



CS 253 Cyber Security Control Hijacking Defenses

ShanghaiTech University

SIST - Yuan Xiao

Admin

- Project 1 released, DDL 9/27 (23:59 pm)
- HW 1 released, DDL 9/27 (23:59 pm)
- Register **Gradescope** and **Piazza!!!**
- Come to claim your **QR code** to access **presentation topic selection** if you enter the course late
- Presentations begin on Week 3/4 (if you pick week 2/3 topics)

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

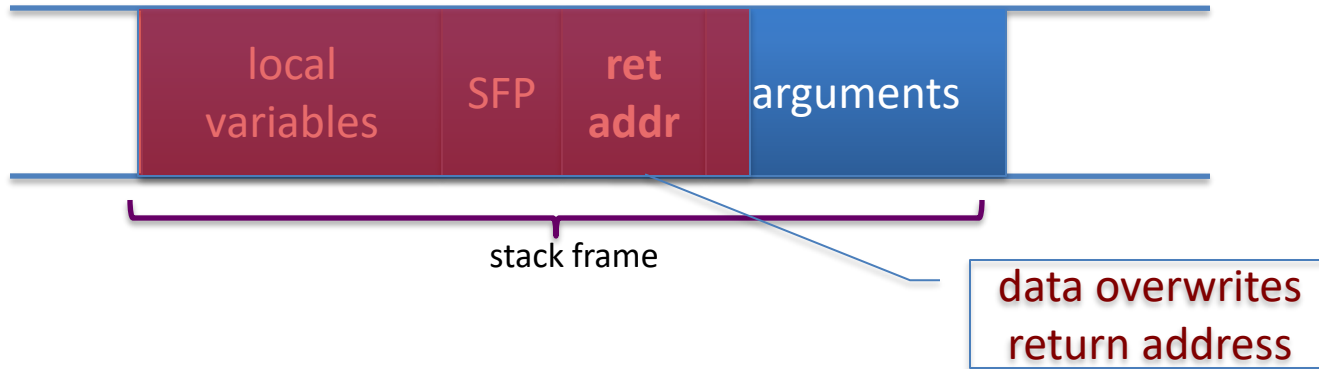
- An ancient design flaw:
 - enables anyone to inject control signals



- 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, ...
- Rewrite software in a type safe language (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 (ASan, MTE), ...

Transform:

Complete Breach



Denial of service

01

PART ONE

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])

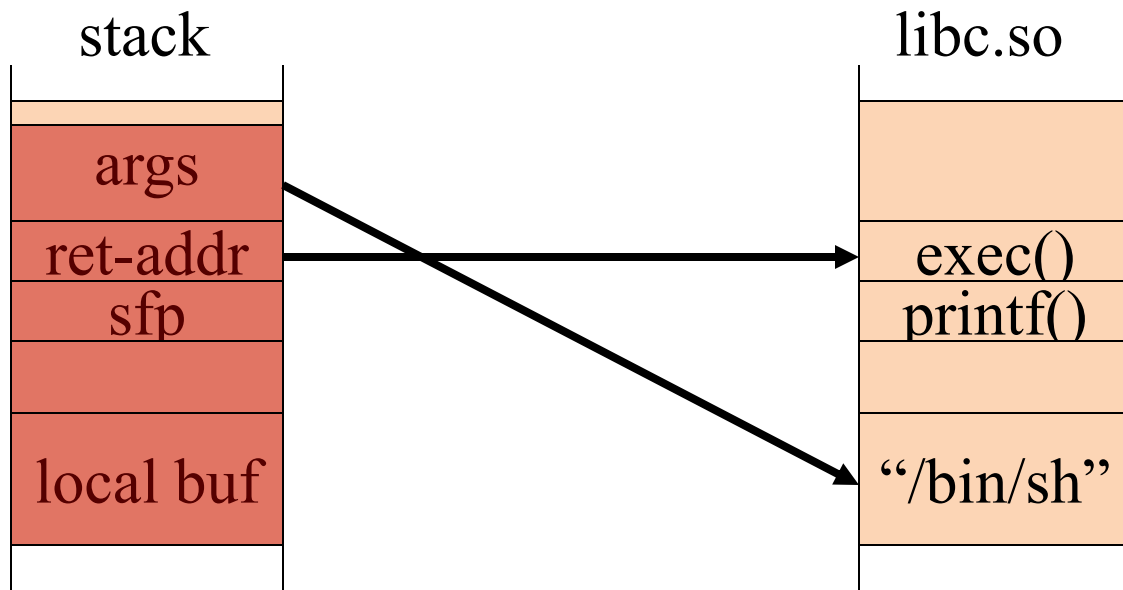
- Limitations:

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

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

Attack: Return Oriented Programming (ROP)

Control hijacking **without injecting code**:

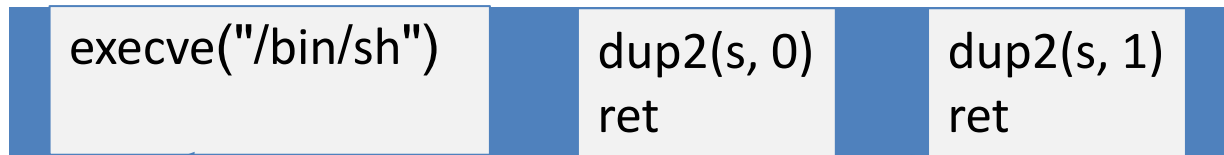


ROP: in more detail

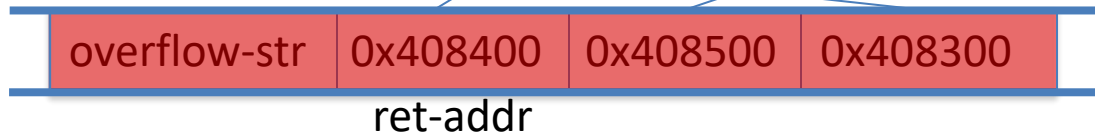
To run `/bin/sh` we must direct ***stdin*** and ***stdout*** to the socket:

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

Gadgets in victim code:

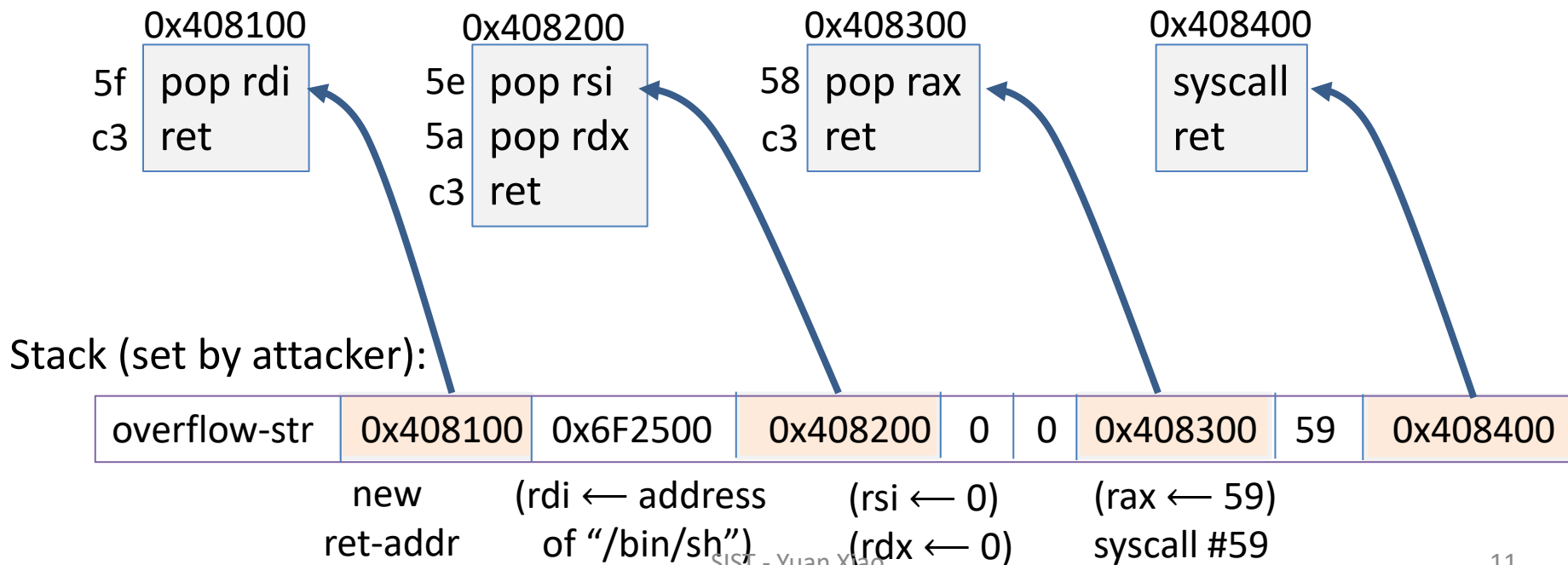


Stack (set by attacker):



ROP: in even more detail

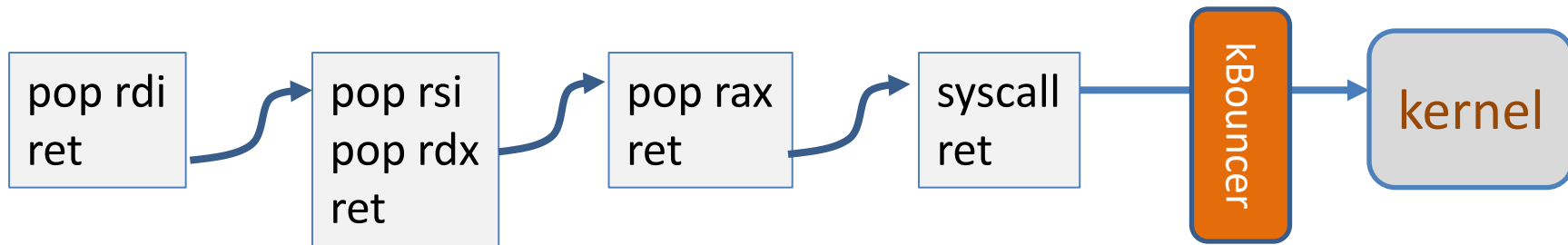
execve("/bin/sh", 0, 0): implemented using gadgets in victim code:



What to do?? Randomization

- **ASLR**: (Address Space Layout Randomization)
 - On load: randomly shift base of code & data in process memory
⇒ Attacker does not know location of code gadgets
 - Deployment: (/DynamicBase)
 - Since **Windows 8**: 24 bits of randomness on 64-bit processors
 - Base of everything must be randomized on load:
 - libraries (DLLs, shared libs), application code, stack, heap
- Other randomization ideas (not used in practice):
 - Sys-call randomization: randomize sys-call id's
 - Instruction Set Randomization (ISR)

A very different idea: kBouncer



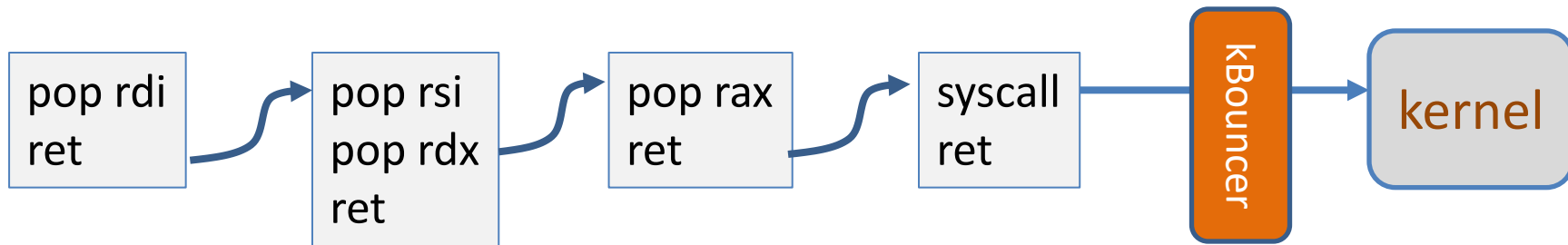
Observation: abnormal execution sequence

- ***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 ***rets*** are normal

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

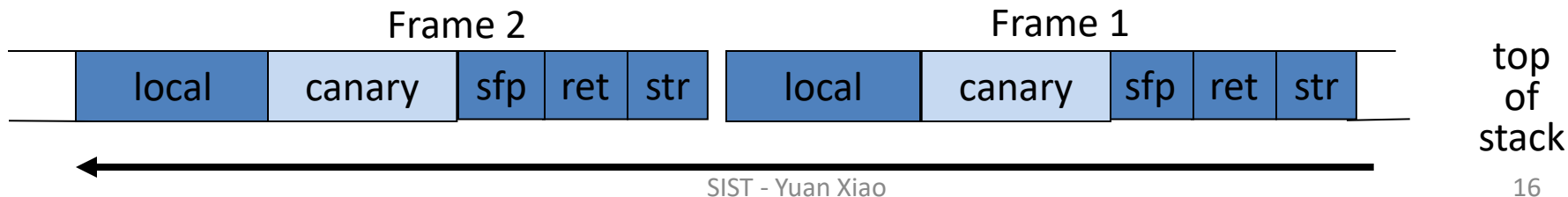
02

PART TWO

Hardening the executable

Run time checking: StackGuard

- Many run-time checking techniques ...
 - we only discuss methods relevant to overflow protection
- Method 1: StackGuard
 - Run time tests for stack integrity.
 - Embed “canaries” in stack frames and verify their integrity prior to function return.



Canary Types

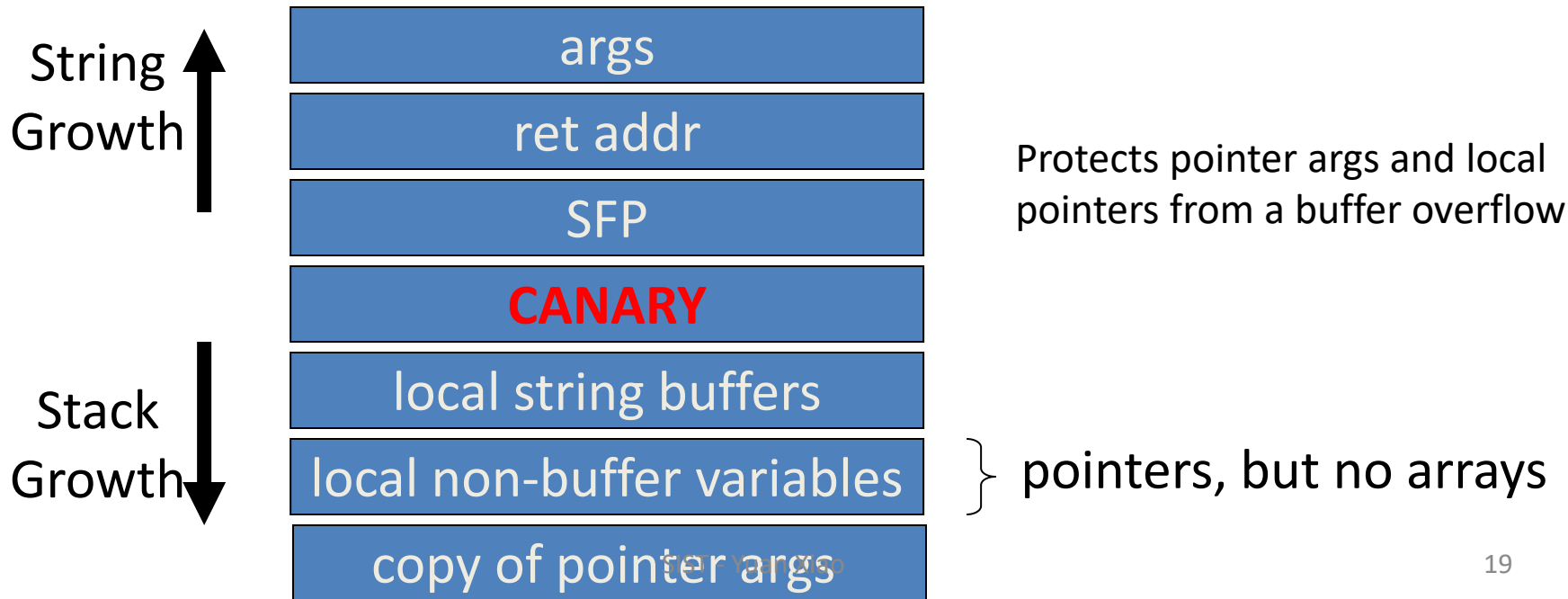
- Random canary:
 - Random string **chosen at program startup**
 - Insert canary string into every stack frame
 - Verify canary before returning from function
 - Exit program if canary changed. Turns potential exploit into DoS.
 - To corrupt, attacker must learn/guess current random string
- Terminator 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 (BufferSecurityCheck)

Compiler /GS option:

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

Function prolog:

```
sub esp, 4    // allocate 4 bytes for cookie  
mov eax, DWORD PTR ___security_cookie  
xor eax, esp  // xor cookie with current esp  
mov DWORD PTR [esp+4], eax // save in stack
```

Function epilog:

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

Protects all stack frames, unless can be proven unnecessary

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
app. code ASLR



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)

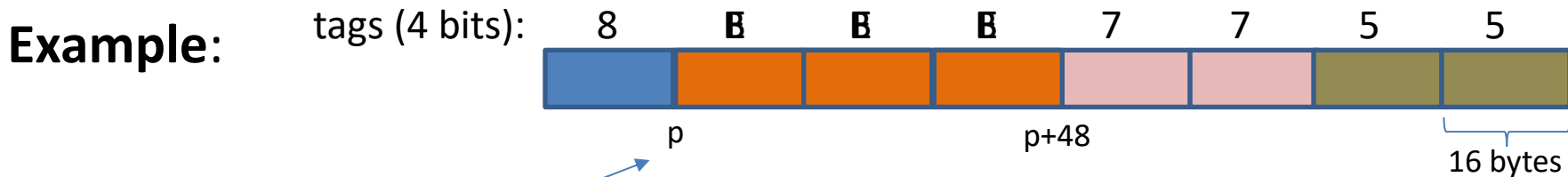
- Idea: (1) every 64-bit **memory pointer** P has a 4-bit “tag” (in top byte)
(2) every 16-byte user **memory region** R has a 4-bit “tag”

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 (used by malloc and free)
- ADDG, SUBG: pointer arithmetic on an address preserving tags

Tags prevent buffer overflows and use after free



- (1) `char *p = new char(40);` // `p = 0x B000 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)

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

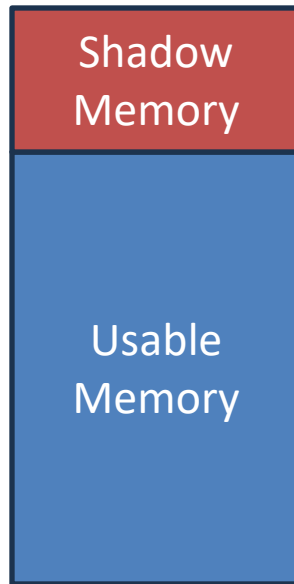
AddressSanitizer (ASan): a software tool

For every 8 bytes of usable memory, allocate one byte in shadow to record its allocation status:

- 0: all 8 bytes are allocated (e.g., by malloc)
- $1 \leq k \leq 7$: first k bytes are allocated
- negative number: 8 bytes should not be accessed

Compiler places a guard before every memory access. Example:

```
ShadowAddr = (Addr >> 3) + ShadowOffset; // address in shadow mem
if (*ShadowAddr != 0) ReportAndCrash(Addr); // crash if not fully alloc.
t = *Addr; // program can now read/write address Addr
```



Shadow memory eats up $1/8^{\text{th}}$ of physical memory $\Rightarrow$ expensive

- ASan is mostly used when fuzzing a program (e.g., Chrome)

AddressSanitizer (ASan): a software tool

Using ASan to detect a buffer overflow on stack or heap:



tags: -1 000004 -1 000000006 -1 0000 -1

in shadow
memory

$5 \times 8 + 4 = 44$ bytes

$8 \times 8 + 6 = 70$ bytes

$4 \times 8 = 32$ bytes

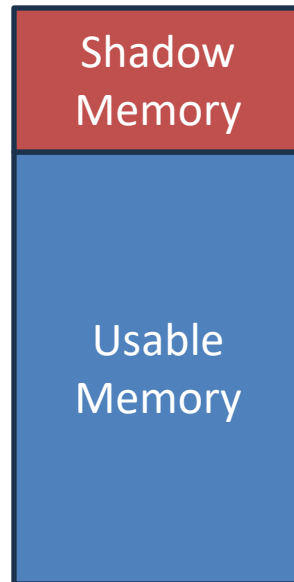
overflow will cause an access to a **red zone** (rz) $\Rightarrow$ crash program

after mem2 is freed:



tags: -1 000004 -1 -1 -1 -1 -1 -1 -1 -1 0000 -1

use-after-free at mem2 $\Rightarrow$ crash program





03

PART THREE

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

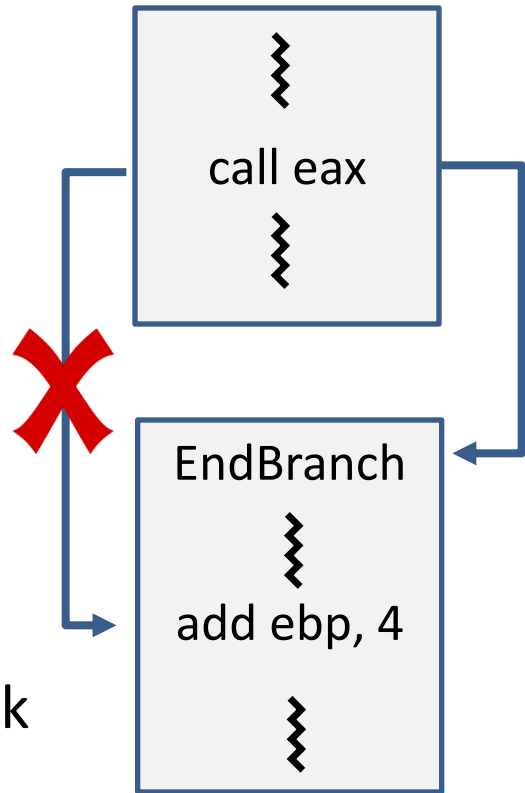
```
rep stosd  
mov     esi, [esi]  
mov     ecx, esi      ; Target  
push    1  
call    @_guard_check_icall@4 ; _guard_check_icall(x)  
call    esi  
add     esp, 4  
xor     eax, eax
```

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
- Ensures an indirect JMP or CALL can only go
to a valid target address $\Rightarrow$ no func. ptr. hijack
(compiler inserts EndBranch at valid locations)



CFG, EndBranch, BTI: limitations

Poorly implemented CFG

- 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(8)  
call esi  
add esp, 4  
xor eax, eax
```

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: $\text{tag} = \text{AES}(k, (\text{jump-address}, 0 \parallel \text{source-address}))$

on stack: $\text{tag} = \text{AES}(k, (\text{jump-address}, 1 \parallel \text{stack-frame}))$

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 ...  
}
```



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