Review

Yajin Zhou (http://yajin.org)

Zhejiang University

- 2021_01_basic_ principles.pdf
- 2021_01_overview.pdf
- 2021_02_buffer_overflow.pdf
- 2021_03_ret2libc.pdf
- 2021_04_rop.pdf
- 2021_04_rop_example.pdf
- 2021_05_bittau-brop-slides.pdf
- 2021_05_blindrop.pdf
- 2021_06_dow.pdf
- 2021_06_format_string.pdf
- 2021_07_integer_heap_overflow.pdf
- 2021_07_uaf_type_confusion.pdf
- 2021_08_00_set_uid.pdf
- 2021_08_1_environment_variables.pdf
- 2021_08_2_shellshock.pdf
- 2021_08_03_heartbleeding.pdf
- 2021_09_LLVM.pdf
- 2021_09_static_analysis.pdf
- 2021_09_symbolic_execution.pdf
- 2021_09_taint_analysis.pdf
- 2021_cfi.pdf
- 2021_HARP-Current.pdf
- 2021_SFI.pdf

Basic Principles

Security vs safety

NTNU definition (Skavland Idsø and Mejdell Jakobsen, 2000):

Safety is protection against random incidents. Random incidents are unwanted incidents that happen as a result of one or more coincidences.

Security is protection against intended incidents. Wanted incidents happen due to a result of deliberate and planned act.

CIA

- Confidentiality
  - An attacker cannot recover protected data
- Integrity
  - An attacker cannot modify protected data
- Availability
  - An attacker cannot stop/hinder computation
- Accountability/non-repudiation may be used as fourth fundamental concept. It prevents denial of message transmission or receipt

Trusted Computing Base (TCB)

- A set of **hardware, firmware, and software that are critical to the security of a computer system**
    - Bugs in the TCB may jeopardize the system's security
    - E.g., a conventional e-voting machine: voting software + hardware
- **Components outside of the TCB can misbehave without affecting security**
- **In general, a system with a smaller TCB is more trustworthy**
- A lot of security research is about how to move components outside of the TCB (i.e., making the TCB smaller)
- E.g., Proof-Carrying Code removes the compiler outside of the TCB
- E.g., voter-verified paper ballots in E-voting

Threat model

- Awareness of entry points (and associated threats) Look at systems **from an attacker's perspective**
    - Decompose application: identify structure
    - Determine and rank threats
    - Determine counter measures and mitigation
- Reading material: https://www.owasp.org/index.php/Application_Threat_Modeling

The threat model defines the abilities and resources of the attacker. Threat models enable structured reasoning about the attack surface.

Policy and Enforcement

- **Policy**
  - What is (what is not) allowed
  - Who is allowed to do what
- **Enforcement**: what we do to cause policy to be followed
  - Means of enforcement
    - Persuasion, Monitoring & deterrence
    - Incentive management
    - **Technical prevention (what we are mostly interested in)**

Fundamental Security Mechanism

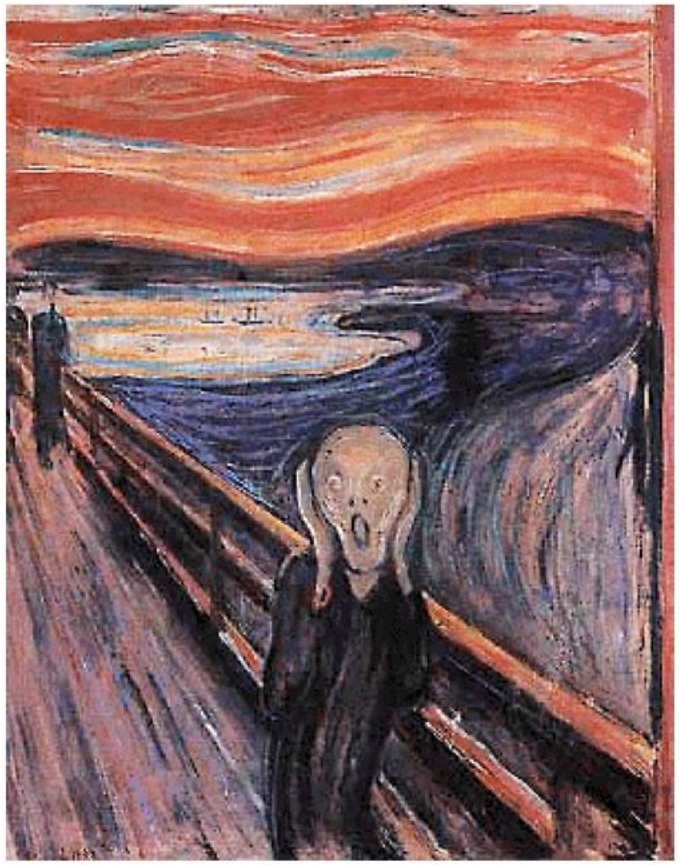
- Isolation
- Least privilege
- Fault compartments
- Trust and correctness

AAA

- Authentication: Who are you (what you know, have, or are)?
- Authorization: Who has access to object?
- Audit/Provenance: I'll check what you did.

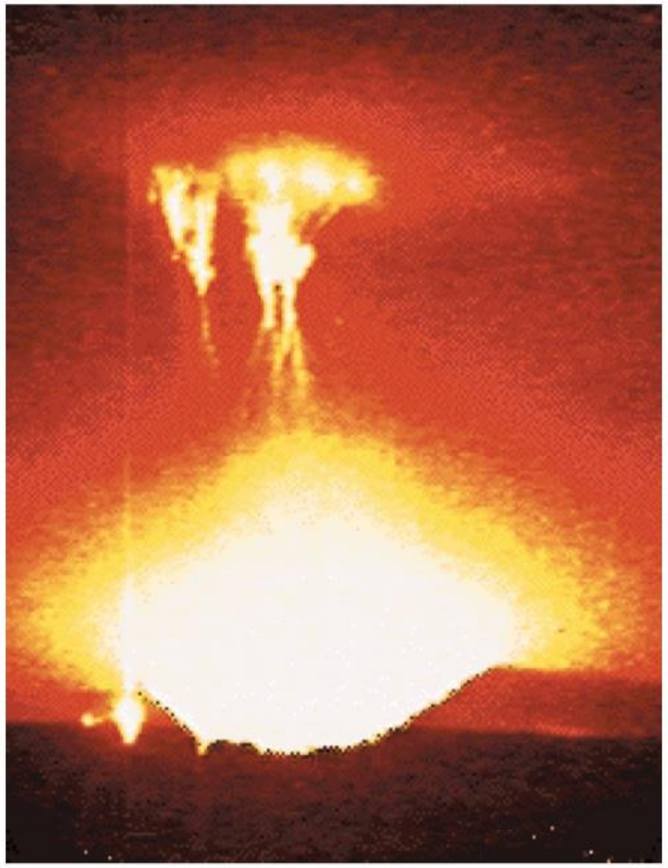
Vulnerabilities

- A vulnerability is a flaw that is accessible (threat) to an adversary who has the capability to exploit that flaw
- Are flaws alone enough for a vulnerability?
  - E.g., buffer-overflow, WEP key reuse
- What is the source of a flaw?
  - Bad software (or hardware)
  - Bad design, requirements
  - Bad policy/configuration
  - System Misuse
    - unintended purpose or environment
    - E.g., student IDs for liquor store



Attacks

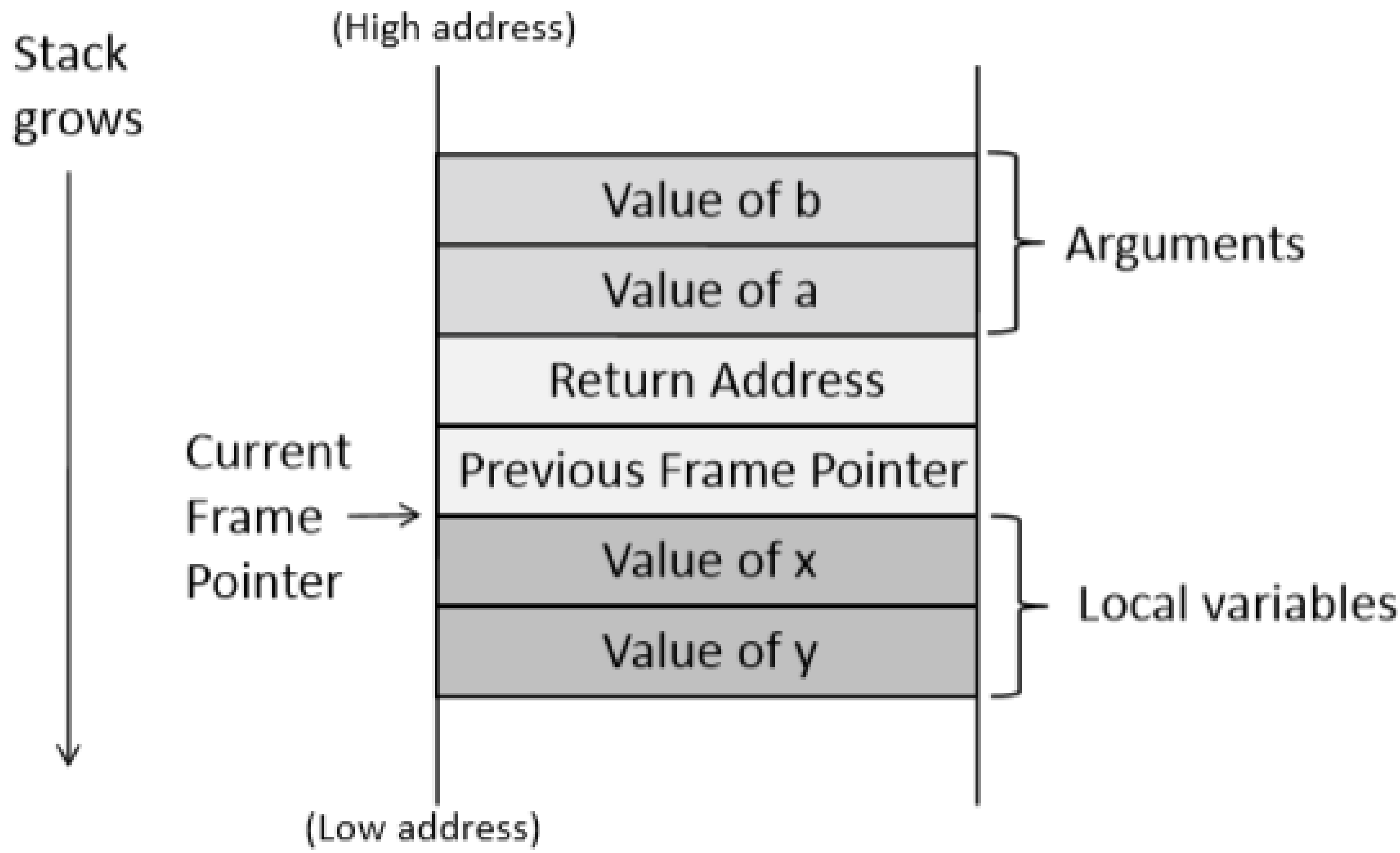
- An attack occurs when someone attempts to exploit a vulnerability
- Kinds of attacks
  - Passive (e.g., eavesdropping)
  - Active (e.g., password guessing)
  - Denial of Service (DOS)
    - Distributed DOS – using many endpoints



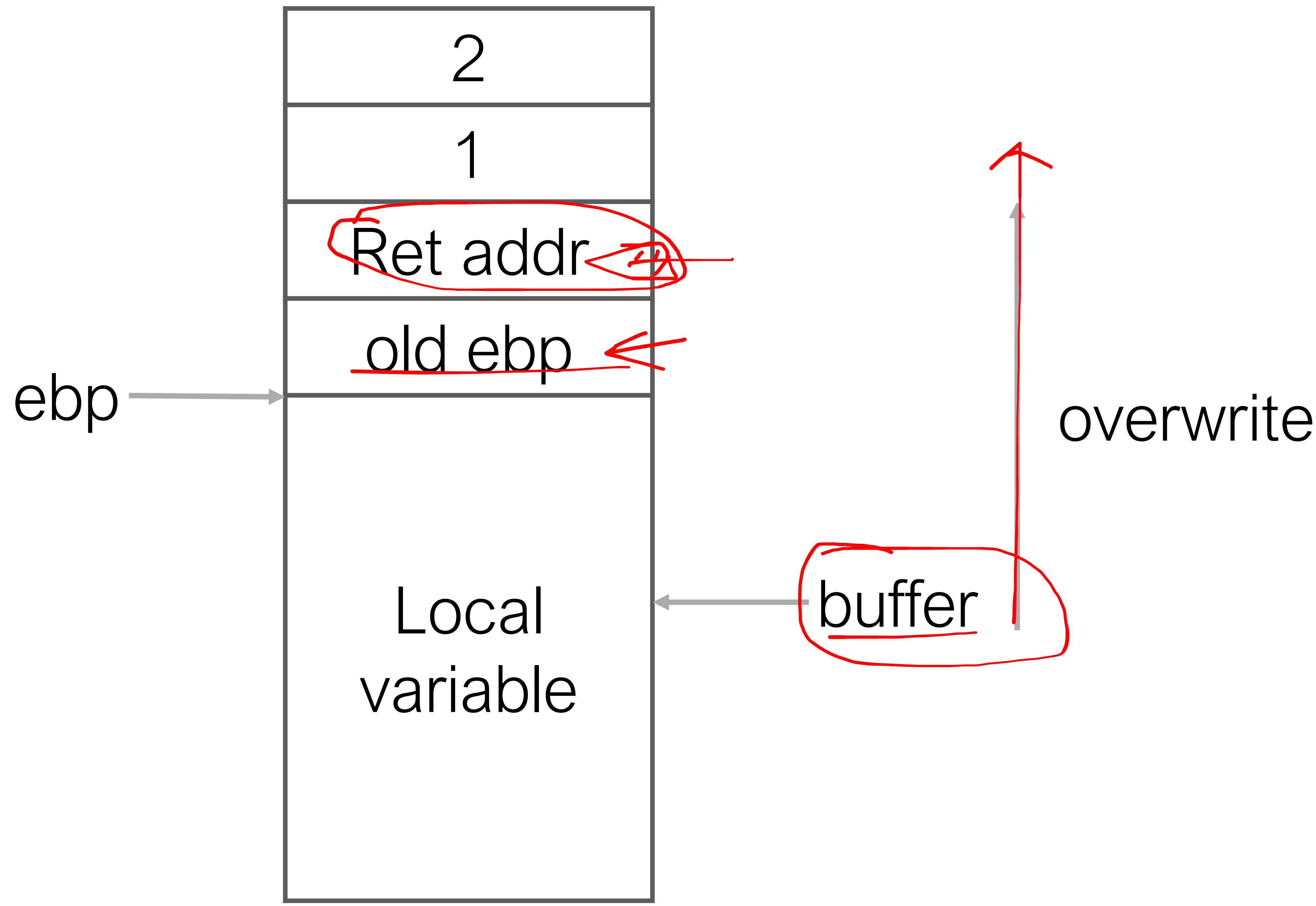
- A compromise occurs when an attack is successful
  - Typically associated with taking over/altering resources

Buffer Overflow

Stack Layout



Gets



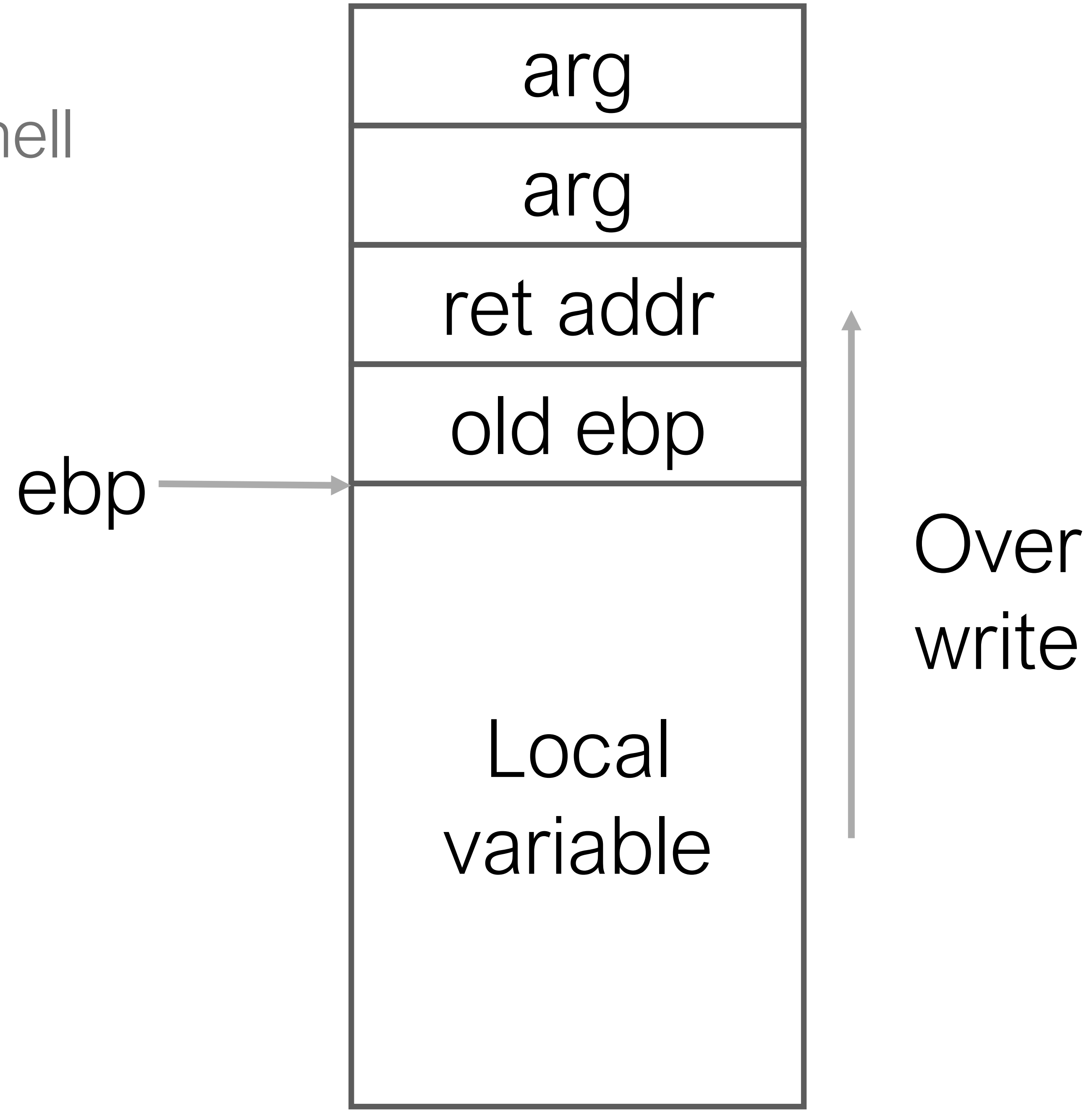
Buffer Overflow

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify
  - **return address on the stack**
  - function pointer
  - local variable
  - heap data structures

ret2libc

What We Have Learnt So Far

- Stack layout
- Overwrite the return address to shell code on the stack
- Defense
    - Stack canary
    - DEP



Runtime Mitigation: DEP (NX)

- Computer architectures follow a Von-Neumann architecture
  - Storing code as data
  - This allows an attacker to inject code into stack or heap, which is supposed to store only data
- A Harvard architecture is better for security
  - Divide the virtual address space into a **data region** and a **code region**
  - The code region is readable (R) and executable (X)
  - The data region is readable (R) and writable (W)
  - No region is both writable and executable
    - An attacker can inject code into the stack, but cannot execute it

Runtime Mitigation: DEP (NX)

- DEP prevents code-injection attacks
  - AKA Nx-bit (non executable bit), W © X
- DEP is now supported by most OSes and ISAs

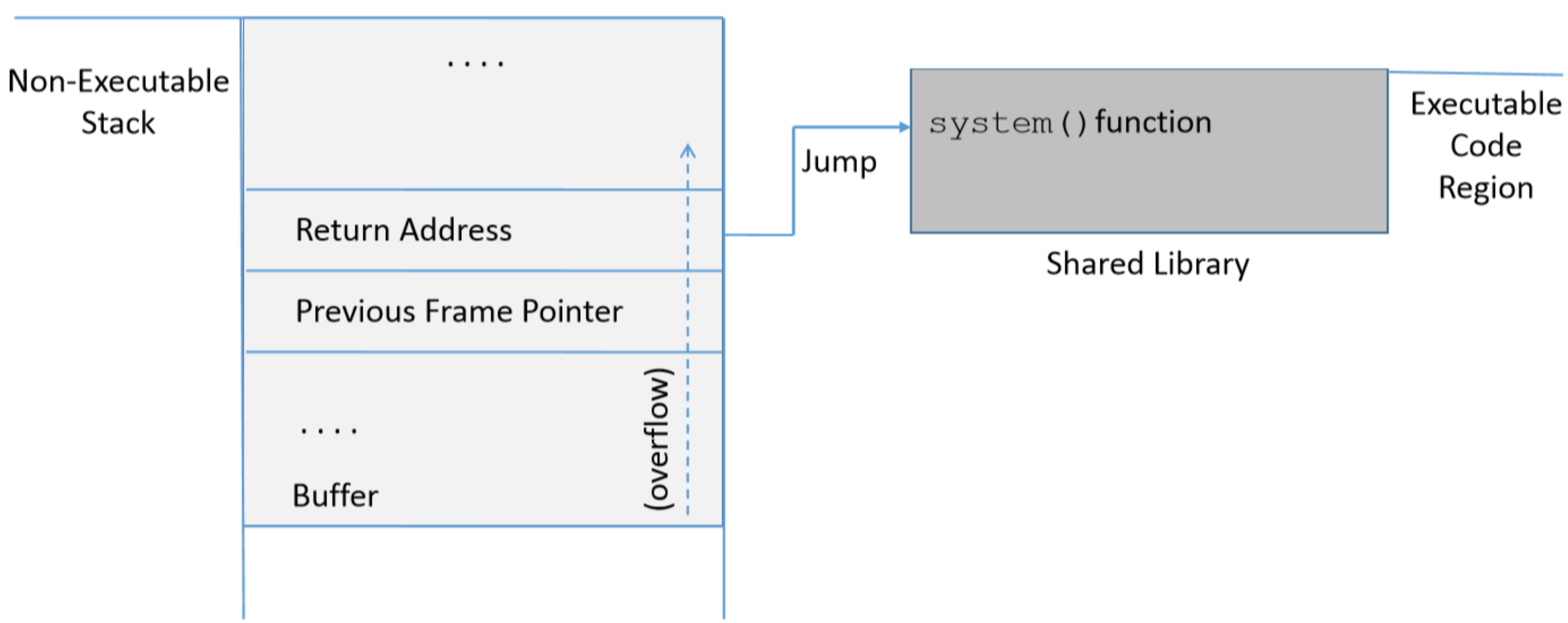
Defeating DEP: Code Reuse Attacks

- Idea: reuse code in the program (and libraries)
  - No need to inject code
- Return-to-libc: replace the return address with the address of a dangerous library function
  - attacker constructs suitable parameters on stack above return address
    - On x64, need more work of setting up parameter-passing registers
  - function returns and library function executes
    - e.g. `execve(“/bin/sh”)`
  - can even chain two library calls

- Ret2libc *without* ASLR

How to Attack: Rethink the Stack Layout

- Step I: find the address of system function
- Step II: find the string “/bin/sh”
- Step III: pass “bin/sh” to system function



ROP

Code Injection vs Code Reuse

- Ret2libc is a code reuse attack
- The difference is subtle, but significant
  - In **code injection**, we wrote the address of execve into buffer on the stack and modified return address to start executing at buffer
    - I.e., we are executing in the stack memory region
  - In **code reuse**, we can modify the return address to point to execve directly, so we continue to execute code
    - Reusing available code to do what the adversary wants

Code Reuse

- In many attacks, a code reuse attack is used as a first step to disable DEP
  - Goal is to allow execution of stack memory
  - There's a system call for that

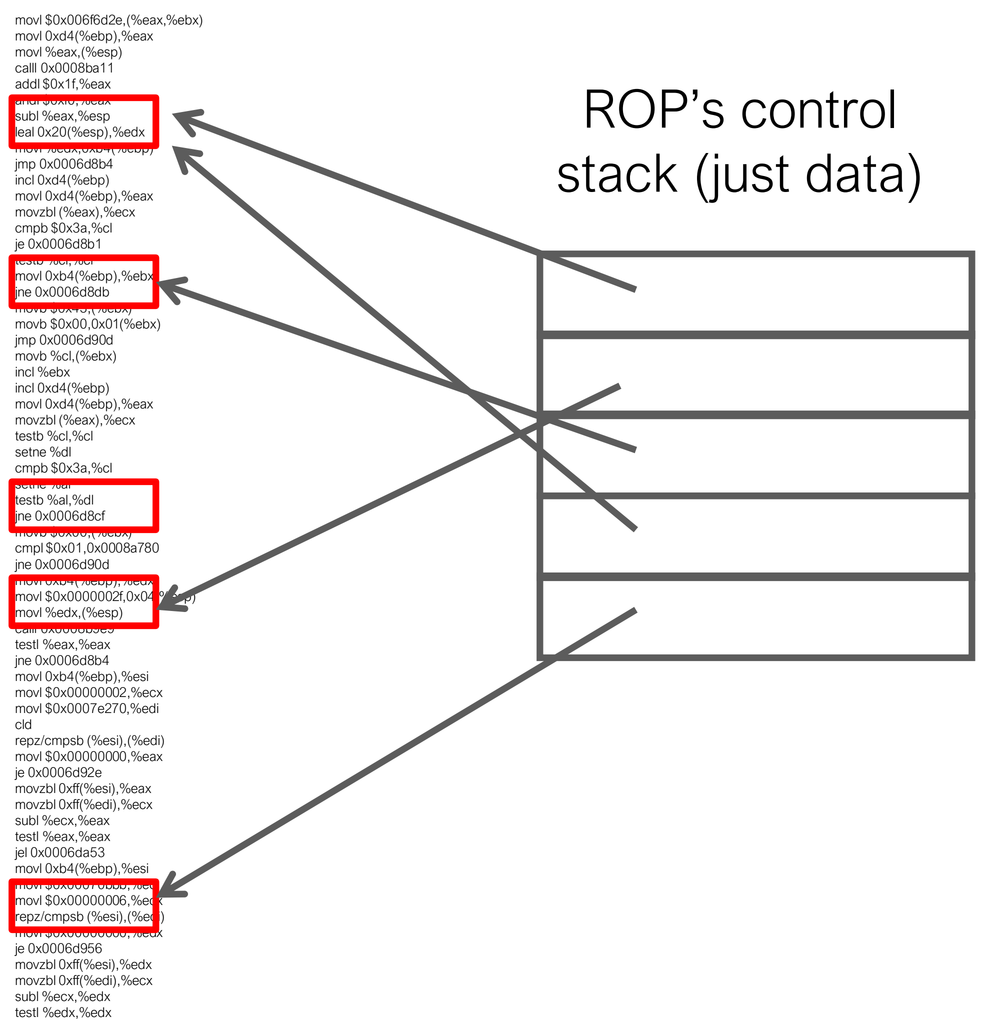
```
int mprotect(void *addr, size_t len, int prot);
```

  - Sets protection for region of memory starting at address
  - Invoke this libray API (system call) to allow execution on stack and then start executing from the injected code

Code Reuse: ROP

- Return-Oriented Programming (ROP)
  - [Shacham et al], 2008
  - **Arbitrary behavior without code injection**
  - Combine snippets of existing code (gadgets)
  - A set of Turing-complete gadgets and a way of chaining these gadgets
  - People have shown that in small programs (e.g., 16KB), they can find a turing-complete set of gadgets

ROP: Illustrated



- Use gadgets to perform general programming
  - arithmetic;
  - arbitrary control flow: jumps; loops; …

ROP Thesis

any sufficiently large program codebase

↓ ↓

arbitrary attacker computation and behavior,
without code injection

Runtime Mitigation: Randomization

- Exploits requires knowing code/data addresses
  - E.g., the start address of a buffer
  - E.g., the address of a library function
- Idea: introduce artificial diversity (randomization)
  - Make addresses unpredictable for attackers
- Many ways of doing randomization
  - Randomize **location of the stack, location of key data structures on the heap, and location of library functions**
  - Randomly pad stack frames
  - At compile time, randomize code generation for defending against ROP

Format String

Overview

- Format string vulnerability was discovered in 2001
- A class of vulnerabilities that take advantage of an attacker-controlled buffer as an argument to printf() function.
- Consequences: the attacker can perform read and write to **arbitrary memory addresses**
- There are a number of functions that accept a format string as an argument
  - Functions: fprintf, printf, vfprintf
  - Programs: syslog….

Format Specifiers

- printf ("The magic number is: %d\n", 1911);

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Look at printf() again

- Myprintf uses Narguments to denote number of arguments, and the type of input arguments is fixed

- However, printf() uses format string for this purpose

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

What if we make a mistake

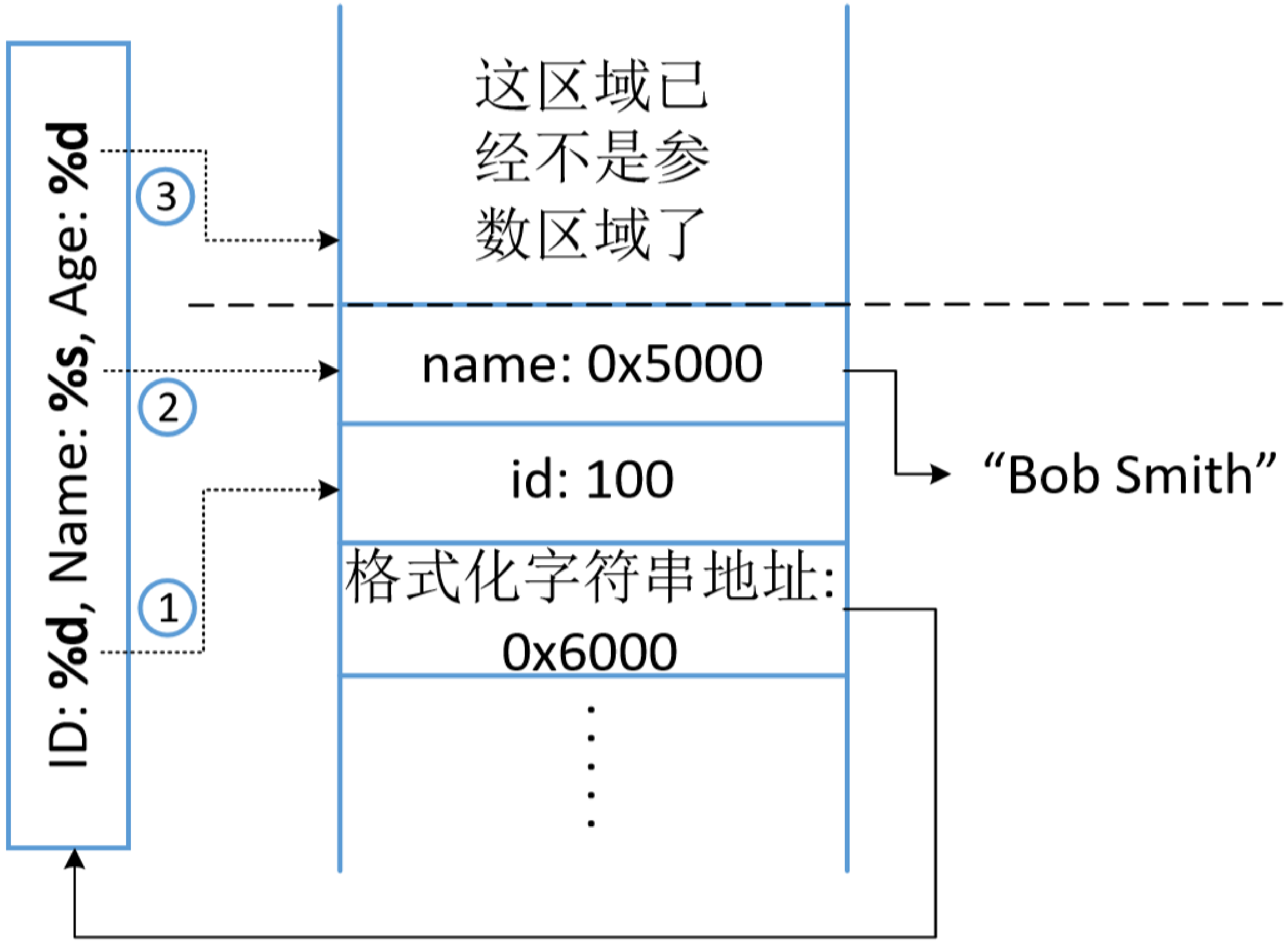


图 6.3: 缺了一个可变参数导致的情况

Integer Overflow

Integer Overflows

- An **integer overflow** occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value
- Standard integer types (signed)
  - signed char, short int, int, long int, long long int
- Signed overflow vs unsigned overflow
  - An unsigned overflow occurs when the underlying representation can no longer represent an integer value.
  - A signed overflow occurs when a value is carried over to the sign bit

Overflow Examples

```c
#include <stdio.h>
#include <limits.h>

int main(int argc, char *const *argv) {
    unsigned int ui;
    signed int si;
    ui = UINT_MAX; // 4,294,967,295;
    printf("ui = %u %x \n", ui, ui);
    ui++;
    printf("ui = %u %x \n", ui, ui);

    si = INT_MAX; // 2,147,483,647
    printf("si = %d %x \n", si, si);
    si++;
    printf("si = %d %x \n", si, si);

    ui = 0;
    printf("ui = %u %x \n", ui, ui);
    ui--;
    printf("ui = %u %x \n", ui, ui);

    si = INT_MIN; // -2,147,483,648;
    printf("si = %d %x \n", si, si);
    si--;
    printf("si = %d %x \n", si, si);

}
```

```
work@ubuntu:~/ssec20/example_code/ssec20/overflow$
ui = 4294967295 ffffffff
ui = 0 0
si = 2147483647 7fffffff
si = -2147483648 80000000
ui = 0 0
ui = 4294967295 ffffffff
si = -2147483648 80000000
si = 2147483647 7fffffff
```

Integer Overflow Example

```c
int main(int argc, char *const *argv) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buff = (char *) malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
}
```

What if the total variable is overflowed because of the addition operation?

Heap Overflow

Heap Overflows

- Another region of memory that may be vulnerable to overflows is heap memory
  - A buffer overflow of a buffer allocated on the heap is called a heap overflow

Overflowing Heap Critical User Data

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];                /* vulnerable input buffer */
    void (*process)(char *);   /* pointer to function */
} chunk t;

void showlen(char *buf) {
    int len; len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

example by Stallings

- Overflow the buffer on the heap so that the function pointer is changed to an arbitrary address

Overflow Heap Meta-Data

- Heap allocators (AKA memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers
  - Metadata adjusted during heap-management functions
    - malloc() and free()
  - Heap metadata often inlined with heap data

An Example

```c
struct toystr {
    void (* message)(char *);
    char buffer[20];
};

void print_super(char * who)
{
    printf("%s is superrr cool......\n", who);
}

void print_cool(char * who)
{
    printf("%s is cool!\n", who);
}

void print_meh(char * who)
{
    printf("%s is meh...\n", who);
}
```

```c
int main(int argc, char * argv[])
{
    struct toystr * coolguy = NULL;
    struct toystr * lameguy = NULL;

    coolguy = malloc(sizeof(struct toystr));
    lameguy = malloc(sizeof(struct toystr));

    coolguy->message = &print_cool;
    lameguy->message = &print_meh;

    printf("Input coolguy's name: ");
    fgets(coolguy->buffer, 200, stdin);
    coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

    printf("Input lameguy's name: ");
    fgets(lameguy->buffer, 20, stdin);
    lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

    coolguy->message(coolguy->buffer);
    lameguy->message(lameguy->buffer);

    return 0;
}
```

Use After Free and Double Free

*adapted from slides by Trent Jaeger

Use After Free

- Error: Program frees memory on the heap, but then references that memory as if it were still valid
  - Adversary can control data written using the freed pointer
- AKA use of dangling pointers

Use After Free

```c
int main(int argc, char **argv) {
    char *buf1, *buf2, *buf3;

    buf1 = (char *) malloc(BUFSIZE1);

    free(buf1);

    buf2 = (char *) malloc(BUFSIZE2);
    buf3 = (char *) malloc(BUFSIZE2);
    strncpy(buf1, argv[1], BUFSIZE1-1);
    …

}
```

What happens here?

Use After Free

- Adversary chooses function pointer value
- Adversary may also choose the address in x->buf
- Become a popular vulnerability to exploit – over 60% of CVEs in 2018

Prevent Use After Free

- What can you do that is not too complex?
  - You can set all freed pointers to NULL
    - Getting a null-pointer dereference if using it
    - Nowadays, OS has built-in defense for null-pointer deference
  - Then, no one can use them after they are freed
  - Complexity: need to set all aliased pointers to NULL

Related Problem: Double Free

```c
main(int argc, char **argv)
{
    …
    buf1 = (char *) malloc(BUFSIZE1);
    free(buf1);
    buf2 = (char *) malloc(BUFSIZE2);
    strncpy(buf2, argv[1], BUFSIZE2-1);
    free(buf1);
    free(buf2);
}
```

What happens here?

Double Free

- Free buf1, then allocate buf2
  - buf2 may occupy the same memory space of buf1
- buf2 gets user-supplied data

```
strncpy(buf2, argv[1], BUFSIZE2-1);
```

- Free buf1 again
  - Which may use some buf2 data as metadata
  - And may mess up buf2's metadata
- Then free buf2, which uses really messed up metadata

Introduction to program analysis

Program Analysis

- Any *automated* analysis *at compile or dynamic time* to find potential *bugs*
- Broadly classified into
  - Dynamic analysis
  - Static analysis

Dynamic Analysis

- Analyze the code when it is running
  - Detection
    - E.g., dynamically detect whether there is an out-of-bound memory access, for a particular input
  - Response
    - E.g., stop the program when an out-of-bound memory access is detected

Dynamic Analysis Limits

- Major advantage
  - After detecting a bug, it is a real one
  - No false positives
- Major limitation
  - Detecting a bug for a particular input - coverage
  - Cannot find bugs for uncovered inputs
  - If (input == 0x134576) {bug()} else {normal(); }

Question

- Can we build a technique that identifies **all bugs**?
  - Turns out that we can: static analysis
  - Is this real? What’s the potential issue?

Static Analysis

- Analyze the code before it is run (during compile time)
- Explore **all possible executions** of a program
  - **All possible inputs**
- **Approximate all possible states**
  - Build **abstractions** to “run in the aggregate”
  - Rather than executing on concrete states
  - Finite-sized abstractions representing a collection of states
- But, it has its own major limitation due to approximation
  - Can identify many false positives (not actual bugs)

Static Analysis

- Broad range of static-analysis techniques:
  - simple syntactic checks like **`grep`**

```
grep " gets(" *.cpp
```

- More advanced greps: ITS4, FlawFinder
  - A database of security-sensitive functions
    - gets, strcpy, strcat, …
    - For each one, suggest how to fix

Static Analysis

- More advanced analyses take into account semantics
  - dataflow analysis, abstract interpretation, **symbolic execution**, constraint solving, model checking, theorem proving
  - Commercial tools: Coverity, Fortify, Secure Software, GrammaTech

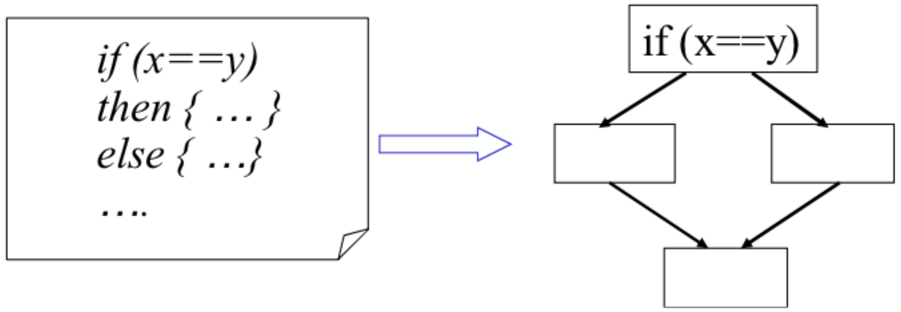
Control Flow Analysis

Program Control Flow

- Control flow
  - Sequence of operations
  - Representations
    - Control flow graph
    - Control dependency
    - Call graph
  - Control flow analysis
    - Analyzing program to discover its control structure

Control Flow Graph

- CFG models flow of control in the program (procedure)
- G = (N, E) as a directed graph
    - Node n $\in$ N: basic blocks
        - A basic block is a maximal sequences of stmts with a single entry point, single exit point and no internal tranches
        - For simplicity, we assume a unique entry node n_0 and a unique exit node n_f in later discussions
    - Edge e=(ni, nj) $\in$ E: possible transfer of control from block ni to block nj



Basic Blocks

- Definition
  - A basic block is a maximal sequence of consecutive statements with a single entry point, a single exit point, and no internal branches
- Basic unit in control flow analysis
- Local level of code optimizations
  - Redundancy elimination, register-allocation
- For security: reachability analysis, liveness analysis …

Complications in CFG Construction

- Function calls
  - Instruction scheduling may prefer function calls as basic block boundaries
  - Special functions as setjmp() and longjmp()
- Exception handling
- Ambiguous jump
  - Jump r1 //target stored in register r1
  - Static analysis may generate edges that never occur at runtime

Call Graph

- So far looked at intraprocedural analysis: a single function
- **Inter-procedural analysis** uses calling relationships **among** procedures
  - Enables more precise analysis information

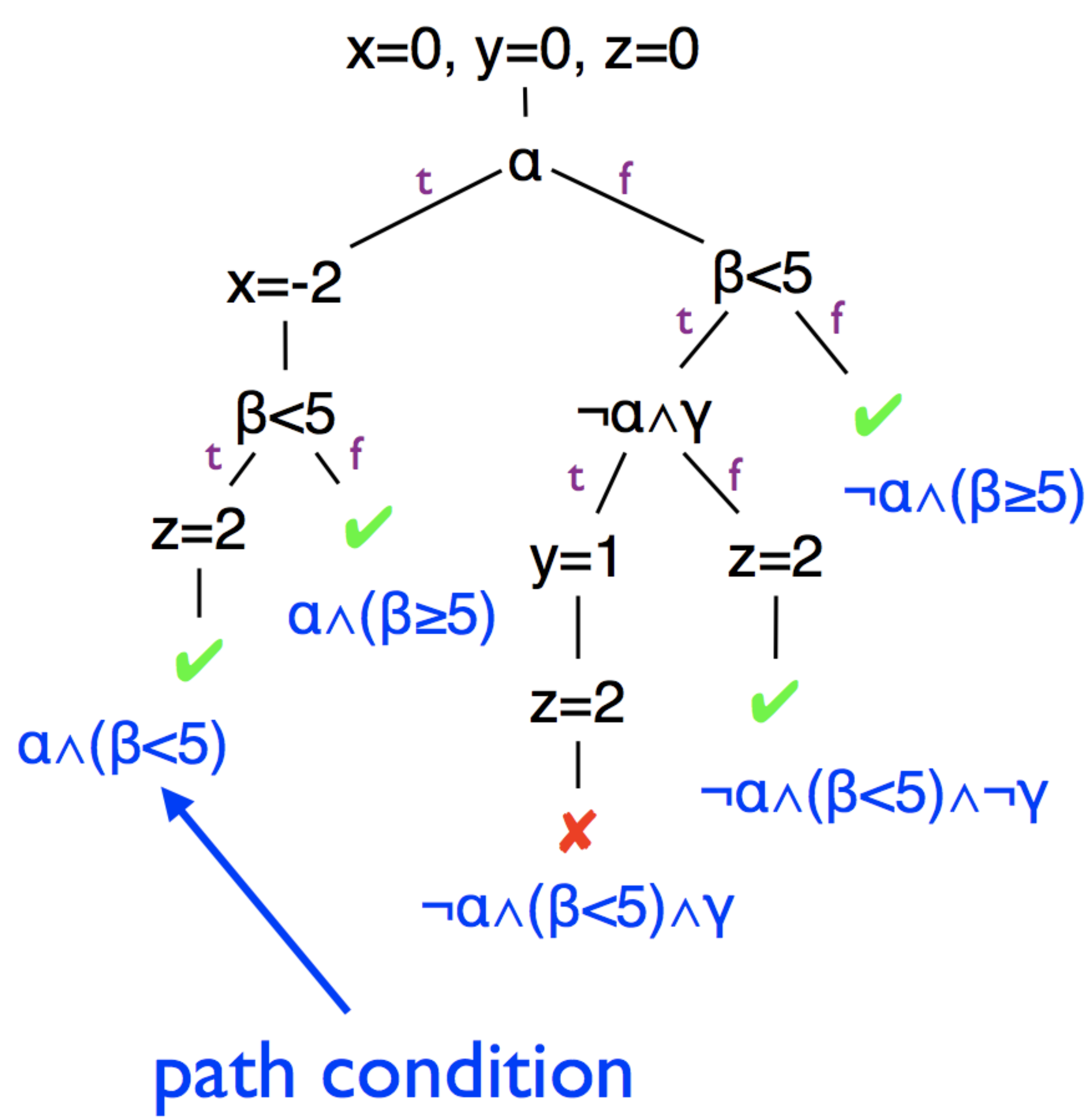
Call Graph

- First problem: **how do we know what procedures are called from where?**
  - Especially difficult in higher-order languages, languages where functions are values **(function pointer)**
  - We’ll ignore this for now, and return to it later in course...
- Let’s assume we have a (static) **call graph**
  - Indicates which procedures can call which other procedures, and from which program points.

Symbolic/Concolic Execution

Symbolic Execution Example

```
int a = α, b = β, c = γ;
                 // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Insight

- Each symbolic execution path stands for many actually program runs
  - In fact, exactly the set of runs whose concrete values satisfy the path condition
  - Thus, we can cover a lot more of the program's execution space than testing

Three Challenges

- Path explosion
- Complex Code and Environment Dependencies
- Constraint solving

Taint Analysis

Confidentiality vs. Integrity

Confidentiality and integrity are dual problems for information flow analysis

(Focus of this lecture: Confidentiality)

Taint Sources and Sinks

- **Possible sources:**
  - Variables
  - Return values of a particular function
  - Data from a type of I/O stream
  - Data from a particular I/O stream
- **Possible sinks:**
  - Variables
  - Parameters given to a particular function
  - Instructions of a particular type (e.g., jump instructions)

Taint Propagation

1) **Explicit flows**

For every operation that produces a new value, propagate labels of inputs to label of output:

$$label(result) \leftarrow label(inp_1) \oplus \ldots \oplus label(inp_2)$$

Taint Propagation (2)

2) Implicit flows

- Maintain security stack S: Labels of all values that influence the current flow of control
- When x influences a branch decision at location loc, push $label(x)$ on S
- When control flow reaches immediate post-dominator of loc, pop $label(x)$ from S
- When an operation is executed while the S is non-empty, consider all labels on S as input to the operation

CFI

Motivation

- Code injection
  - W^X
- Code reuse
  - ASLR
  - CFI

Control-flow integrity

- CFI is a security policy
- Execution must follow a path of a Control-Flow Graph
- CFG can be pre-computed
  - source-code analysis
  - binary analysis
  - execution profiling
- **Forward-edge and backward-edge**
- But how can we enforce this extracted control-flow?

Limitation

- Overhead is high
- Precise CFG construction is hard (or even impossible)

CFI in real systems

- Control Flow Guard
- LLVM
- Hardware features
  - Shadow stack
  - IBT: indirect branch tracking

SFI

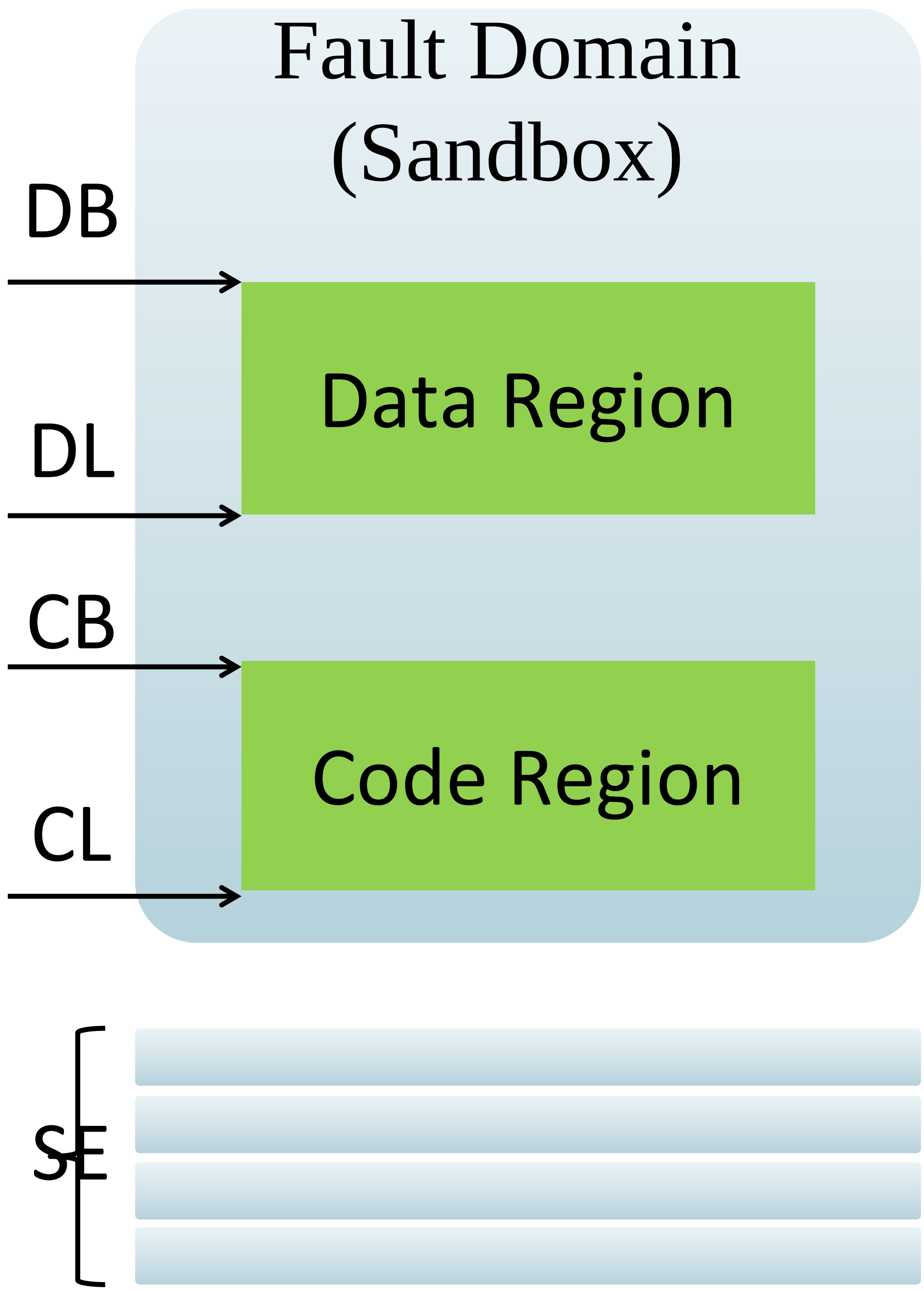
Isolation via Protection Domains

- A fundamental idea in computer security
  - [Lampson 74] “Protection”
- Structure a computer system to have multiple **protection domains**
  - Each domain is given a set of privileges, according to its trustworthiness

Software-Based Fault Isolation (SFI)

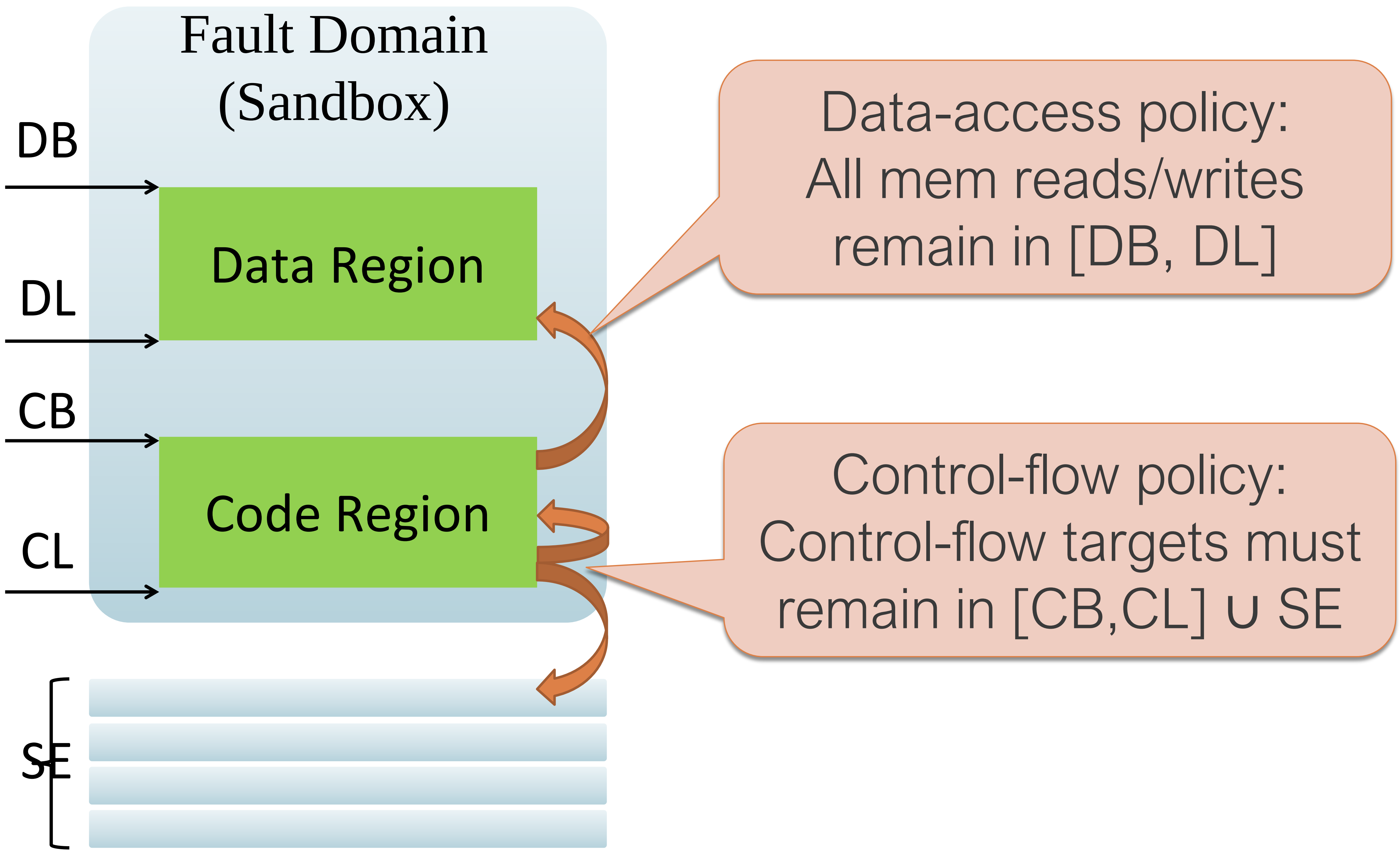
- Introduced by [Wahbe et al. 93] for MIPS
    - PittSFIeld [McCamant & Morrisett 06] extended it to x86
- SFI isolation is within the same process address space
    - Each protection domain has a designated memory region
    - Same process: avoiding costly context switches
- Implementation by inserting software checks before critical instructions
    - E.g., memory reads/writes, indirect branches.
- Pros: fine grained, flexible, low context-switch overhead
- Cons: may require some compiler support and software engineering effort

The SFI Sandbox Setup



- Data region (DR): [DB,DL]
  - Holds data: stack, heap
- Code region (CR): [CB,CL]
  - Holds code
- Safe External (SE) addresses
  - Host trusted services that require higher privileges
  - Code can jump to them for accessing resources
- DR, CR, and SE are disjoint

The SFI Policy



SFI Enforcement Overview

- *Dangerous instructions*: memory reads, memory writes, control-transfer instructions
  - They have the potential of violating the SFI policy
- An SFI enforcement
  - Checks every dangerous instruction to ensure it obeys the policy
- Two general enforcement strategies
  - Dynamic binary translation
  - Inlined reference Monitors

SFI Optimizations

- Address masking
- Data guard

Control-Flow Policy

- Recall the policy: control-flow targets must stay in [CB,CL] ∪ SE
- However, when using the IRM approach for SFI enforcement
  - Must also restrict the control flow to disallow bypassing of guards

Risk of Indirect Branches

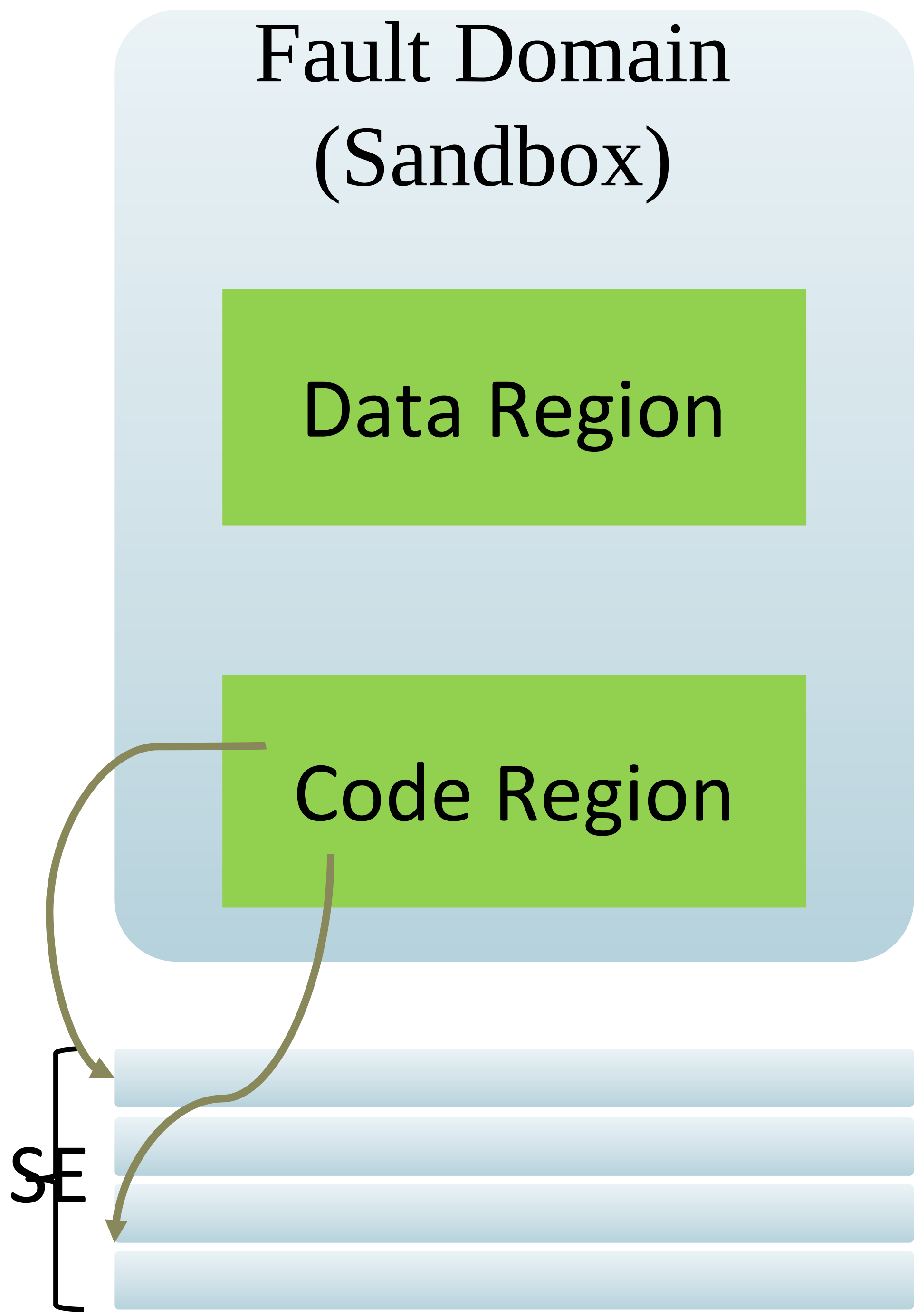
l1:	r10 := r1 + 12
l2:	r10 := dGuard(r10)
l3:	mem(r10) := r2

- Worry: what if there is a return instruction somewhere else and the attacker corrupts the return address so that the return jumps to l3 directly?
  - Then the attacker bypasses the guard at l2!
  - If attacker can further control the value in r10, then he can write to arbitrary memory location

Risk of Indirect Branches

- In general, any **indirect branch** might cause such a worry
  - If not carefully checked, it may bypass the guard
- Indirect branches include
  - Indirect calls (calls via register or memory operands)
  - Indirect jumps (jumps via register or memory operands)
  - Return instructions
- In contrast, direct branches are easy to deal with
  - Targets of a direct branch encoded in the instruction; can statically inspect the target

Allow Only Controlled Interaction



- The sandboxed code can jump to a pre-defined set of SE (Safe External) addresses
- Each SE address holds a trusted service
  - E.g., service for invoking OS syscalls (fopen, fread, …)
  - E.g., service for allowing communication with other fault domains