Control Hijacking: Defenses

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

Recap: control hijacking attacks

Stack smashing: overwrite return address or function pointer

Heap spraying: reliably exploit a heap overflow

Use after free: attacker writes to freed control structure, which then gets used by victim program

Integer overflows

Format string vulnerabilities

⋮

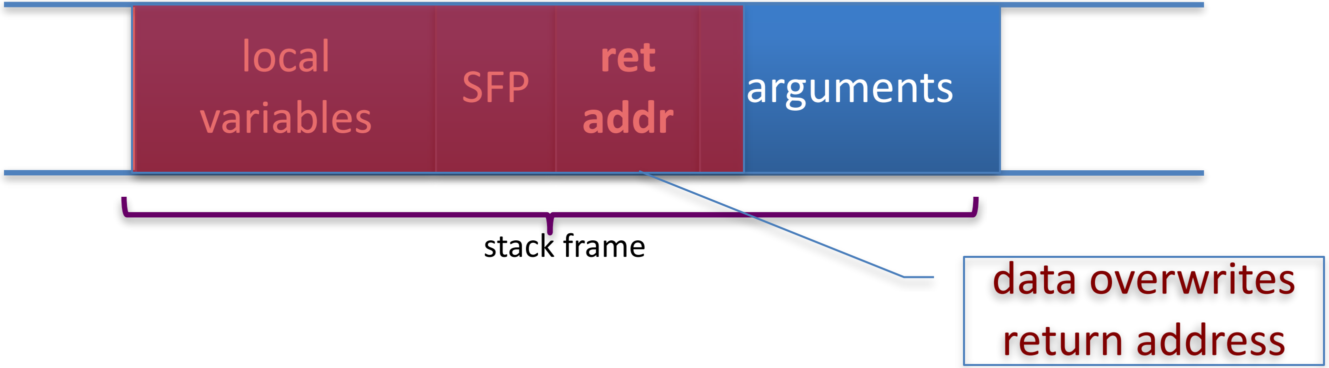
The mistake: mixing data and control

- An ancient design flaw:
  - enables anyone to inject control signals

- 1971: AT&T learns never to mix control and data

Control hijacking attacks

The problem: mixing data with control flow in memory



Later we will see that mixing data and code is also the reason for XSS, a common web vulnerability

Preventing hijacking attacks

1. Fix bugs:
   - Audit software
     - Automated tools: Coverity, Infer, … (more on this next week)
   - Rewrite software in a type safe languange (Java, Go, Rust)
     - Difficult for existing (legacy) code …

2. Platform defenses: prevent attack code execution

3. Harden executable to detect control hijacking
   - Halt process and report when exploit detected
   - StackGuard, ShadowStack, Memory tagging, …

Transform:

Complete Breach

↓

Denial of service

Control Hijacking

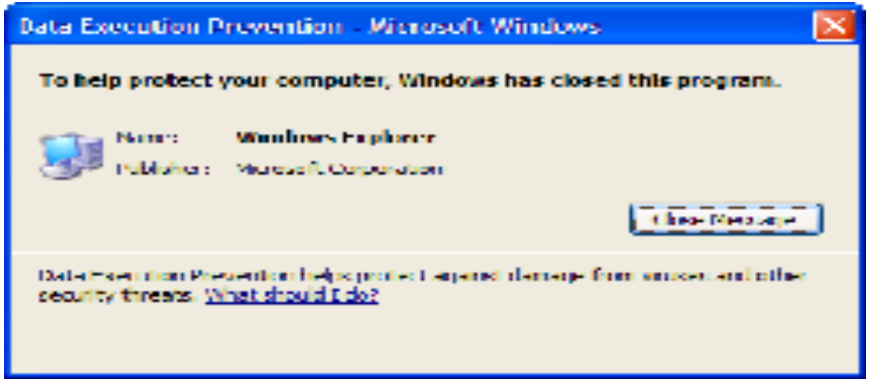
Platform Defenses

Marking memory as non-execute (DEP)

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

- **NX-bit** on AMD64, **XD-bit** on Intel x86 (2005), **XN-bit** on ARM
  - disable execution: an attribute bit in every Page Table Entry (PTE)
- Deployment:
  - All major operating systems
    - Windows DEP: since XP SP2 (2004)
      - Visual Studio: **/NXCompat[:NO]**
- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Can be easily bypassed using **Return Oriented Programming (ROP)**

Examples: DEP controls in Windows



Data Execution Prevention - Microsoft Windows

To help protect your computer, Windows has closed this program.

Name: Windows Explorer

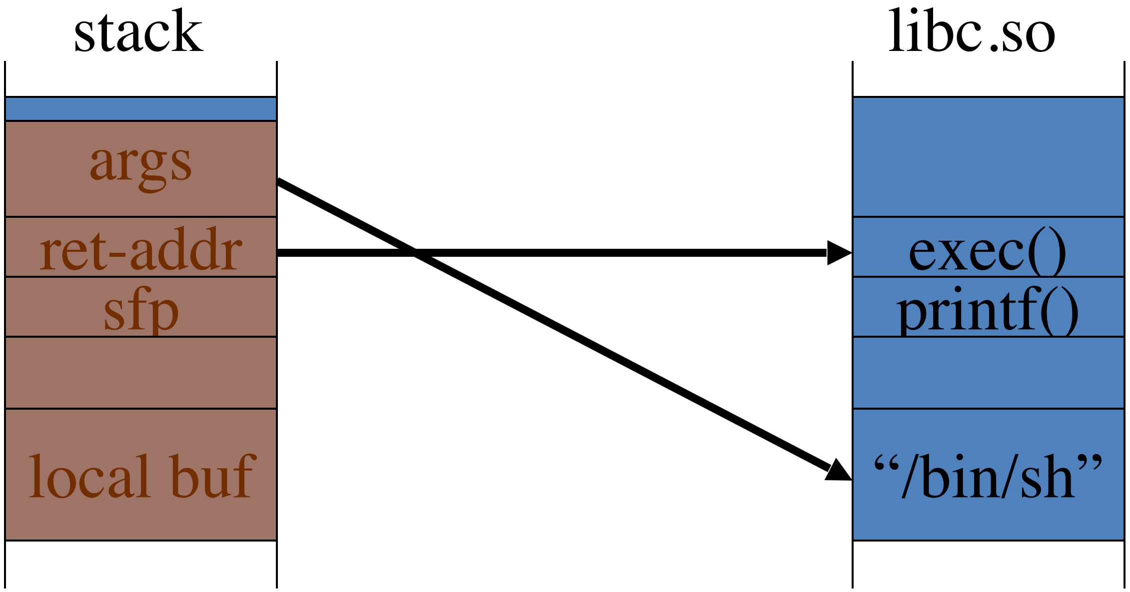
Publisher: Microsoft Corporation

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

DEP terminating a program

Attack: Return Oriented Programming (ROP)

Control hijacking without injecting code:



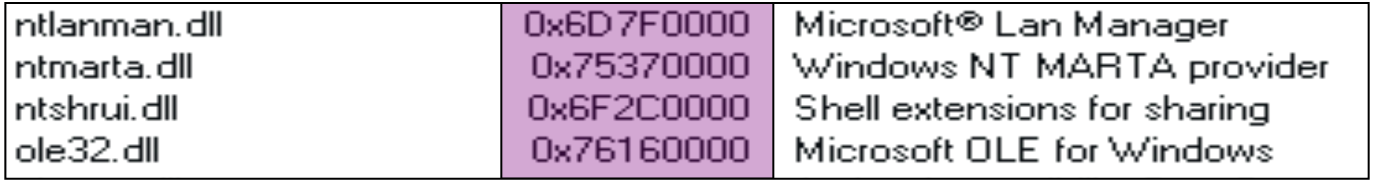
What to do?? Randomization

- **ASLR**: (Address Space Layout Randomization)
  - Randomly shift location of all code in process memory

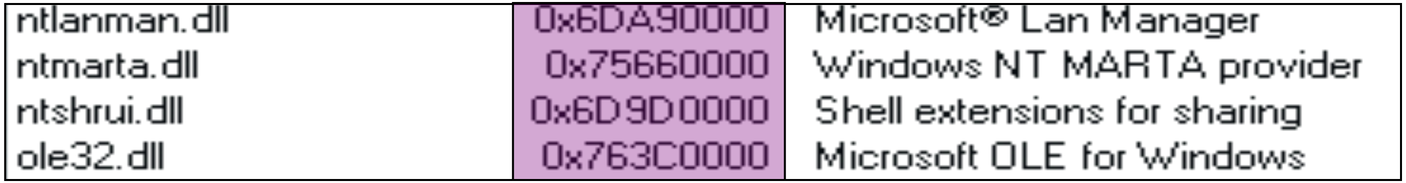
    ⇒ Attacker cannot jump directly to exec function
  - Deployment: (/DynamicBase)
    - **Windows 7**: 8 bits of randomness for DLLs
      - aligned to 64K page in a 16MB region ⇒ 256 choices
    - **Windows 8:** 24 bits of randomness on 64-bit processors
- Other randomization ideas (not used in practice):
  - 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 randomly shifted

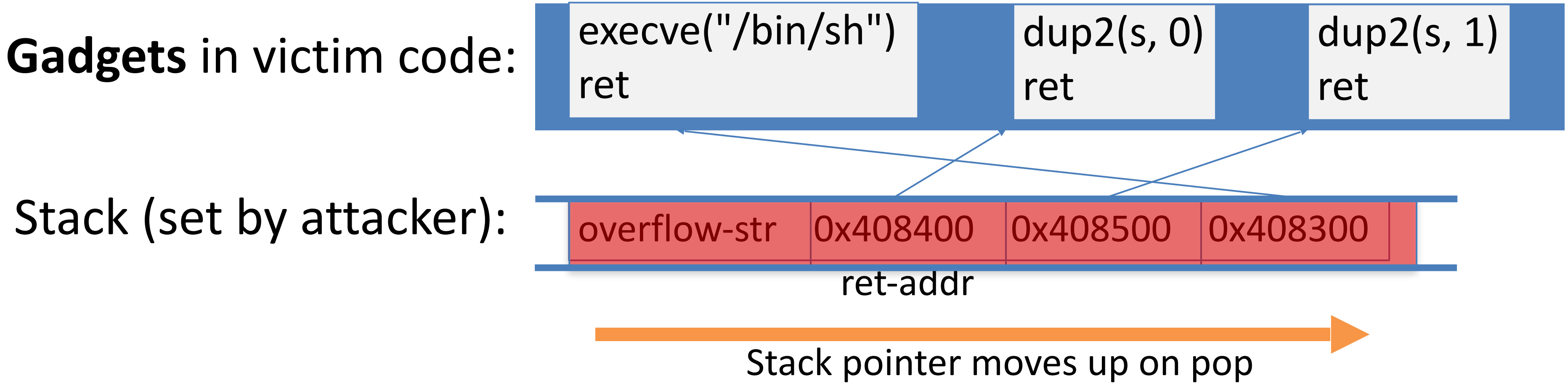
stack, heap, shared libs, base image

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

ROP: in more detail

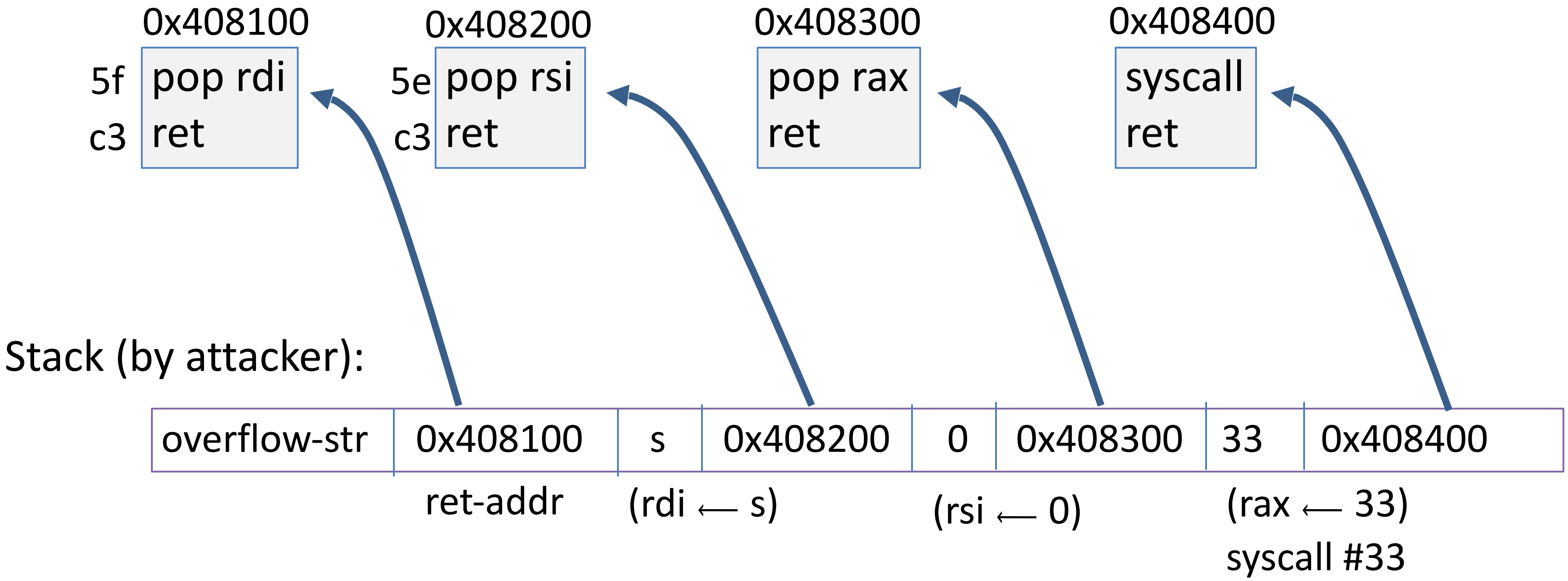
To run /bin/sh we must direct ***stdin*** and ***stdout*** to the socket:

```
dup2(s, 0)          // map stdin to socket
dup2(s, 1)          // map stdout to socket
execve("/bin/sh", 0, 0);
```



ROP: in even more detail

dup2(s,0) implemented as a sequence of gadgets in victim code:



0x408100

5f pop rdi
c3 ret

0x408200

5e pop rsi
c3 ret

0x408300

pop rax
ret

0x408400

syscall
ret

Stack (by attacker):

overflow-str	0x408100	s	0x408200	0	0x408300	33	0x408400
	ret-addr	(rdi ⟵ s)		(rsi ⟵ 0)		(rax ⟵ 33) syscall #33	

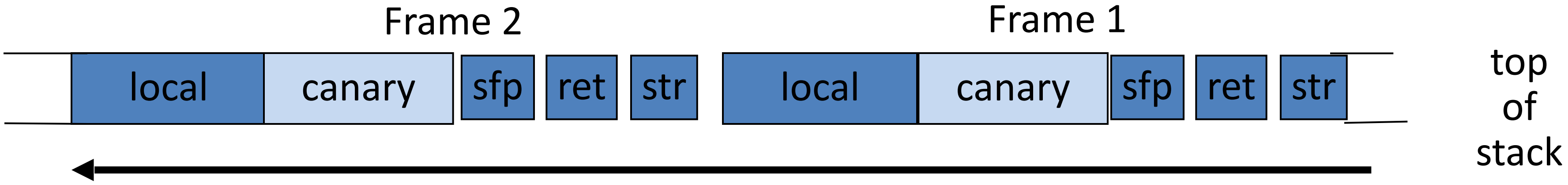
Control Hijacking Defenses

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.

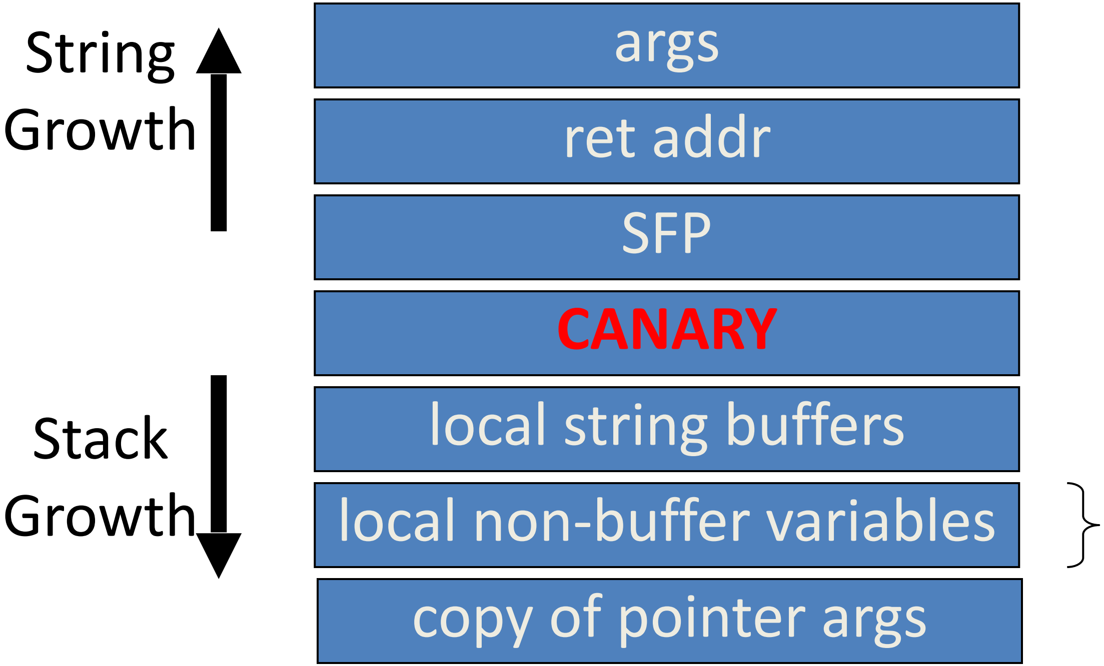
StackGuard (Cont.)

- StackGuard implemented as a GCC patch
  - Program must be recompiled

- Minimal performance effects: 8% for Apache

StackGuard enhancement: ProPolice

- ProPolice - since 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

Summary: Canaries are not full proof

- Canaries are an important defense tool, but do not prevent all control hijacking attacks:
  - Some stack smashing attacks leave canaries unchanged: how?
  - Heap-based attacks still possible
  - Integer overflow attacks still possible

Even worse: canary extraction

A common design for crash recovery:

- When process crashes, restart automatically (for availability)
- Often canary is unchanged (reason: relaunch using fork)

Danger:

- canary extraction byte by byte

Similarly: extract ASLR randomness

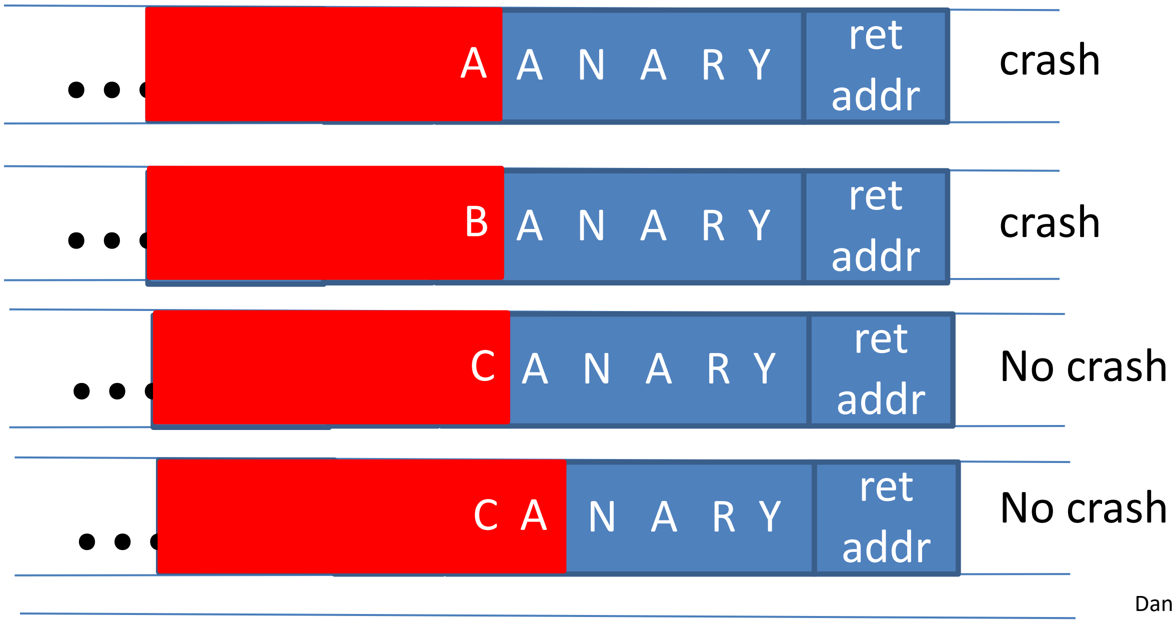
A common design for crash recovery:

- When process crashes, restart automatically (for availability)
- Often canary is unchanged (reason: relaunch using fork)

Danger:

Extract ret-addr to de-randomize code location

Extract stack function pointers to de-randomize heap



More methods: Shadow Stack

Shadow Stack: keep a copy of the stack in memory

- **On call**: push ret-address to shadow stack on call
- **On ret**: check that top of shadow stack is equal to ret-address on stack. Crash if not.
- Security: memory corruption should not corrupt shadow stack

Shadow stack using **Intel CET:** (supported in Windows 10, 2020)

- New register SSP: shadow stack pointer
- Shadow stack pages marked by a new "shadow stack" attribute:
  only "call" and "ret" can read/write these pages

More Methods: CFI [ABEL'05, …]

CFI: Control Flow Integrity

Ultimate Goal: ensure control flows as specified by code's flow graph

```
void HandshakeHandler(Session *s, char *pkt) {
      ...
      s->hdlr(s, pkt)
}
```

Compile time: build list of possible call targets for s->hdlr

Run time: before call, check that s->hdlr value is on list

Coarse CFI: ensure that every indirect call and indirect branch leads to a valid function entry point or branch target

THE END