Control Hijacking: Defenses

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

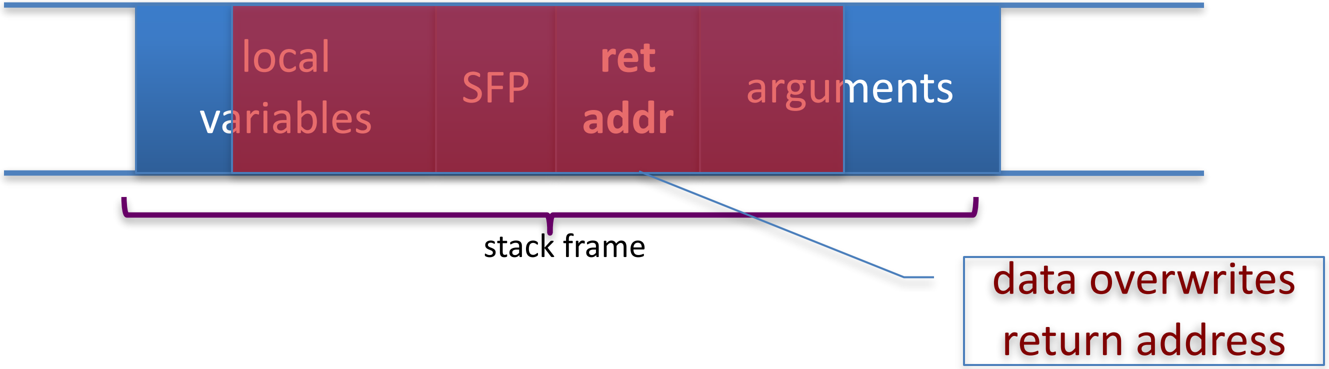
An ancient design flaw:

enables anyone to inject control signals

1971:   AT&T learns neve to mix control and nats

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, Prefast/Prefix.
   - Rewrite software in a type safe languange (Java, ML)
     - Difficult for existing (legacy) code …

2. Platform defenses: prevent attack code execution

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

Transform:

Complete Breach

Denial of service

Control Hijacking

Platform Defenses

Marking memory as non-execute

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

NX bit in every Page Table Entry (PTE)

<u>Deployment</u>:

Linux (via PaX project); OpenBSD

Windows: since XP SP2 (DEP)

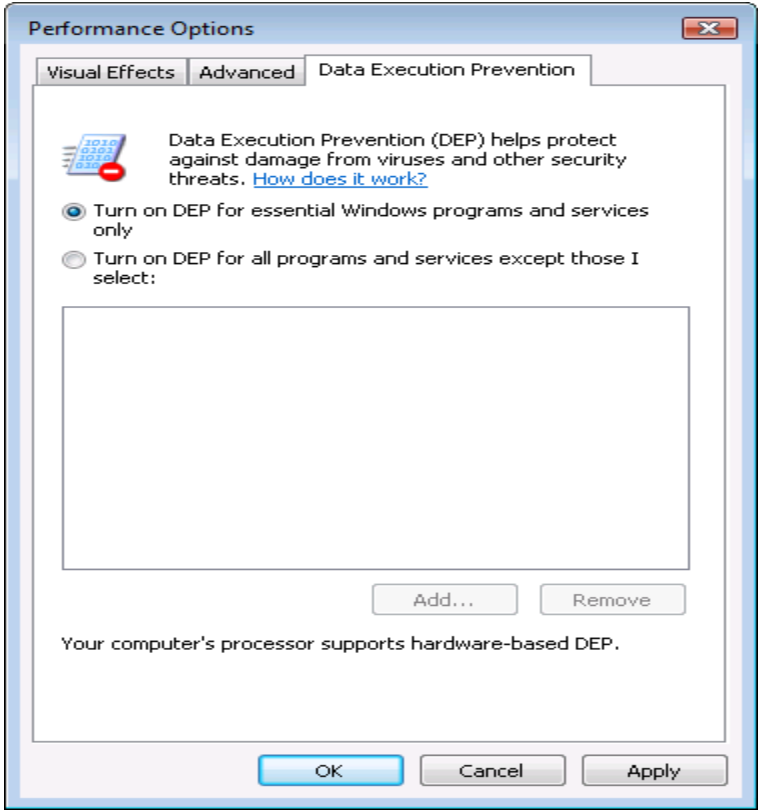
Visual Studio: **/NXCompat[:NO]**

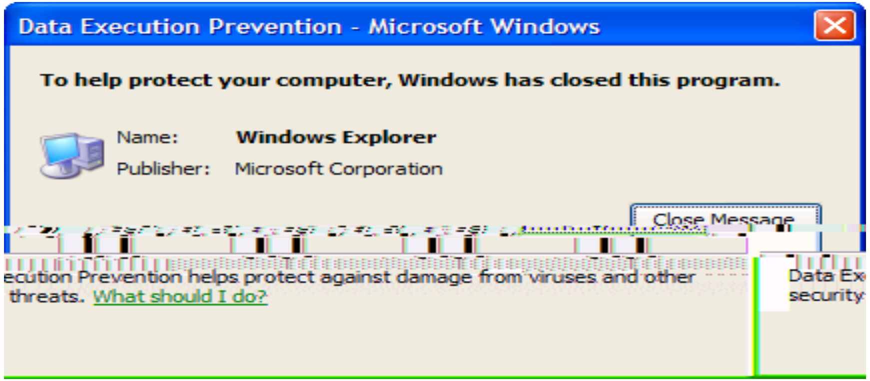
<u>Limitations</u>:

Some apps need executable heap (e.g. JITs).

Can be easily bypassed using **Return Oriented Programming (ROP)**

Examples: DEP controls in Windows

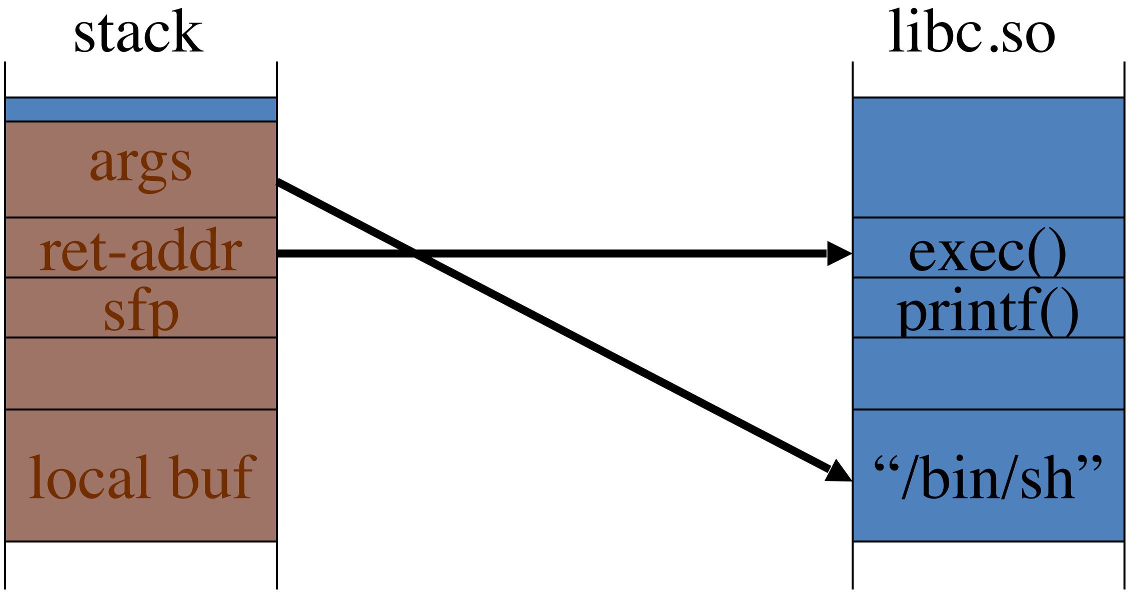




DEP terminating a program

Attack: Return Oriented Programming (ROP)

Control hijacking without injecting code:



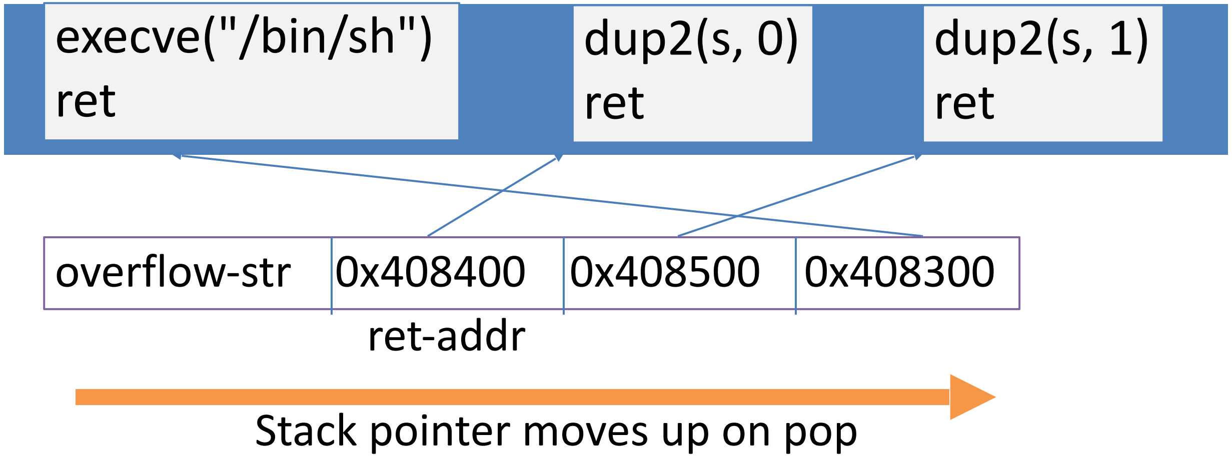
ROP: in more detail

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);
```

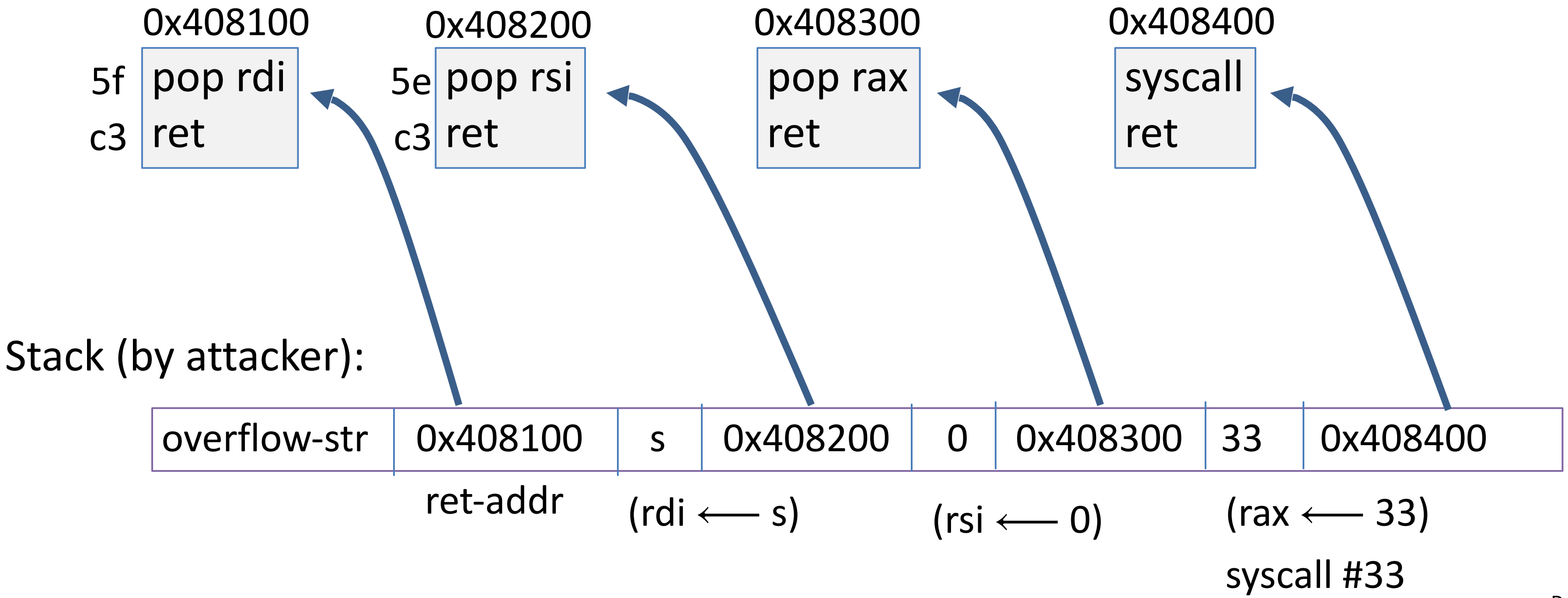
Gadgets in victim code:

Stack (set by attacker):



ROP: in even more detail

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



What to do?? Randomization

ASLR: (Address Space Layout Randomization)

Map shared libraries to rand location 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 methods:

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

Instruction Set Randomization ( )

ASLR Example

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

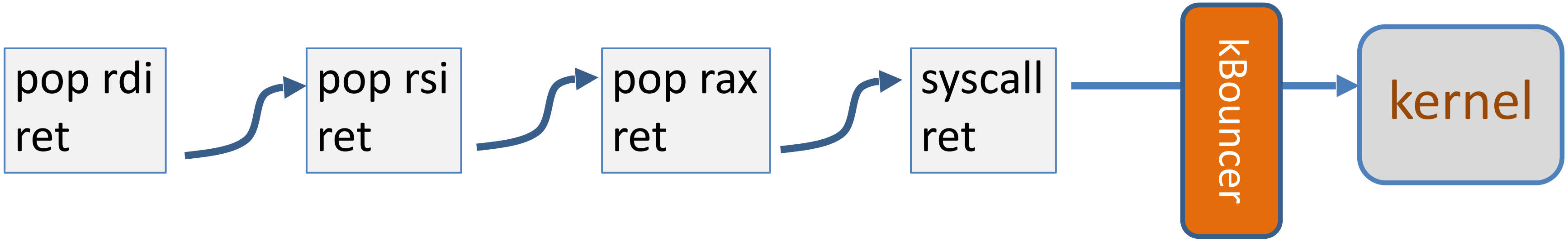
ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Note: everything in process memory must be randomized

stack, heap, shared libs, base image

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

A very different idea: kBouncer



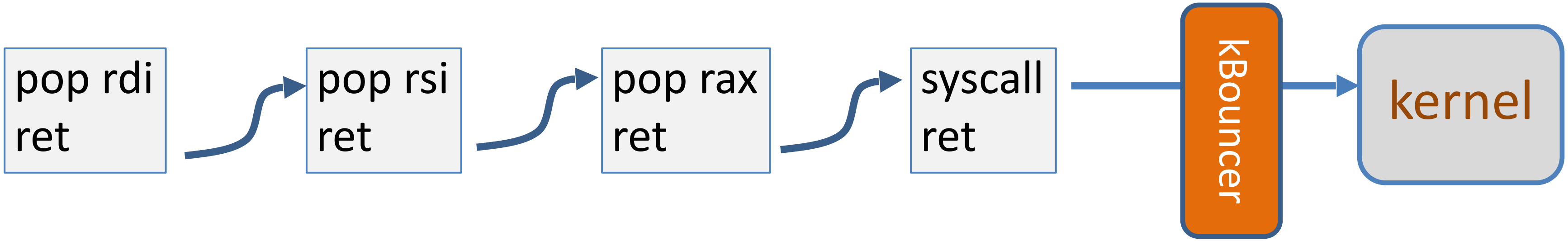
Observation: abnormal execution sequence

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

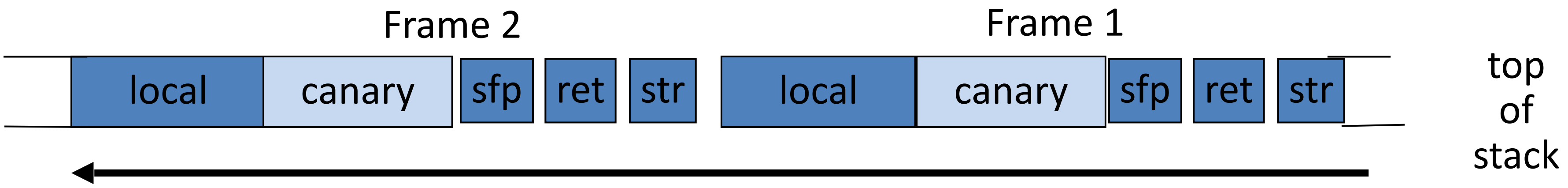
Many run-time checking techniques …

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

Note: Canaries do not provide full protection

- Some stack smashing attacks leave canaries unchanged

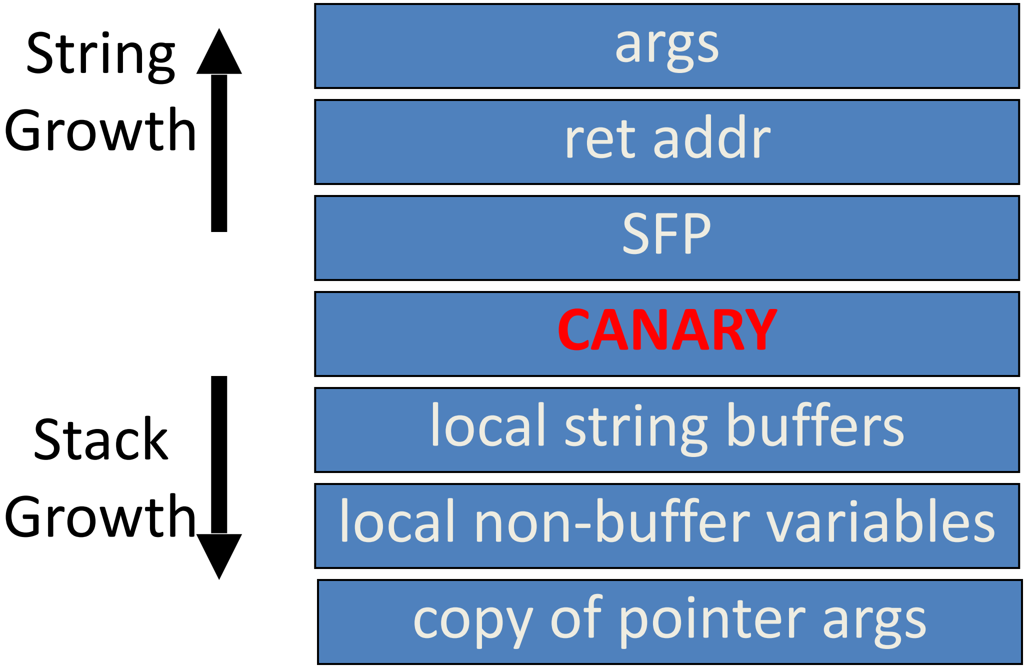
Heap protection: PointGuard

- 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 **-fstack-protector**

Rearrange stack layout to prevent ptr overflow.



Protects pointer args and local pointers from a buffer overflow

pointers, but no arrays

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
```

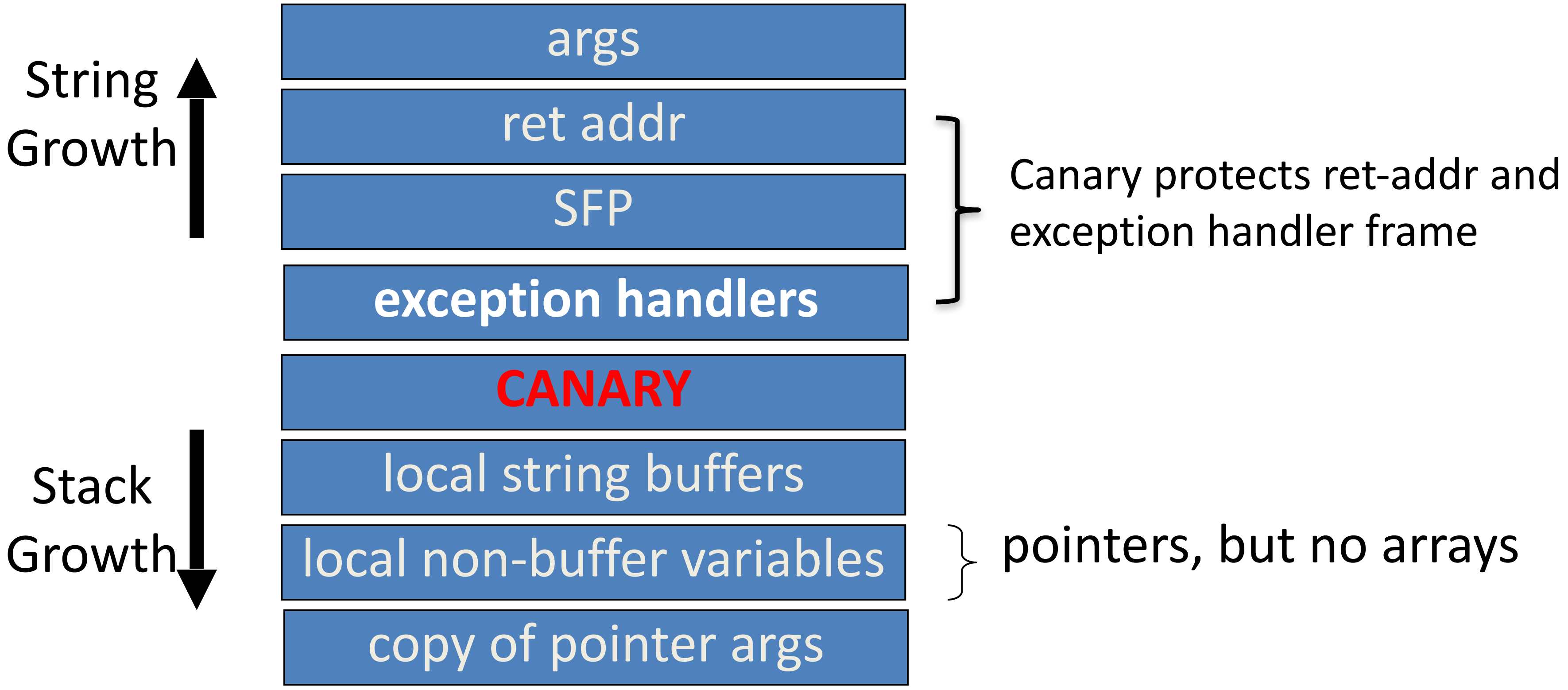
Function epilog:

```
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



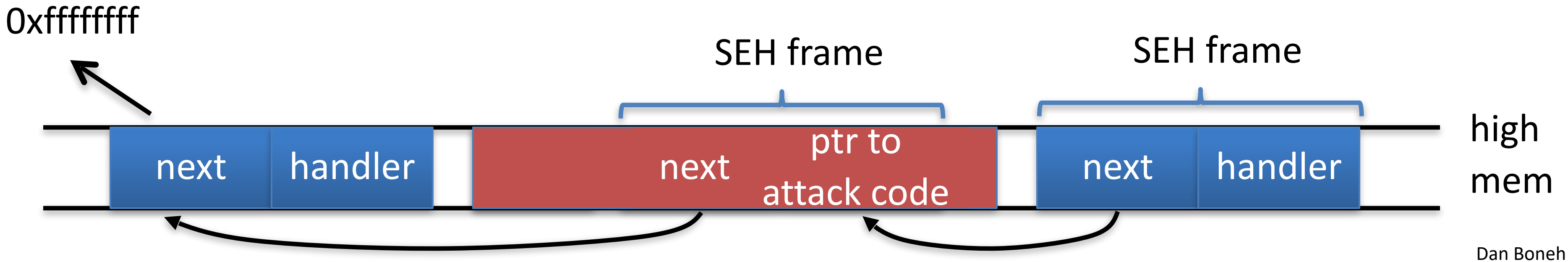
Evading /GS with exception handlers

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)

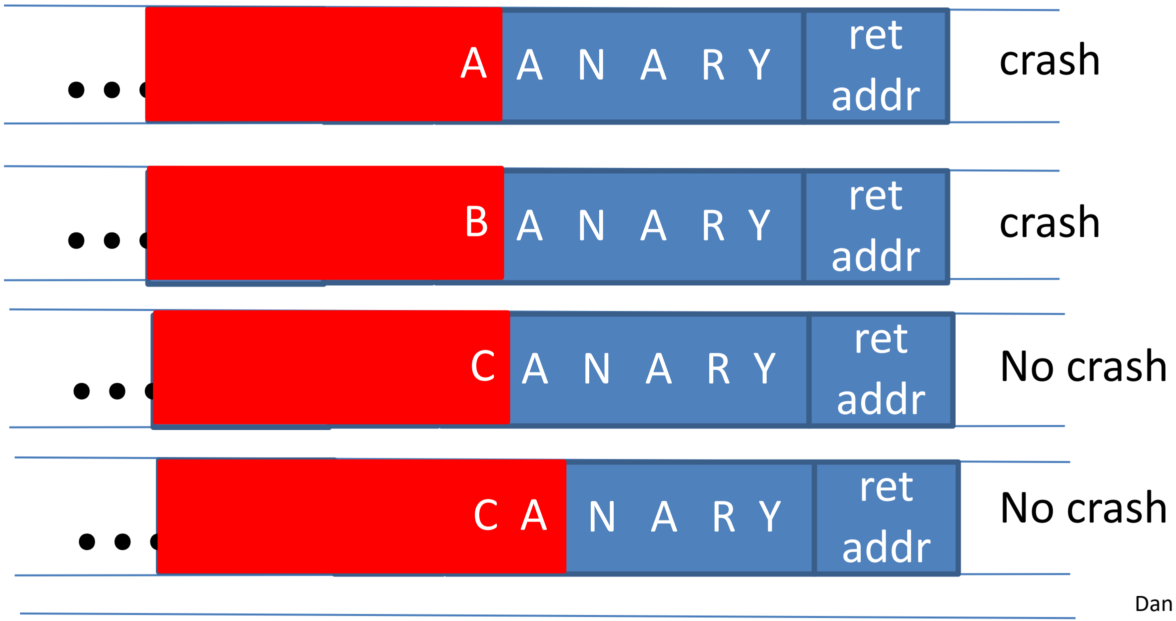
Even worse: canary extraction

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 stack location
- Extract stack function pointers to de-randomize heap

What if can't recompile: Libsafe

<u>Solution 2</u>: Libsafe (Avaya Labs)

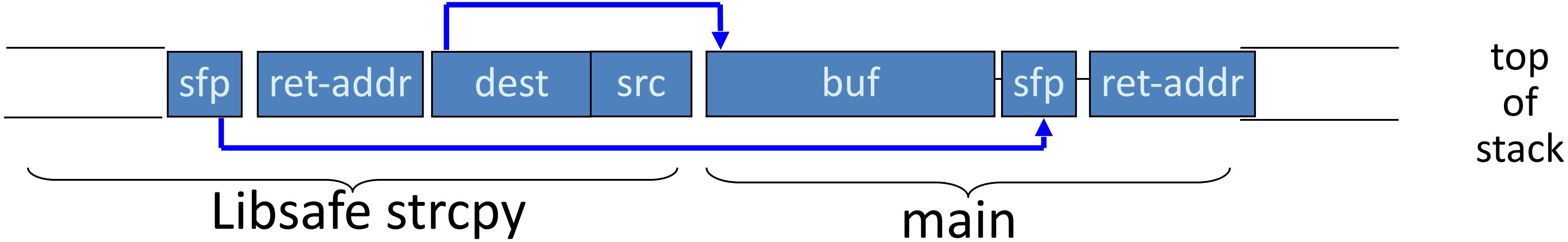
Dynamically loaded library (no need to recompile app.)

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 …

- **StackShield**
  - At function prologue, copy return address and to “safe” location (beginning of data segment)
  - Upon return, check that and is equal to copy.
  - Implemented as assembler file processor ( )
- **Control Flow Integrity** (CFI)
  - A combination of static and dynamic checking
    - Statically determine program control flow
    - Dynamically enforce control flow integrity

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

Run time: before call, check validity of s->hdlr

Lots of academic research on CFI systems:

CCFIR (2013), kBouncer (2013), FECFI (2014), CSCFI (2015), …

and many attacks …

Control Flow Guard (CFG) (Windows 10)

Poor man's version of 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

Control Flow Guard (CFG) (Windows 10)

Poor man's version of CFI:

Pr… valid
fu…

Does 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 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

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

Cryptographic Control Flow Integrity (CCFI)

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 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 ...

}

Attacker controls handler

void **LoginHandler**(Session *s, char *pkt) {

    bool auth = **CheckCredentials**(pkt);

    s->dhandler = &**DataHandler**;

}

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

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

THE END