Control Hijacking

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

⋮

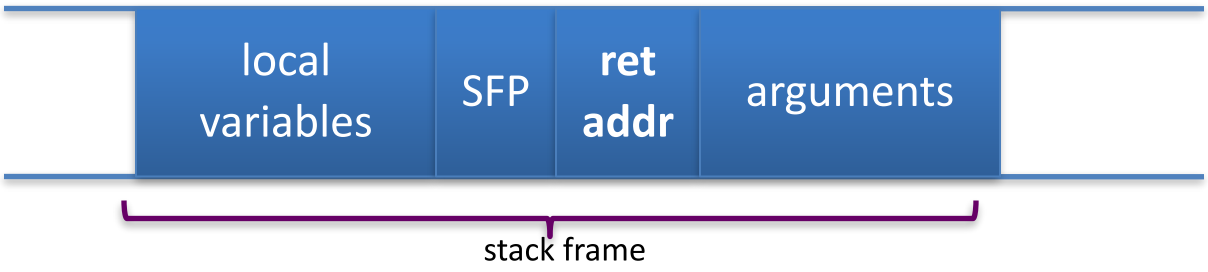
The mistake: mixing data and control

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

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

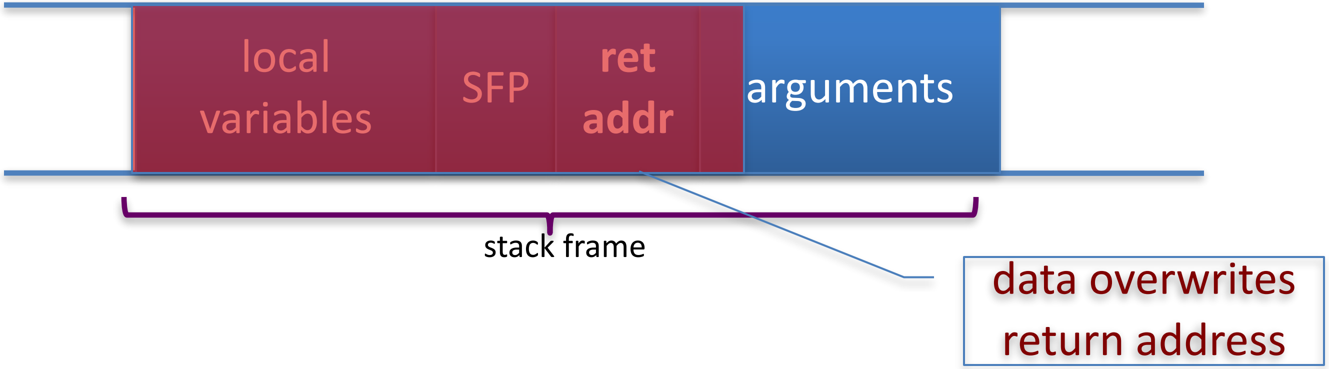
Control hijacking attacks

The problem: mixing data with control flow in memory



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, …

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**

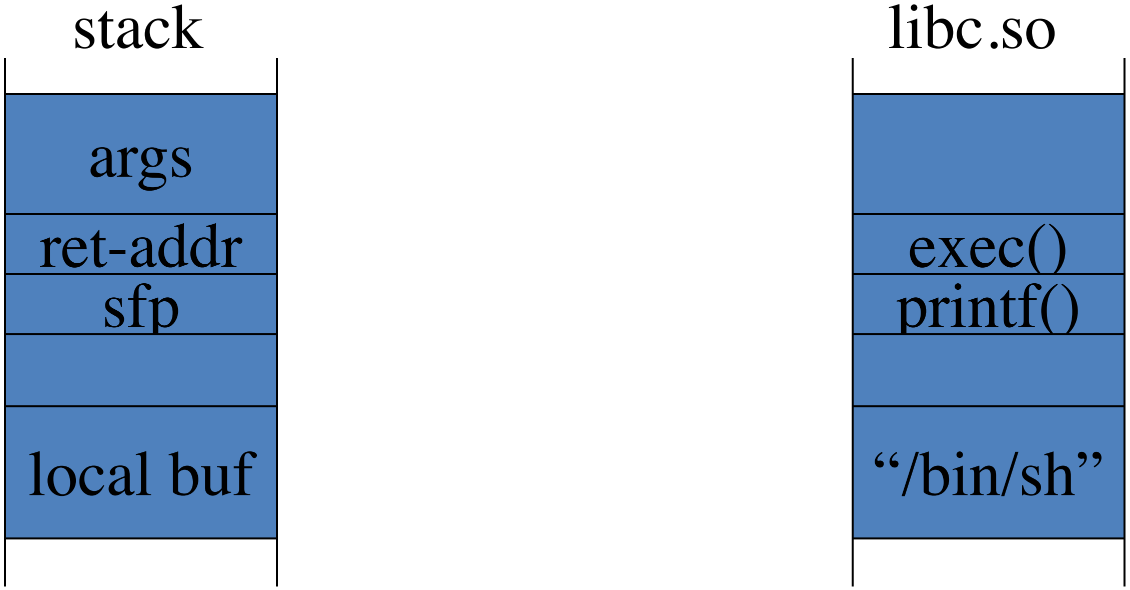
- **NX-bit** on AMD64, **XD-bit** on Intel x86 (2005), **XN-bit** on ARM
  - disable execution: an attribute bit in every Page Table Entry (PTE)
- Deployment:
  - All major operating systems
    - Windows DEP: since XP SP2 (2004)
      - Visual Studio: **/NXCompat[:NO]**
- Limitations:
  - Some apps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

Examples: DEP controls in Windows

DEP terminating a program

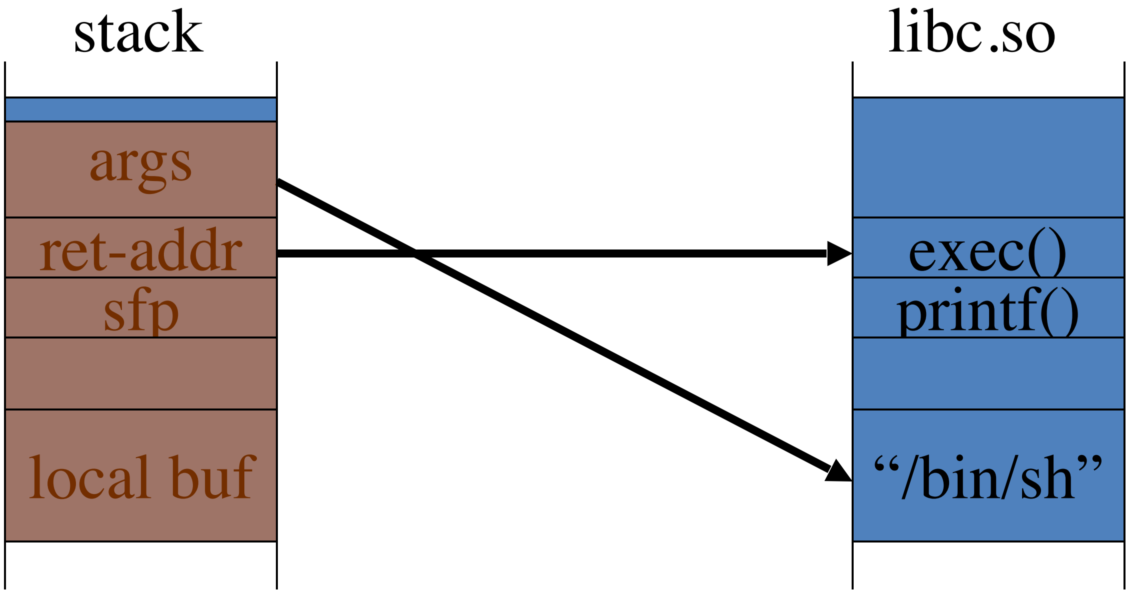
Attack: Return Oriented Programming (ROP)

Control hijacking without injecting code:



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 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);
```

Gadgets in victim code:

execve("/bin/sh") ret	dup2(s, 0) ret	dup2(s, 1) ret

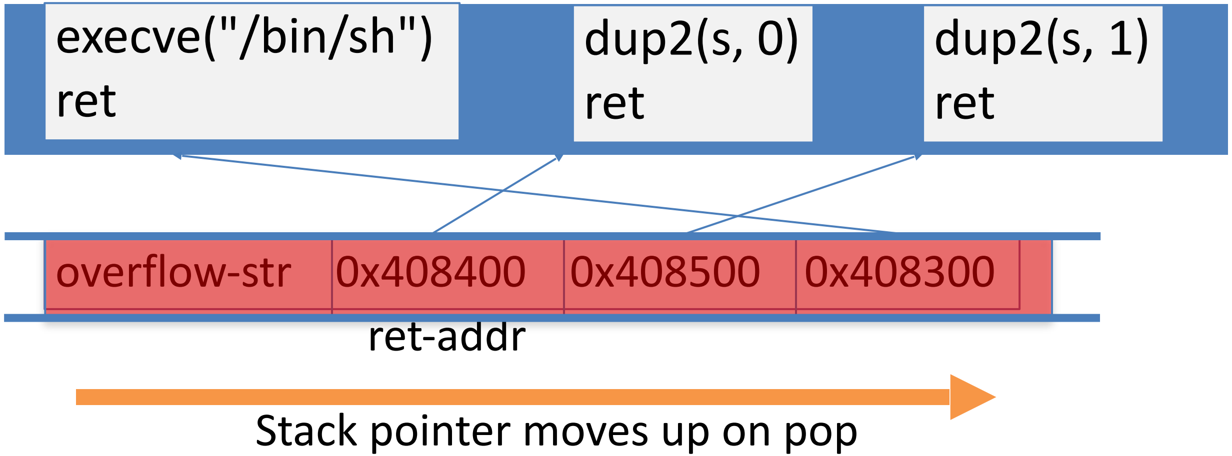
ROP: in more detail

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);
```

Gadgets in victim code:

Stack (set by attacker):



ROP: in even more detail

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

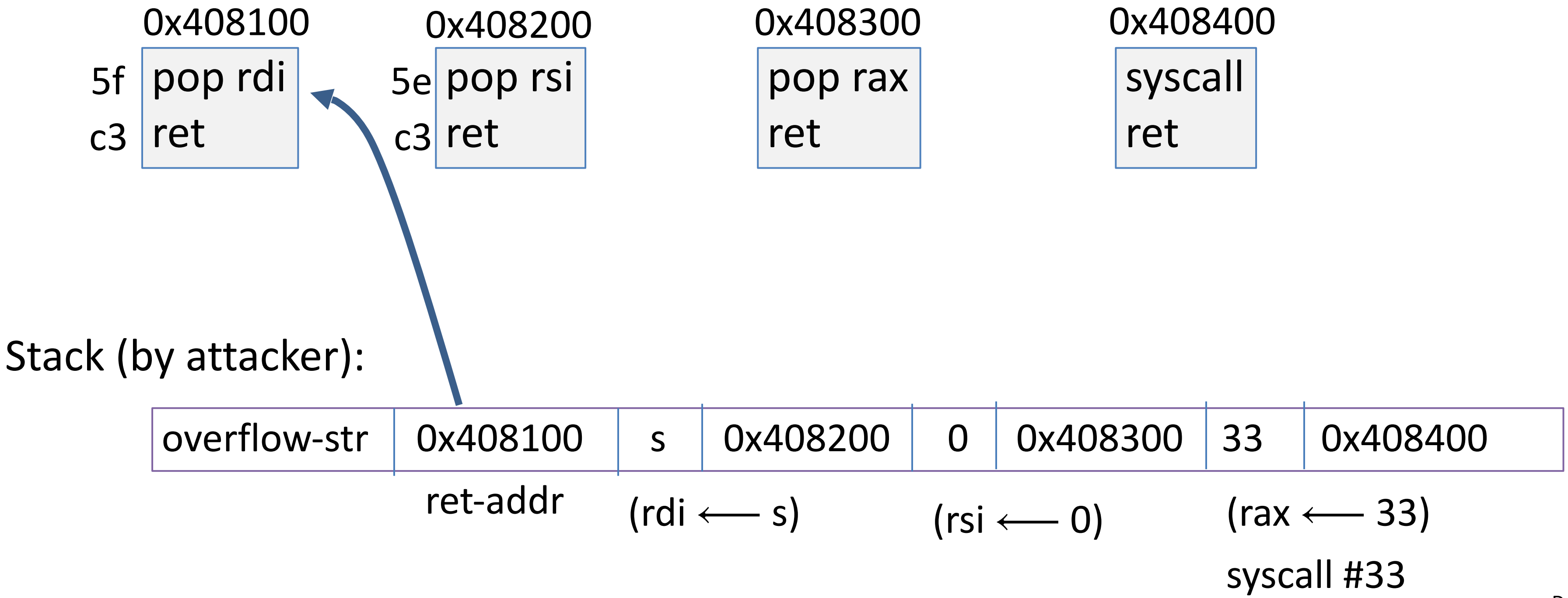
	0x408100		0x408200	0x408300	0x408400
5f	pop rdi	5e	pop rsi	pop rax	syscall
c3	ret	c3	ret	ret	ret

Stack (by attacker):

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

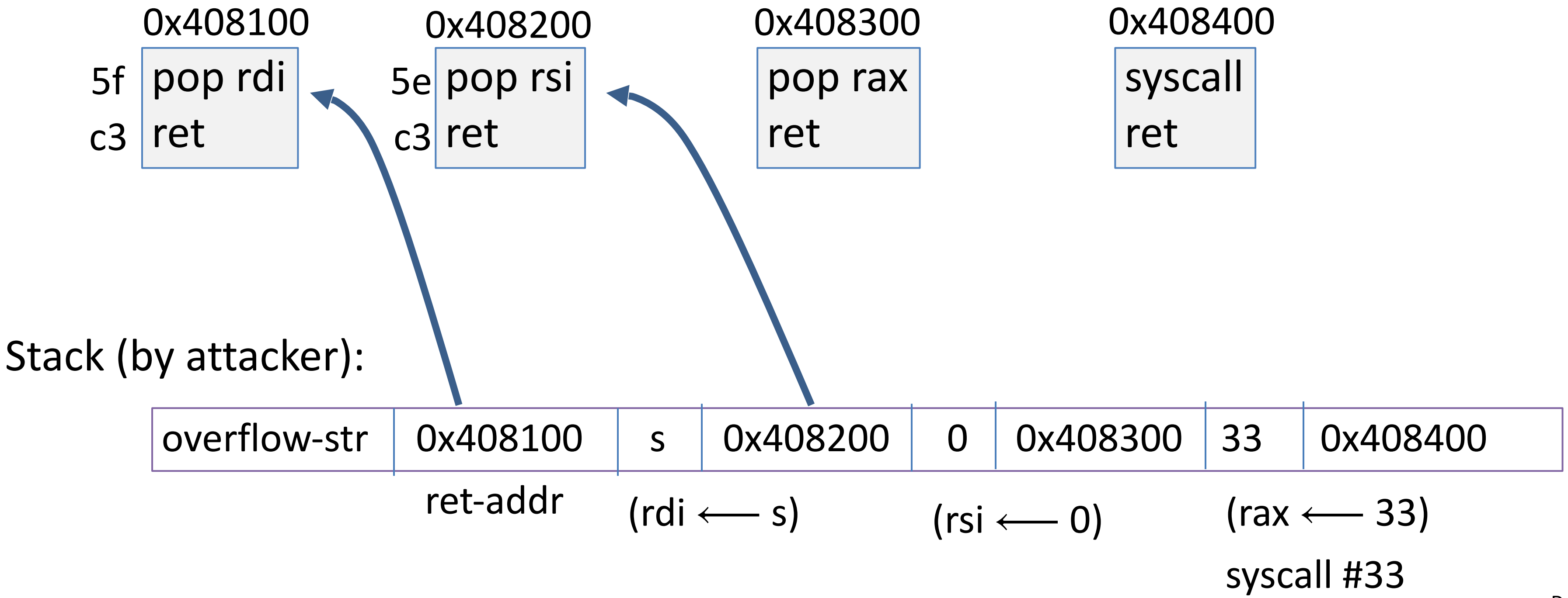
ROP: in even more detail

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



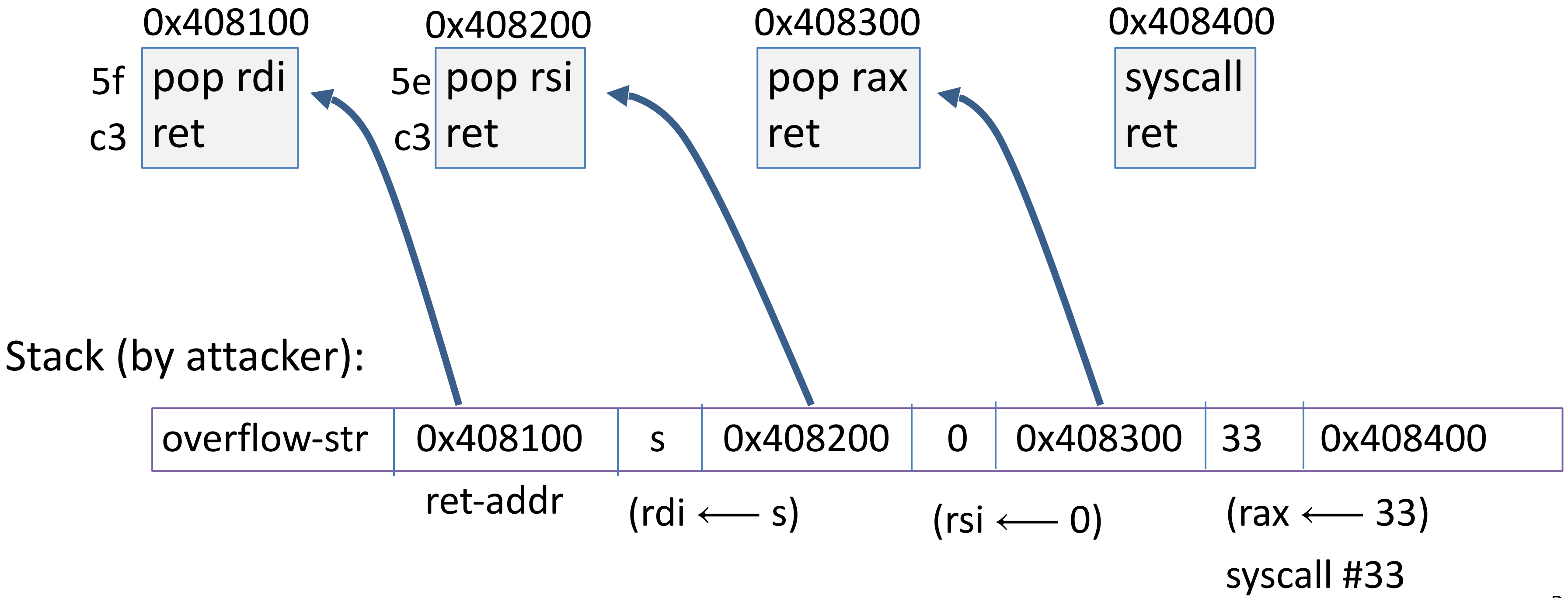
ROP: in even more detail

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



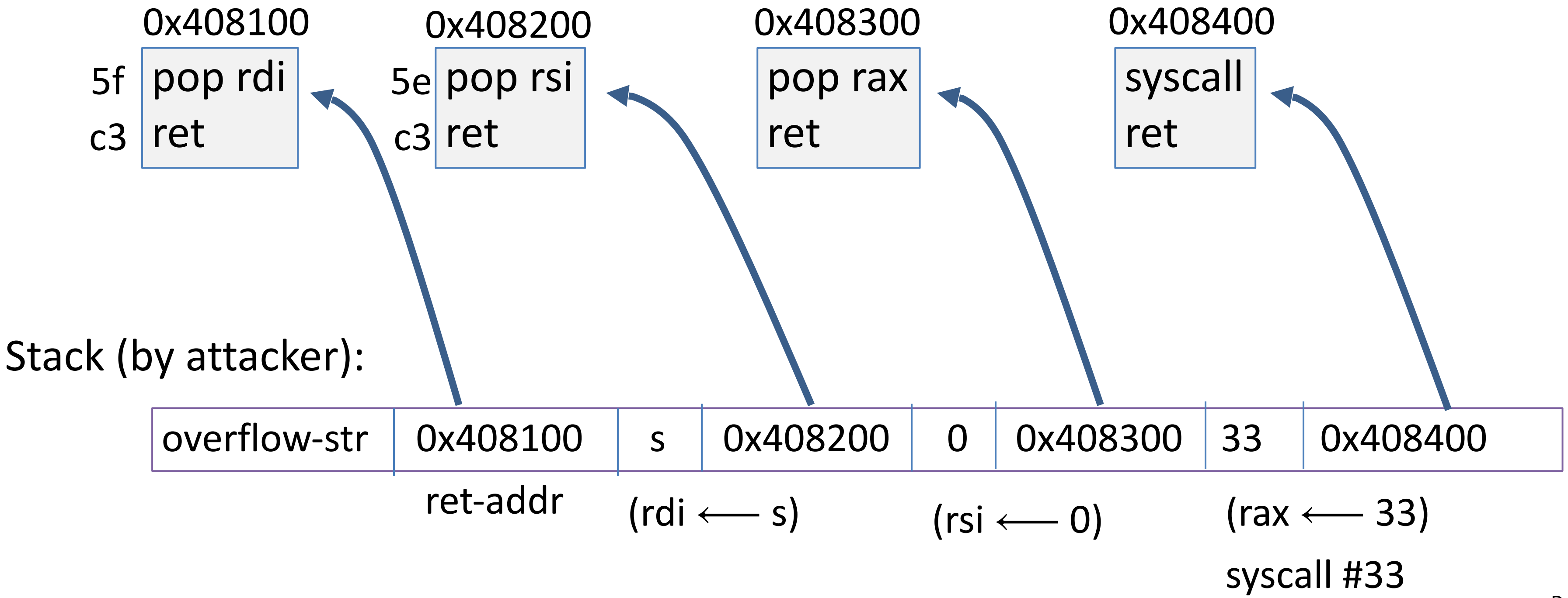
ROP: in even more detail

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

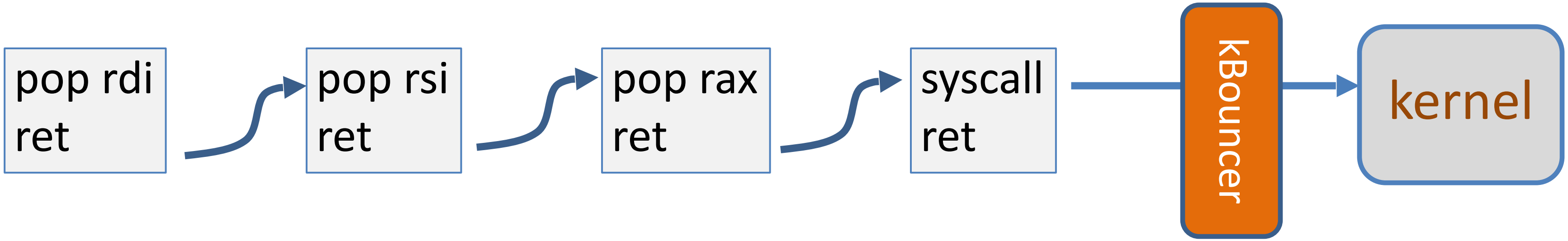


ROP: in even more detail

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



A very different idea: kBouncer (2012)



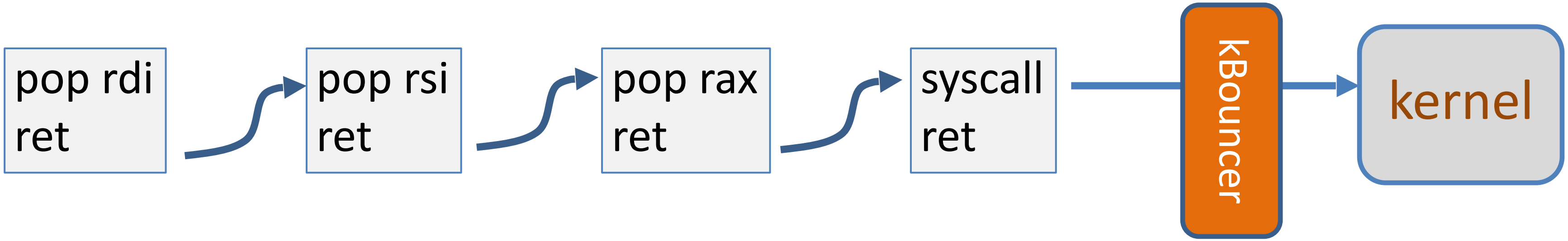
Observation: abnormal execution sequence

- ***ret*** returns to an address that does not follow a ***call***

Idea: before a syscall, check that every prior ret is not abnormal

- How: use Intel's *Last Branch Recording* (LBR)

A very different idea: kBouncer



Inte's **Last Branch Recording** (LBR):

- store 16 last executed branches in a set of on-chip registers (MSR)
- read using ***rdmsr*** instruction from privileged mode

kBouncer: before entering kernel, verify that last 16 ***ret***s are normal

- Requires no app. code changes, and minimal overhead
- Limitations: attacker can ensure 16 calls prior to syscall are valid

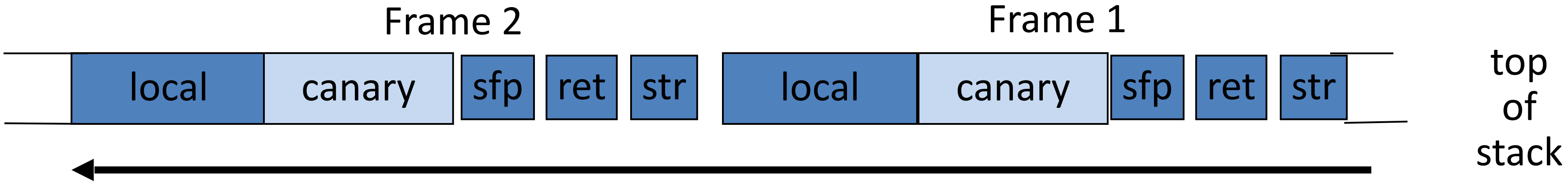
Control Hijacking Defenses

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.

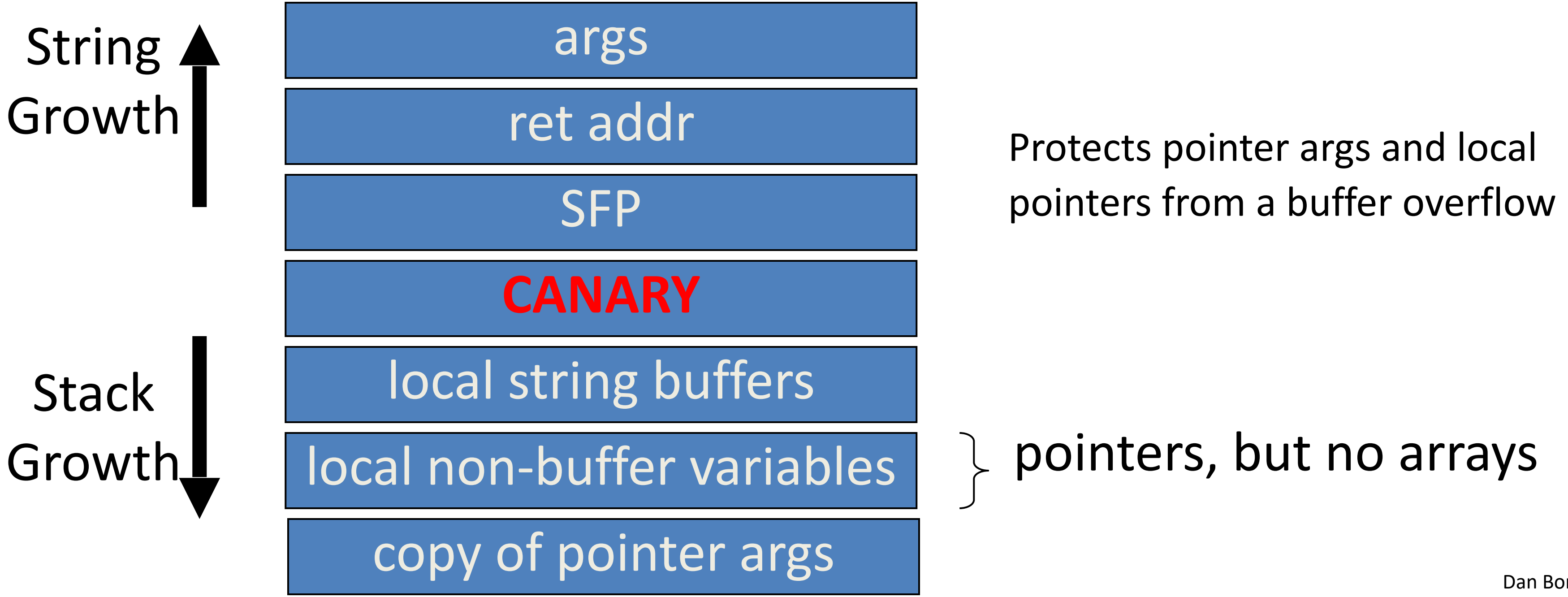
StackGuard (Cont.)

- 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.



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:

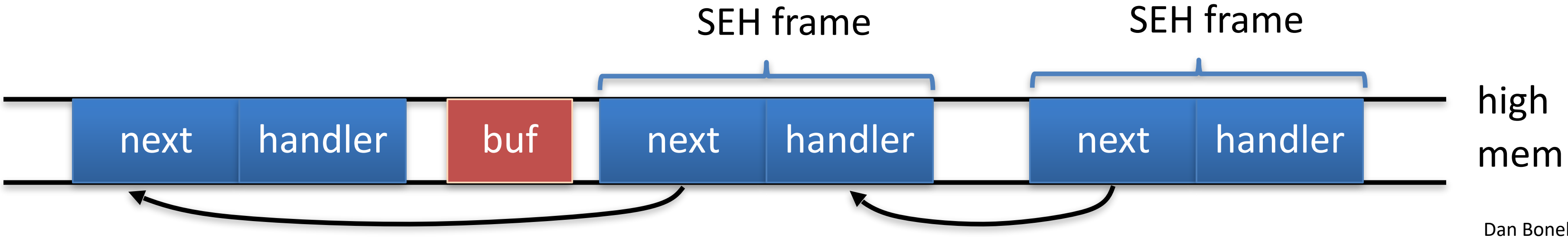
```
mov   ecx, DWORD PTR  [esp+8]
xor   ecx, esp
call  @__security_check_cookie@4
add   esp, 8
```

Enhanced /GS in Visual Studio 2010:

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

Evading /GS with exception handlers

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)



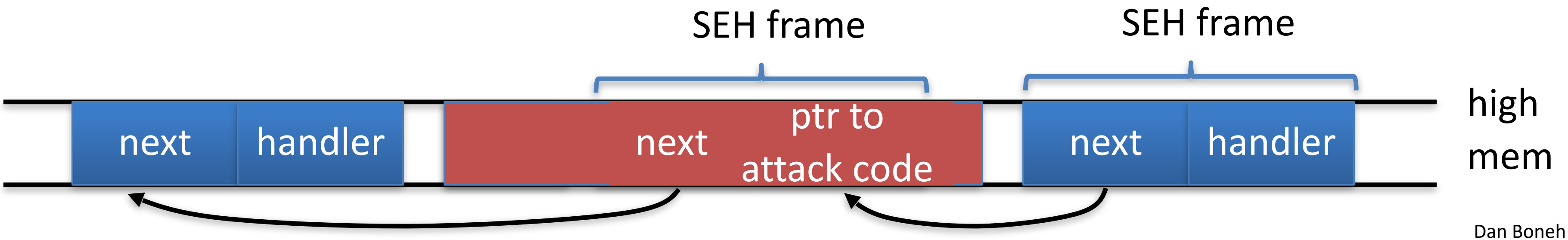
Evading /GS with exception handlers

- 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:
  - Some stack smashing attacks leave canaries unchanged: how?
  - Heap-based attacks still possible
  - Integer overflow attacks still possible
  - /GS by itself does not prevent Exception Handling attacks

    (also need SAFESEH and SEHOP)

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

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

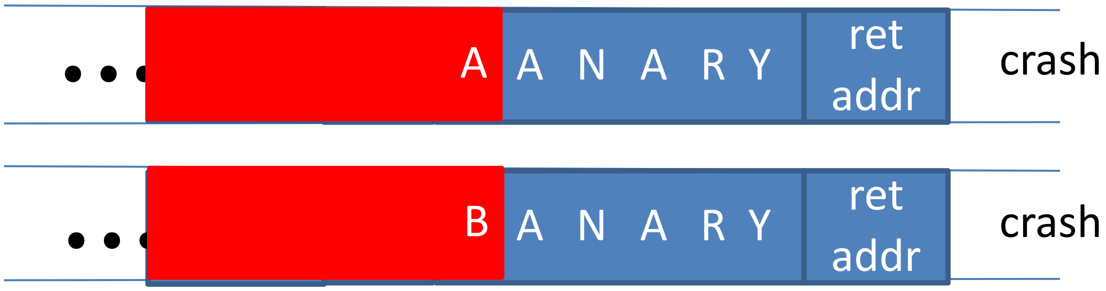
Even worse: canary extraction

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



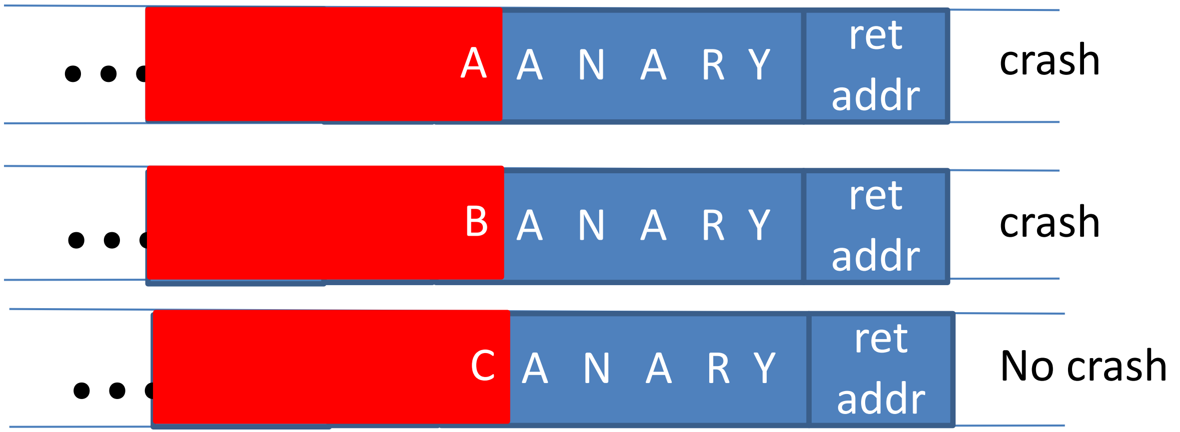
Even worse: canary extraction

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



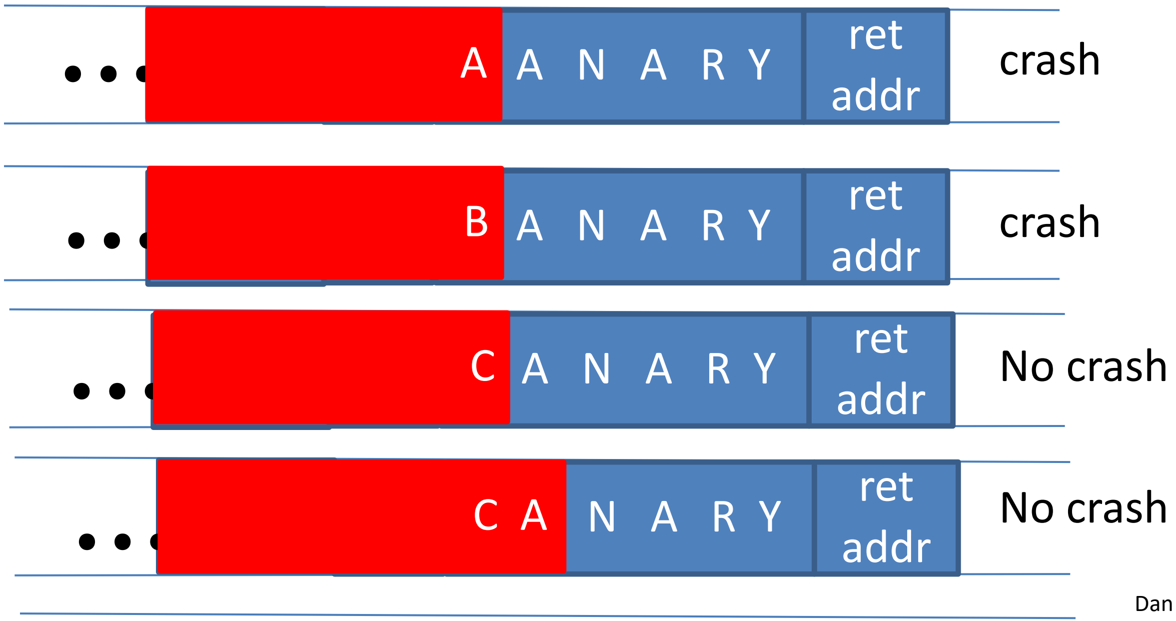
Even worse: canary extraction

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



Dan Boneh

Similarly: extract ASLR randomness

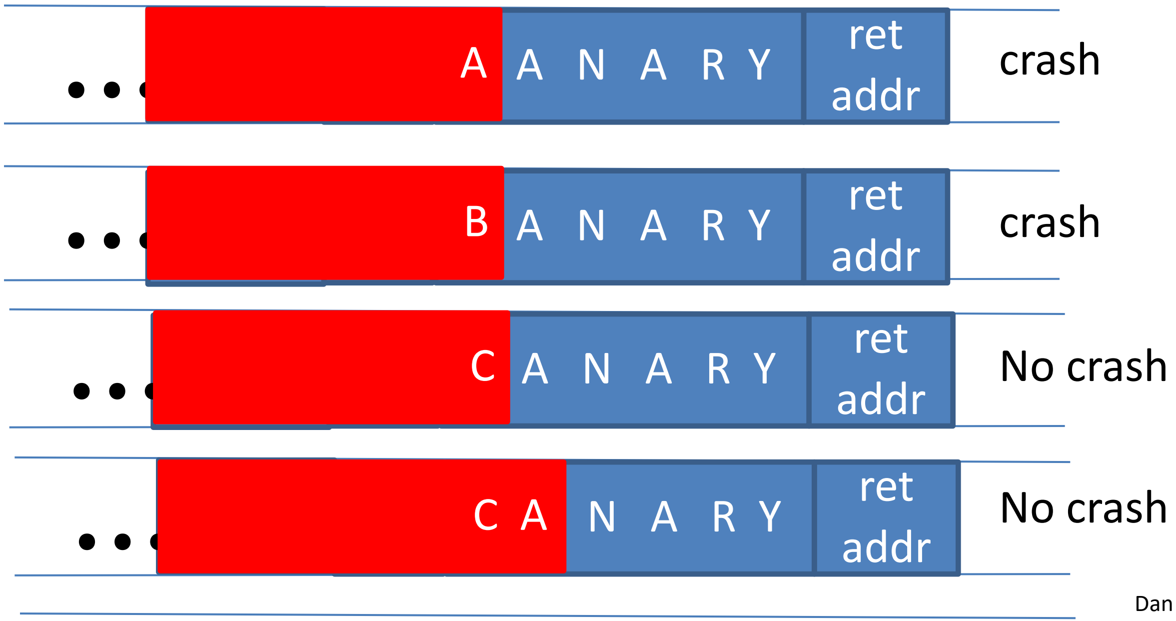
A common design for crash recovery:

- 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 <u>copy</u> 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

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

ARM Memory Tagging Extension (MTE)

Idea: (1) every 64-bit **memory pointer** P has a 4-bit "tag" (in top byte)

(2) every 16-byte user **memory region** R has a 4-bit "tag"

Processor ensures that: if P is used to read R then tags are equal

– otherwise: hardware exception

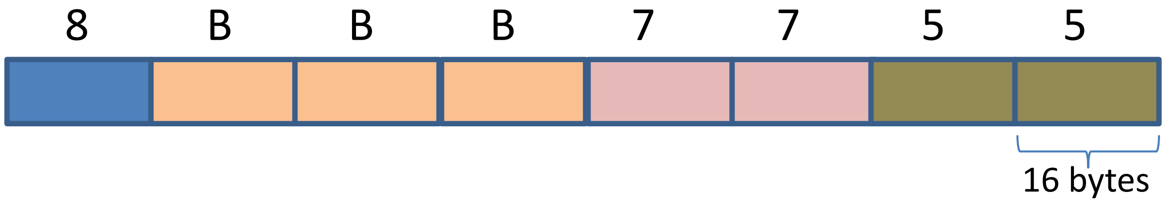
Tags are created using new HW instructions:

- LDG, STG: load and store tag to a memory region (use by malloc and free)
- ADDG, SUBG: pointer arithmetic on an address preserving tags

Tags prevent buffer overflows <u>and</u> use after free

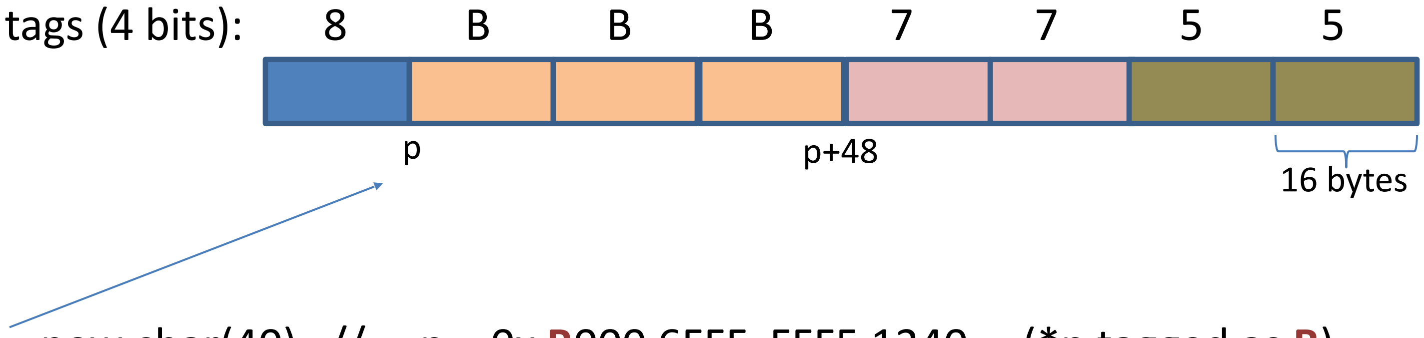
Example:

tags (4 bits):



Tags prevent buffer overflows <u>and</u> use after free

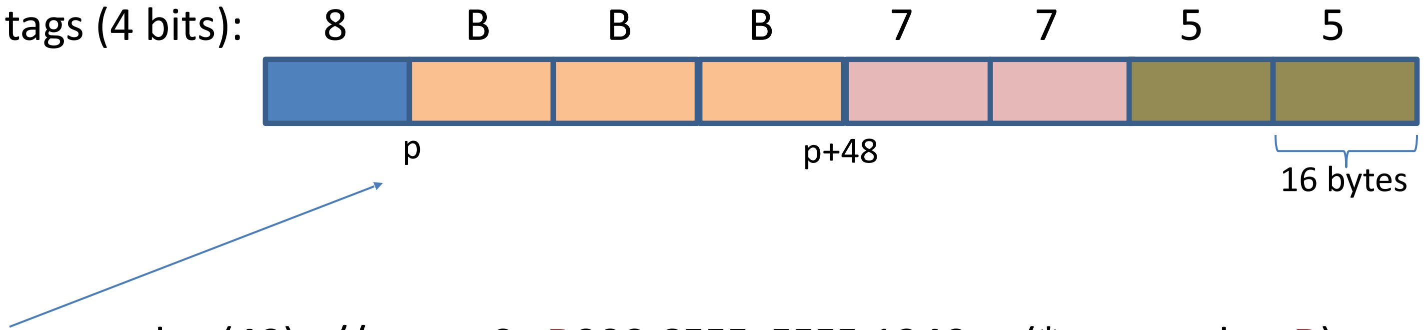
Example:



(1) char *p = new char(40); // p = 0x **B**000 6FFF FFF5 1240 (*p tagged as **B**)

Tags prevent buffer overflows <u>and</u> use after free

Example:

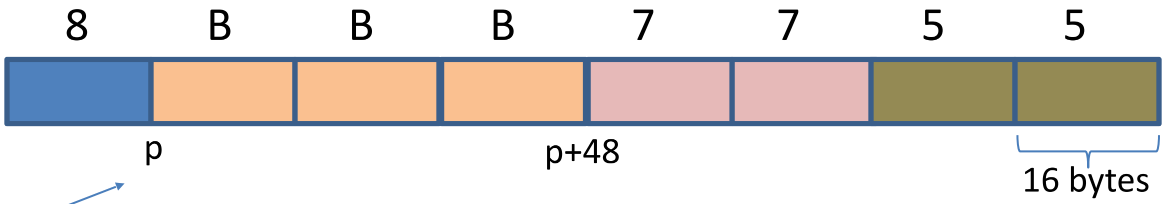


(1) char *p = new char(40); // p = 0x **B**000 6FFF FFF5 1240 (*p tagged as **B**)

(2) p[50] = ’a’; // B≠7 ⟹ tag mismatch exception (buffer overflow)

Tags prevent buffer overflows <u>and</u> use after free

Example: tags (4 bits):



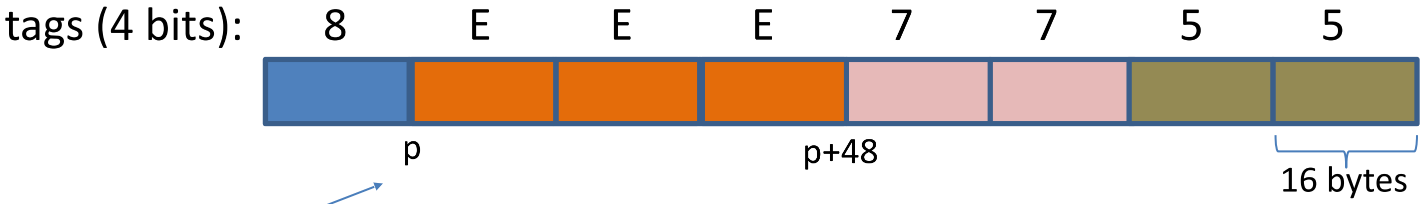
(1) char *p = new char(40); // p = 0x **B**000 6FFF FFF5 1240 (*p tagged as **B**)

(2) p[50] = 'a'; // B≠7 ⟹ tag mismatch exception (buffer overflow)

(3) delete [] p; // memory is re-tagged from **B** to **E**

Tags prevent buffer overflows <u>and</u> use after free

Example:



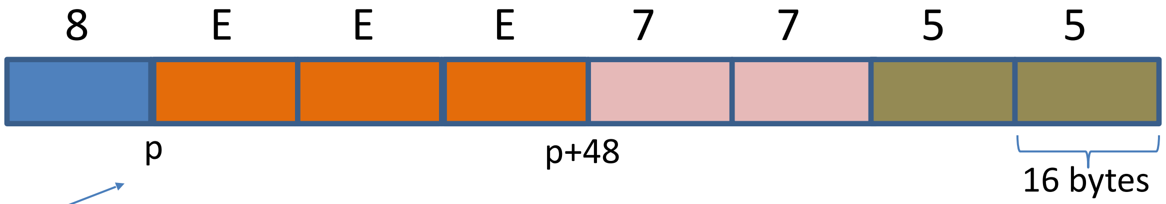
(1) char *p = new char(40); // p = 0x **B**000 6FFF FFF5 1240 (*p tagged as **B**)

(2) p[50] = ’a’; // B≠7 $\Longrightarrow$ tag mismatch exception (buffer overflow)

(3) delete [] p; // memory is re-tagged from **B** to **E**

Tags prevent buffer overflows and use after free

Example: tags (4 bits):



(1) char *p = new char(40); // p = 0x **B**000 6FFF FFF5 1240 (*p tagged as **B**)

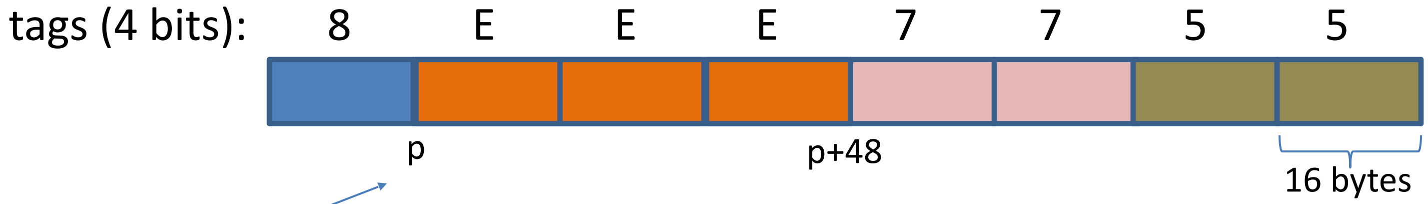
(2) p[50] = ’a’; // B≠7 ⟹ tag mismatch exception (buffer overflow)

(3) delete [] p; // memory is re-tagged from **B** to **E**

(4) p[7] = ‘a’; // B≠E ⟹ tag mismatch exception (use after free)

Tags prevent buffer overflows <u>and</u> use after free

Example:



(1) char *p = new char(40); // p = 0x B000 6FFF FFF5 1240 (*p tagged as **B**)

(2) p[50] = ’a’; // B≠7 $\Longrightarrow$ tag mismatch exception (buffer overflow)

(3) delete [] p; // memory is re-tagged from **B** to **E**

(4) p[7] = ‘a’; // B≠E $\Longrightarrow$ tag mismatch exception (use after free)

Note: out of bounds access to p[44] at (2) will not be caught.

Control Hijacking Defenses

Control Flow Integrity (CFI)

Control flow integrity (CFI) [ABEL'05, …]

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

Coarse CFI: Control Flow Guard (CFG) (Windows 10)

Coarse 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
```

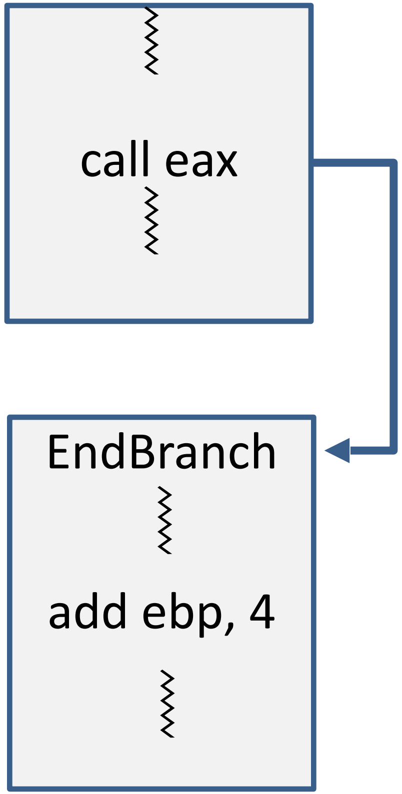
ensures target is the entry point of a function

Coarse CFI using **EndBranch** (Intel) and **BTI (ARM)**

New instruction **EndBranch** (Intel) and **BTI** (ARM):

- After an indirect **JMP** or **CALL**:
  the next instruction in the instruction stream must be **EndBranch**

- If not, then trigger a #CP fault and halt execution

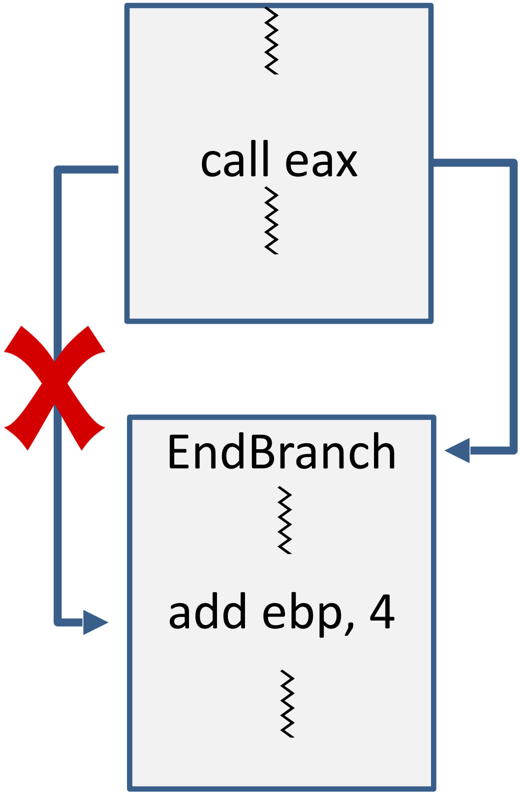


Coarse CFI using **EndBranch** (Intel) and **BTI** (ARM)

New instruction **EndBranch** (Intel) and **BTI** (ARM):

- After an indirect **JMP** or **CALL**:
  the next instruction in the
  instruction stream must be **EndBranch**

- If not, then trigger a #CP fault
  and halt execution



Coarse CFI using **EndBranch** (Intel) and **BTI** (ARM)

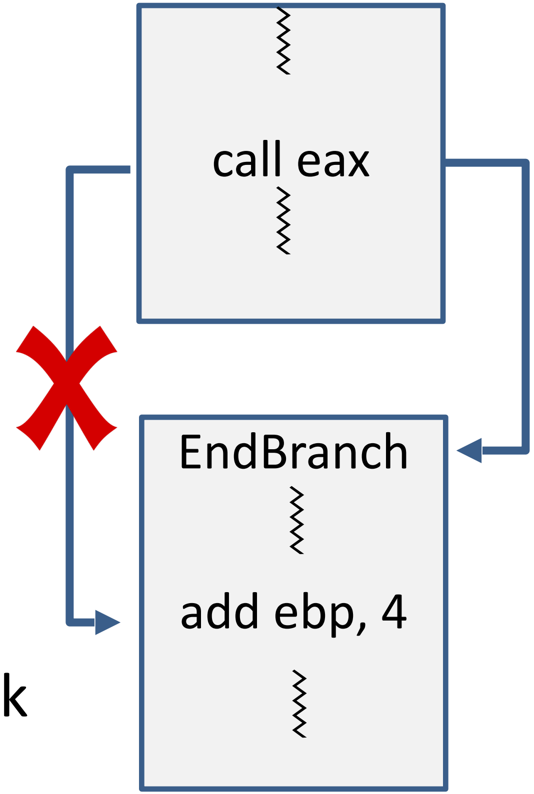
New instruction **EndBranch** (Intel) and **BTI** (ARM):

- After an indirect **JMP** or **CALL**:
  the next instruction in the
  instruction stream must be **EndBranch**

- If not, then trigger a #CP fault
  and halt execution

- Ensures an indirect JMP or CALL can only go
  to a valid target address ⇒ no func. ptr. hijack

  (compiler inserts EndBranch at valid locations)



CFG, EndBranch, BTI: limitations

Poor man's version of CFI:

- Pr… valid fu…

- Do not prevent attacker from causing a jump to a valid **wrong** function

- Hard to build accurate control flow graph statically

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

…s …of a

An example

```
void HandshakeHandler(Session *s, char *pkt) {
    s->hdlr = &LoginHandler;
    ... Buffer overflow over Session struct ...
}

void LoginHandler(Session *s, char *pkt) {
    bool auth = CheckCredentials(pkt);
    s->dhandler = &DataHandler;
}

void DataHandler(Session *s, char *pkt);
```

An example

```
void HandshakeHandler(Session *s, char *pkt) {
    s->hdlr = &LoginHandler;
    ... Buffer overflow over Session struct ...
}

void LoginHandler(Session *s, char *pkt) {
    bool auth = CheckCredentials(pkt);
    s->dhandler = &DataHandler;
}

void DataHandler(Session *s, char *pkt);
```

Attacker controls handler

An example

void **HandshakeHandler**(Session *s, char *pkt) {
    s->hdlr = &**LoginHandler**;
    ... Buffer overflow over Session struct ...
}

Attacker controls handler

void **LoginHandler**(Session *s, char *pkt) {
    bool auth = **CheckCredentials**(pkt);
    s->dhandler = &**DataHandler**;
}

void **DataHandler**(Session *s, char *pkt);

static CFI: attacker can call **DataHandler** to bypass authentication

Cryptographic Control Flow Integrity (CCFI)
(ARM PAC - pointer authentication)

Threat model: attacker can read/write **anywhere** in memory,
program should not deviate from its control flow graph

CCFI approach: Every time a jump address is written/copied anywhere in memory:
compute 64-bit AES-MAC and append to address

On heap: **tag = AES(k, (jump-address, 0 ll source-address))**

on stack: **tag = AES(k, (jump-address, 1 ll stack-frame))**

Before following address, verify AES-MAC and crash if invalid

Where to store key **k**? In xmm registers (not memory)

Back to the example

```
void HandshakeHandler(Session *s, char *pkt) {
    s->hdlr = &LoginHandler;
    ... Buffer overflow in Session struct ...
}

void LoginHandler(Session *s, char *pkt) {
    bool auth = CheckCredentials(pkt);
    s->dhandler = &DataHandler;
}

void DataHandler(Session *s, char *pkt);
```

Attacker controls handler

CCFI: Attacker cannot create a valid MAC for **DataHandler** address

THE END