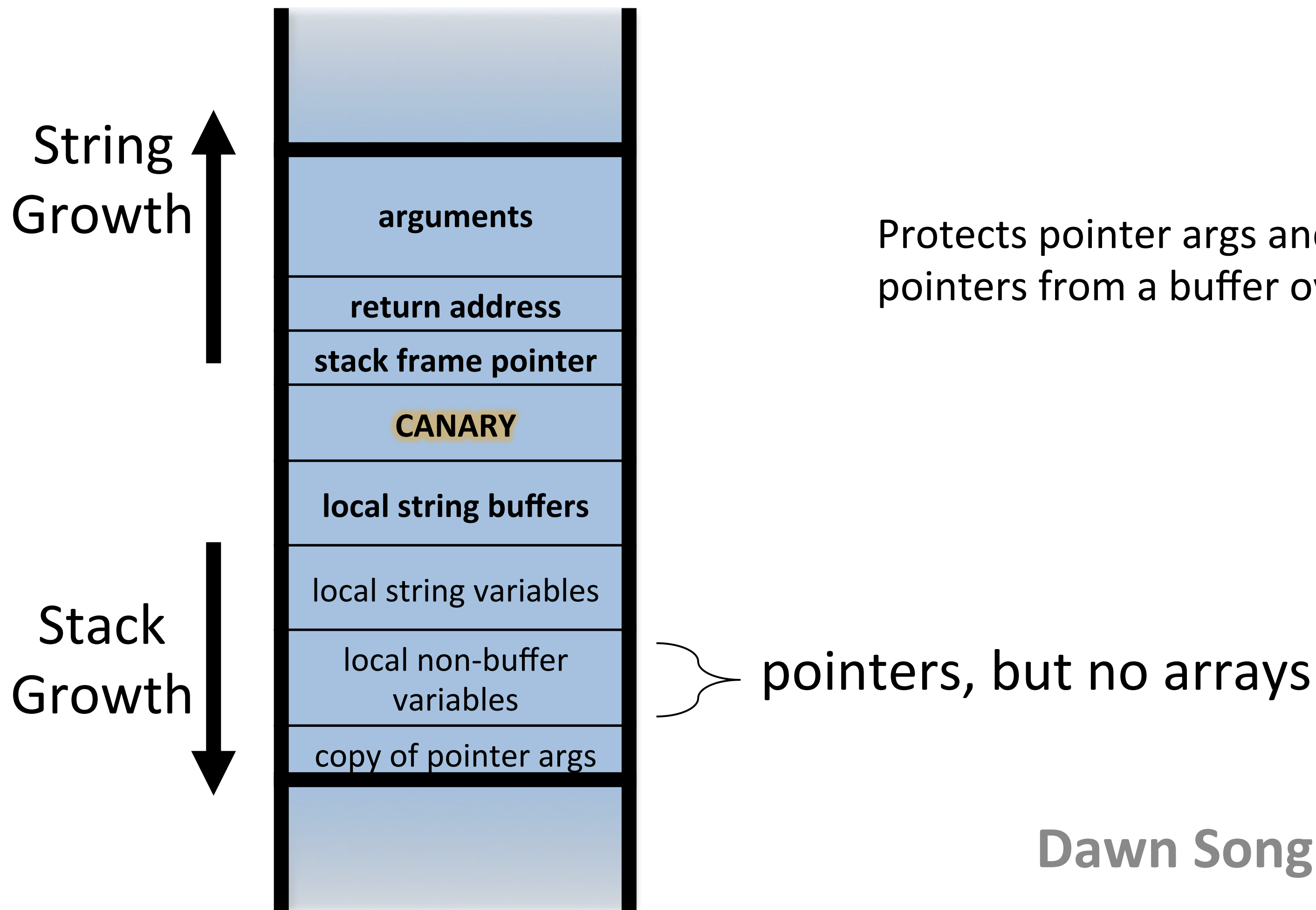
- Terminator canary: Canary = {0, newline, linefeed, EOF}
  - String functions will not copy beyond terminator.
  - Attacker cannot use string functions to corrupt stack.

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.



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)**

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
```

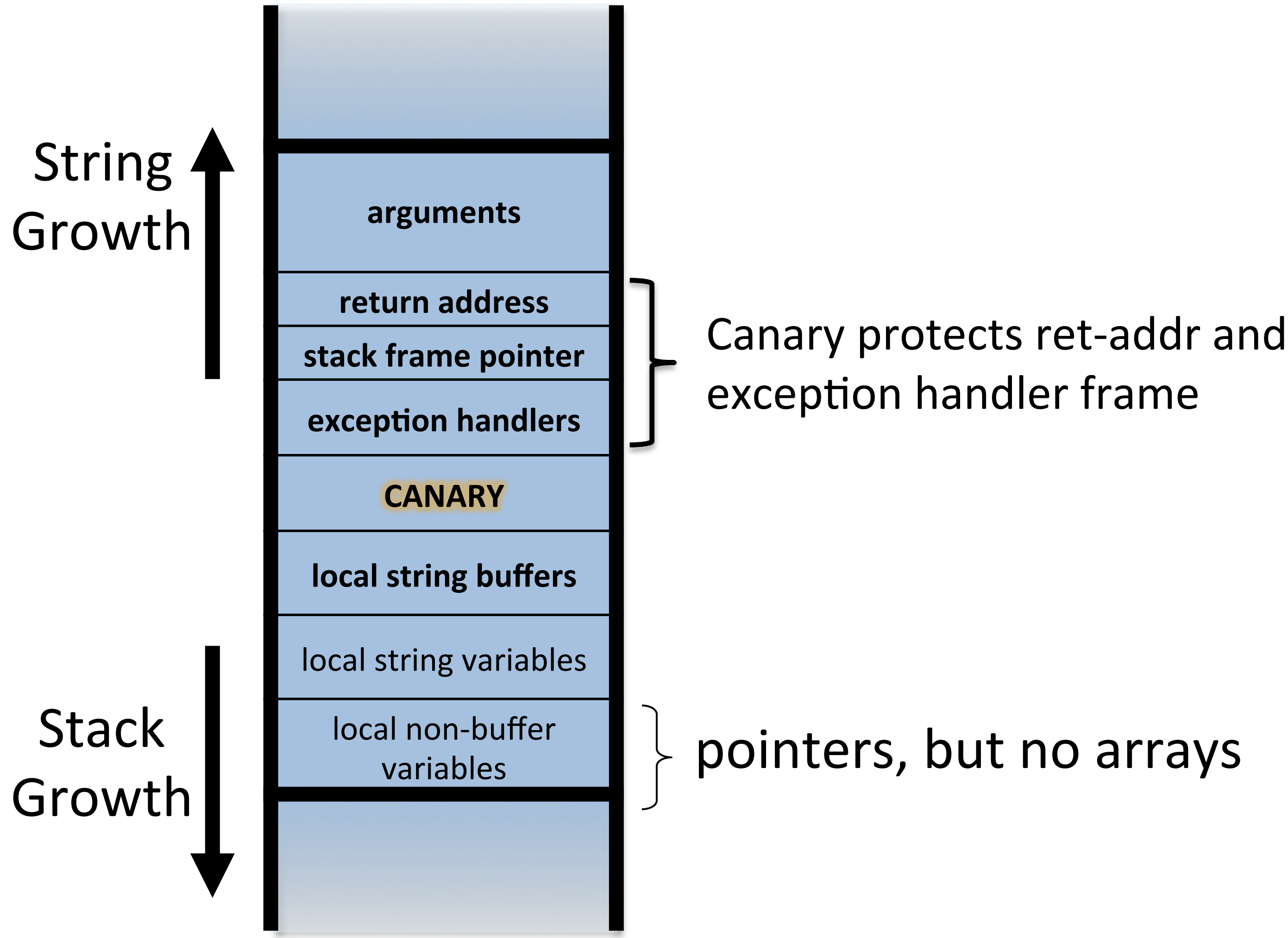
Function epilog:

```
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



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

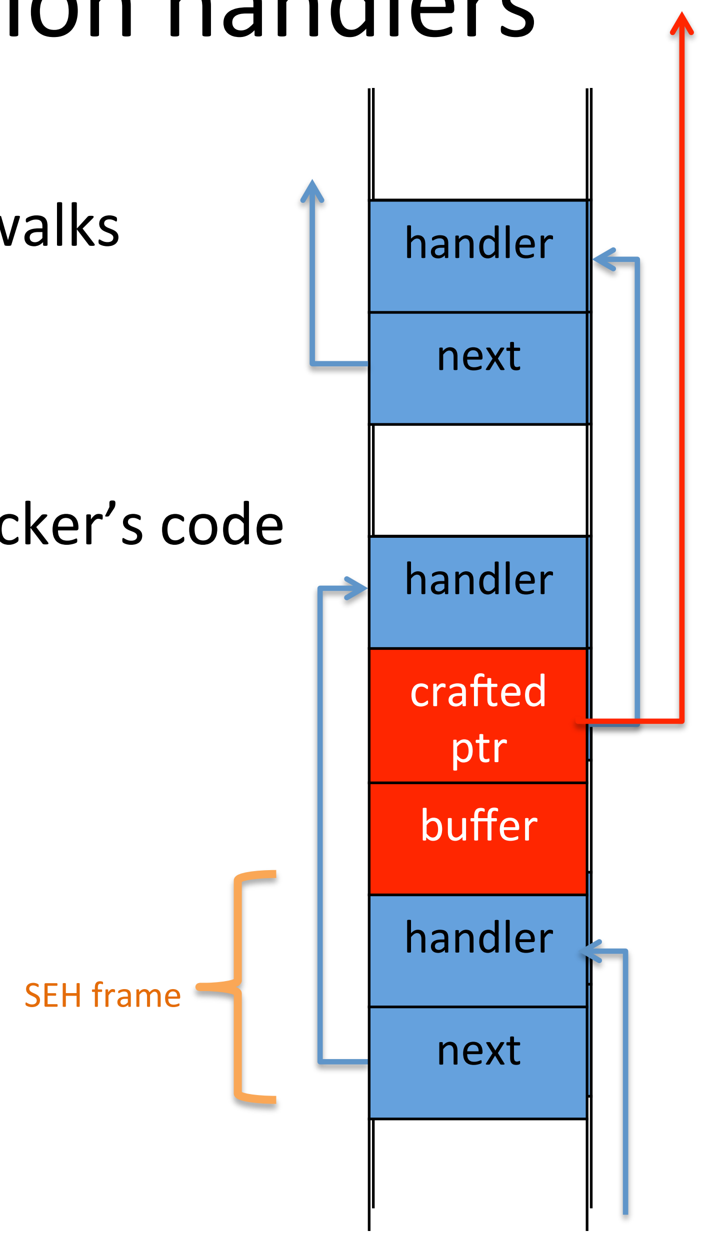
Evading /GS with exception handlers

- 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

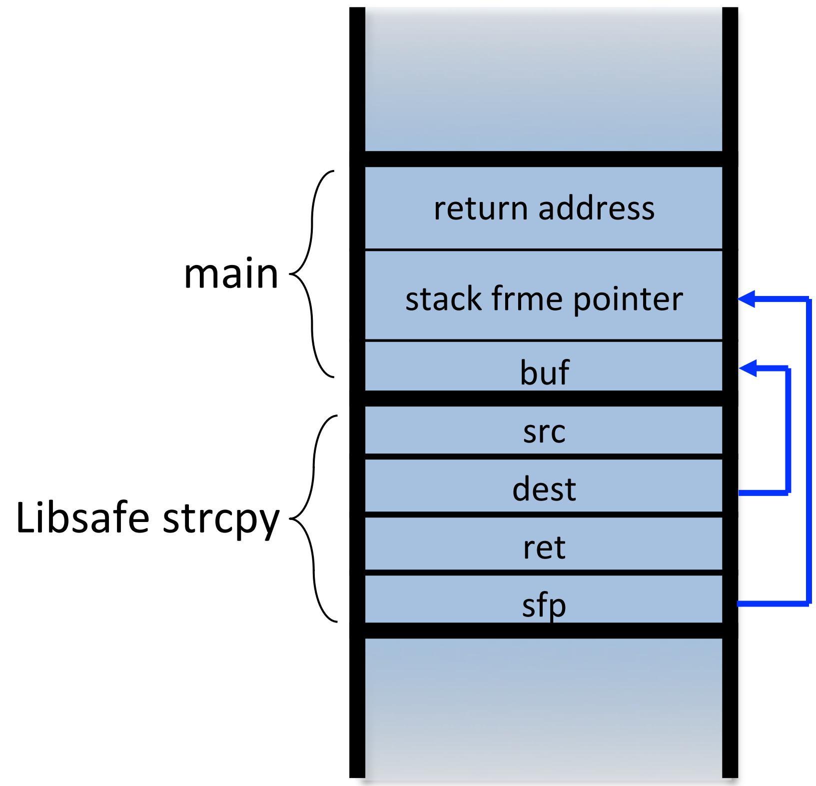
- Limitations:
  - Require recompilation

* 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 /GSl ibsafe **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

Software Security (III):
Other types of software vulnerabilities

Common Coding Errors

- Input validation vulnerabilities

- Memory management vulnerabilities

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 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 II (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 III

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 IV

Example V

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

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