New trends in system software security

Secappdev 2019

Frank Piessens, February 18, 2019

Introduction

What is ICT security?

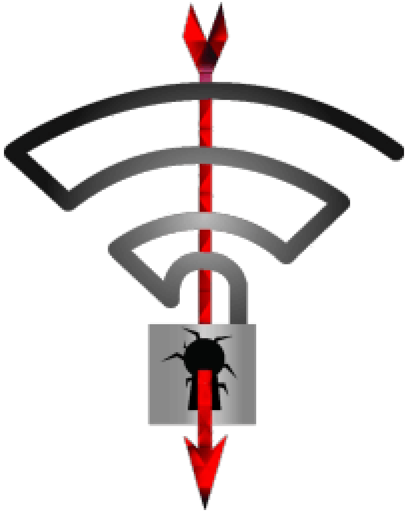
- The field of ICT security addresses the problem of
  - *Maintaining* ***desirable properties*** *of ICT systems in the presence of* ***intelligent adversaries*** *trying to break these properties*
- In practice:
  - "Desirable properties" are hard to nail down
  - But we recognize security failures when we see them
    - Viruses, worms, defacements, data leaks, ransomware, DDOS, jailbreaks, ...

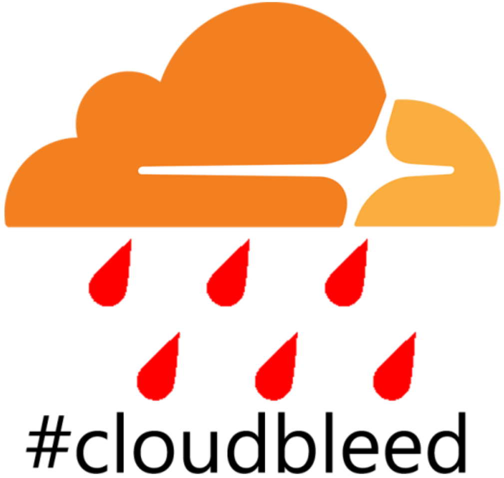
An important underlying cause: insecure software

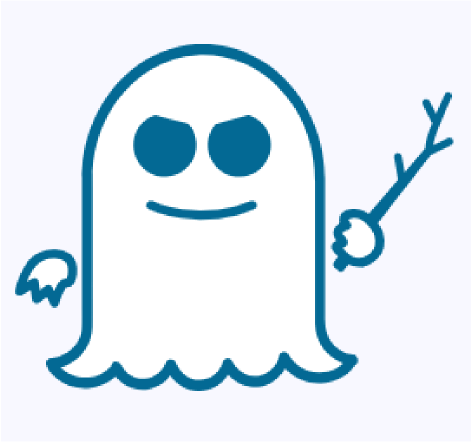
- Software implementation vulnerability =
  - A defect in software code (a “bug”) that can be exploited by an attacker to break some security objective of the software
- Around 100.000 such vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE) list
  - Buffer overflows, SQL injection, cross-site scripting, race conditions, side-channel vulnerabilities, information leaks, incomplete access mediation, cross-site scripting, double free, …


DIRTY COW


Image Tragick




#cloudbleed



Vulnerabilities in system software

- System software (operating systems, low-level libraries, servers, browsers, …) is often programmed in performance-friendly, “close-to-hardware” languages like C / C++
- These languages are infamous for the broad class of *memory management vulnerabilities*
- While many mitigation techniques are deployed, these vulnerabilities still represent a major threat
- Moreover, a vulnerability in system software affects the security of all applications running on the system

Purpose of this lecture:

- Provide insight in a number of new trends in **system software security**
  - New hardware architectures for the safe execution of C
    - Capability machines
  - New language designs for systems programming, safe by design
    - Rust ownership types
  - [If time: new attacks]
    - [Spectre style attacks]

Overview of the rest of the talk

- System model and attacker model
  - Recap of how C-like languages are executed on standard processors
  - Interactive attacker model
- Memory capabilities for run-time security
- Ownership types for compile-time security
- [The next wave of attacks]
- Conclusions

System model / attacker model / security objective

- A rigorous study of security requires:
  - A **system model**: a model of the system under attack that is sufficiently detailed to explain the attacks one cares about
    - Our systems are essentially C programs, but we need to model how compilation works to explain relevant attacks
  - An **attacker model**: a precise description of what an attacker can and cannot do
  - A **security objective**: either a description of system properties to be maintained, or of attacks/threats to be avoided

Platform model

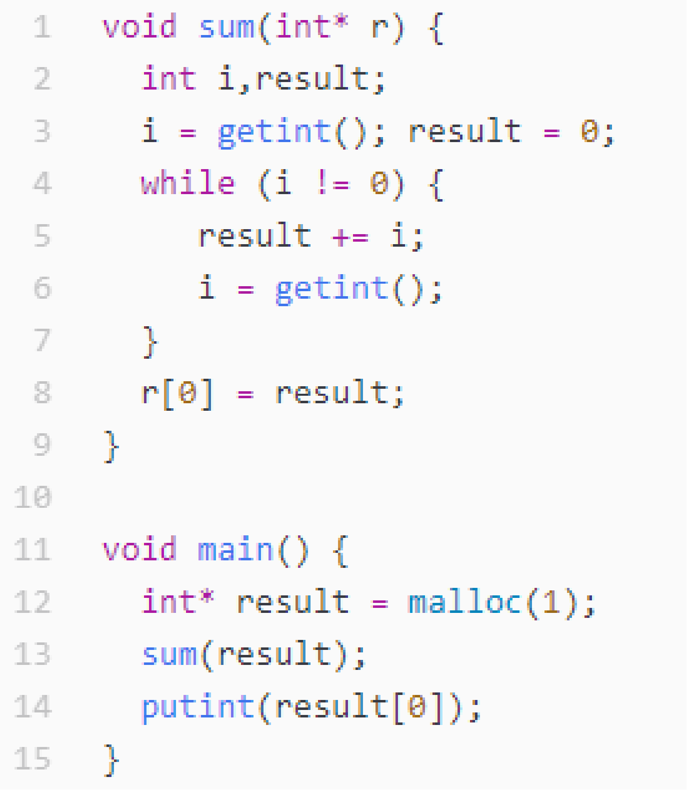
- Target platform consists of:
  - A memory of MAX words (addresses 0 .. MAX-1)
    - Making abstraction of issues like word-size, padding, …
  - A CPU with
    - Registers:
      - PC, x0 (=0), x1(=ra), x2(=sp), x3(=gp), x4,…
    - Typical RISC-like instructions
      - Arithmetic/logical/shift
      - Memory access
      - Conditional/unconditional branch
    - Instructions can be encoded as words
- Details vary across platforms

Example instruction	**Semantics**
`add x5,x6,x7`	x5 = x6 + x7
`addi x4,x5,10`	x4 = x5 + 10
`lw x4,50(x5)`	x4 = M[x5 + 50]
`sw x5,30(x4)`	M[x4+30] = x5
`beq x5,x6,12`	if x5==x6 goto PC+12
`jal 12`	x1 = PC+1; goto PC+12
`jalr 10(x5)`	x1 = PC+1; goto x5+10

Source code model

- Simple C-like language
    - Types: char, int, void, pointers (e.g. char*, int**, …), arrays (e.g. char[10])
    - Local and global variables
        - Array variable is a pointer to the first element of the array
    - Statements and expressions:
        - Constants, variables, logical and arithmetic expressions, array indexing
        - If / while / sequencing / blocks / assignment / function calls
        - Library functions for I/O and memory management:
            - getchar(), putchar(),gets(), printf() + other typical C functions for I/O – we will just use getint() and putint()
            - malloc() and free()

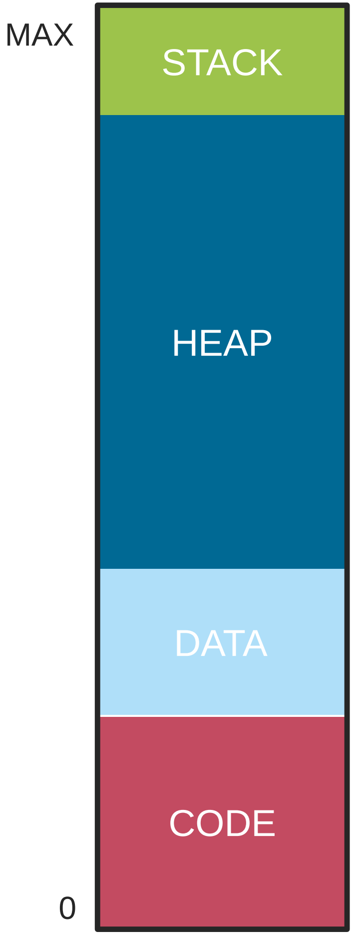
Example source program



```
void sum(int* r) {
  int i,result;
  i = getint(); result = 0;
  while (i != 0) {
     result += i;
     i = getint();
  }
  r[0] = result;
}

void main() {
  int* result = malloc(1);
  sum(result);
  putint(result[0]);
}
```

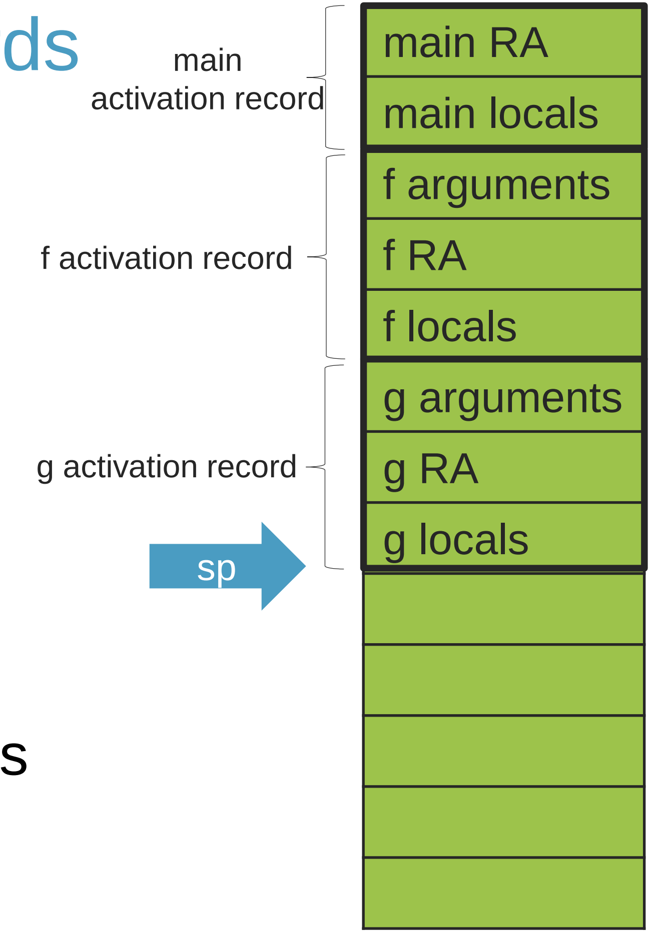
Compilation: Memory layout

- A compiled program uses memory for 4 purposes:
  - CODE: contains compiled machine code
  - DATA: contains global variables
  - HEAP: contains dynamically allocated program data
  - STACK: contains the call stack that tracks function invocations



Compilation: Stack activation records

- Call stack is a stack of activation records, each containing:
  - Call arguments
  - Return address
  - Space for local variables
- NOTE: many real-world compilation details elided (frame pointer, using registers, …)

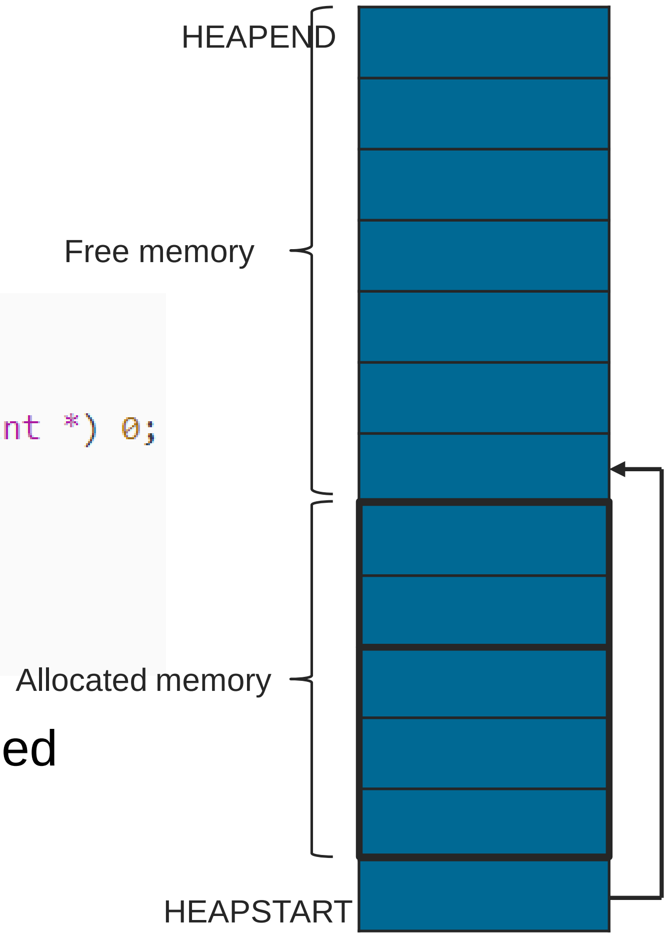


Compilation: malloc() and the heap

- Simplified malloc() implementation:

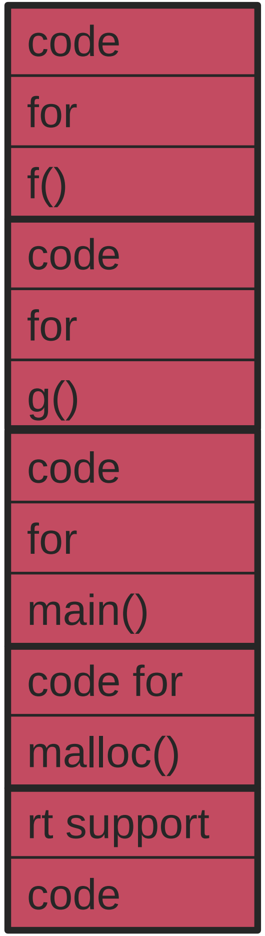
```c
int* malloc(int n) {
  int* result;
  if ((((int *) HEAPSTART[0]) + n) >= HEAPEND) return (int *) 0;
  result = (int *) HEAPSTART[0];
  HEAPSTART[0] = (int) (result + n);
  return result;
}
```

- Note: many real-world implementation details elided (supporting free(), supporting virtual memory,…)



Compilation: the CODE section

- Code for every function is compiled separately
  - Prologue: allocates space for activation record
  - Code for the body
  - Epilogue: put result in designated register, clear space for activation record
- We do not show the implementation of I/O
  - Could be syscall, instruction, memory mapped, …



code
for
f()

code
for
g()

code
for
main()

code for
malloc()

rt support
code

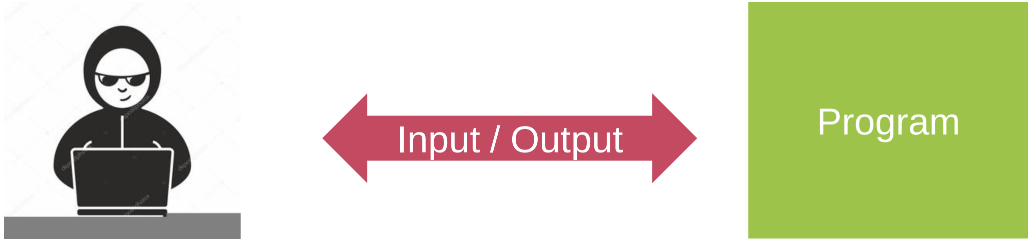
Example Compilation

```
void sum(int* r) {
  int i,result;

  i = getint();

  result = 0;
  while (i != 0) {
       result += i;
       i = getint();
  }
  r[0] = result;
}
```

```
sum:
  addi sp,sp,-4    // activation record: arg,ra,i,result
  sw x10, 3(sp)    // save argument in activation record
  sw ra, 2(sp)     // save return address in act record

  jal getint       // call getint()
  sw x10, 1(sp)    // store return value in i

  sub x5, x5, x5   // x5 = 0 (no need to store in memory)

loop:
  beq x10, x0, end // if (i==0) goto end
  add x5, x5, x10  // x5 += i (still in x10)
  jal getint       // call getint() -> return value in x10
  beq x0,x0,loop   // unconditional jump to start of loop

end:
  lw x6, 3(sp)     // load r in x6
  sw x5, 0(x6)     // r[0] = x5
  lw ra, 2(sp)     // restore return address
  addi sp,sp,4     // remove activation record
  jalr ra          // return
```

Interactive attacker

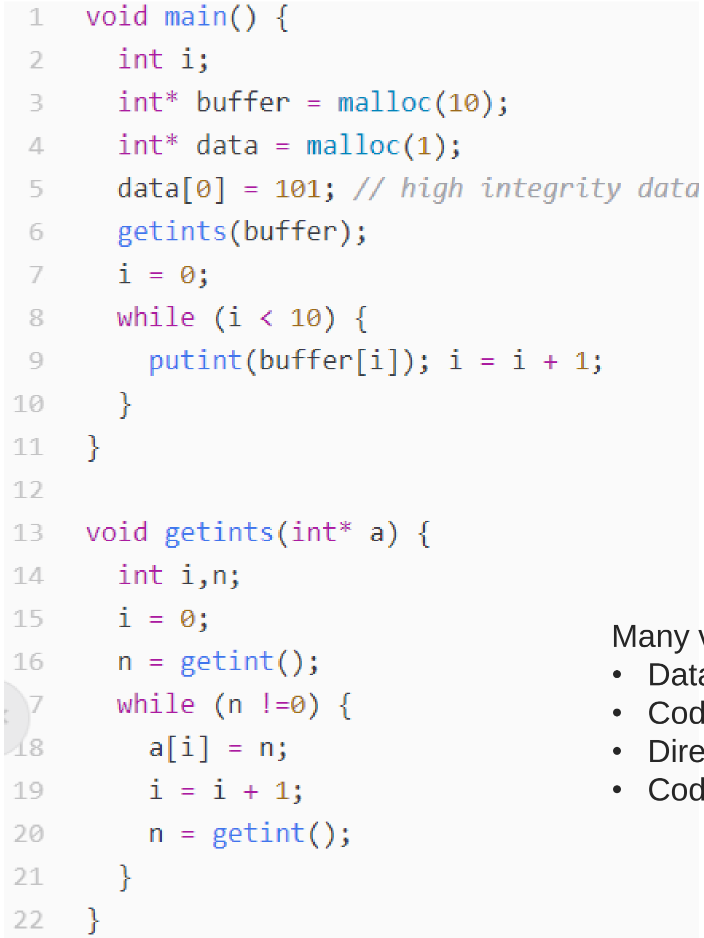
- Models attacks that consist of crafting malicious input and learning from output of the program
  - In our system model: attacker gets to see putint() arguments and gets to choose getint() results



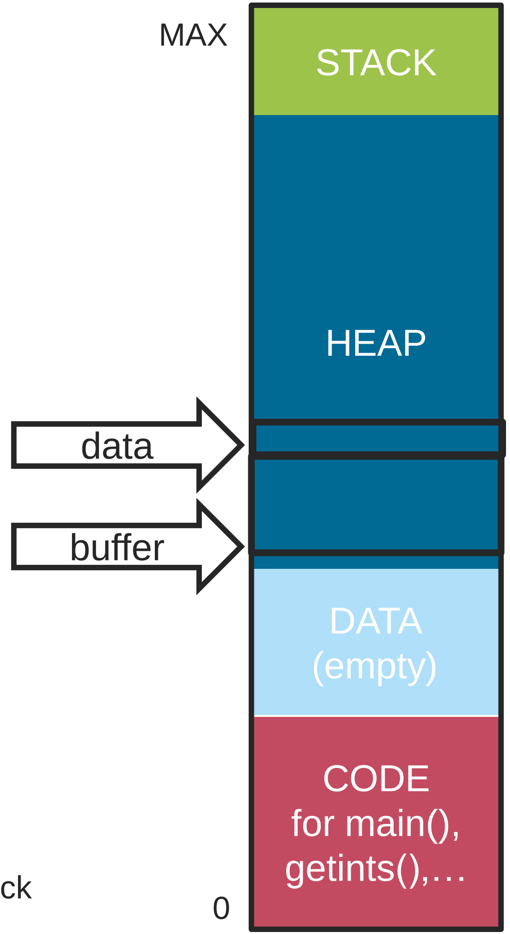
Input / Output

Program

Example attack 1: buffer overwrite



```
void main() {
  int i;
  int* buffer = malloc(10);
  int* data = malloc(1);
  data[0] = 101; // high integrity data
  getints(buffer);
  i = 0;
  while (i < 10) {
    putint(buffer[i]); i = i + 1;
  }
}

void getints(int* a) {
  int i,n;
  i = 0;
  n = getint();
  while (n !=0) {
    a[i] = n;
    i = i + 1;
    n = getint();
  }
}
```

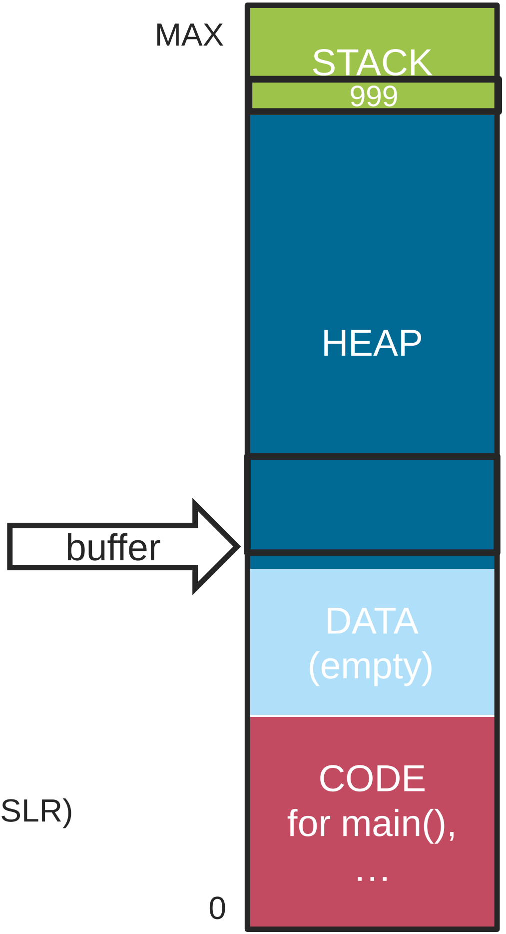


Many variants exist:

- Data-only attack
- Code corruption attack
- Direct code injection attack
- Code reuse (indirect code injection) attack

Example attack 2: buffer over-read

```
void main() {
  int i;
  int SECRET = 999; //  secret data
  int* buffer = malloc(10);
  buffer[0] = 1; buffer[1] = 2; ... // public data in buffer

  i = getint();
  putint(buffer[i]);
}
```



These attacks can leak:

- Application secrets (e.g. keys)
- Secrets that enable other attacks (e.g. ASLR)

Other vulnerabilities

- The example attacks exploited *spatial* memory vulnerabilities
- But other kinds of software bugs can also lead to memory reads or writes that are not allowed:
  - Temporal memory vulnerabilities
  - Uninitialized variables
  - Variadic function misuse
  - …

Overview of the rest of the talk

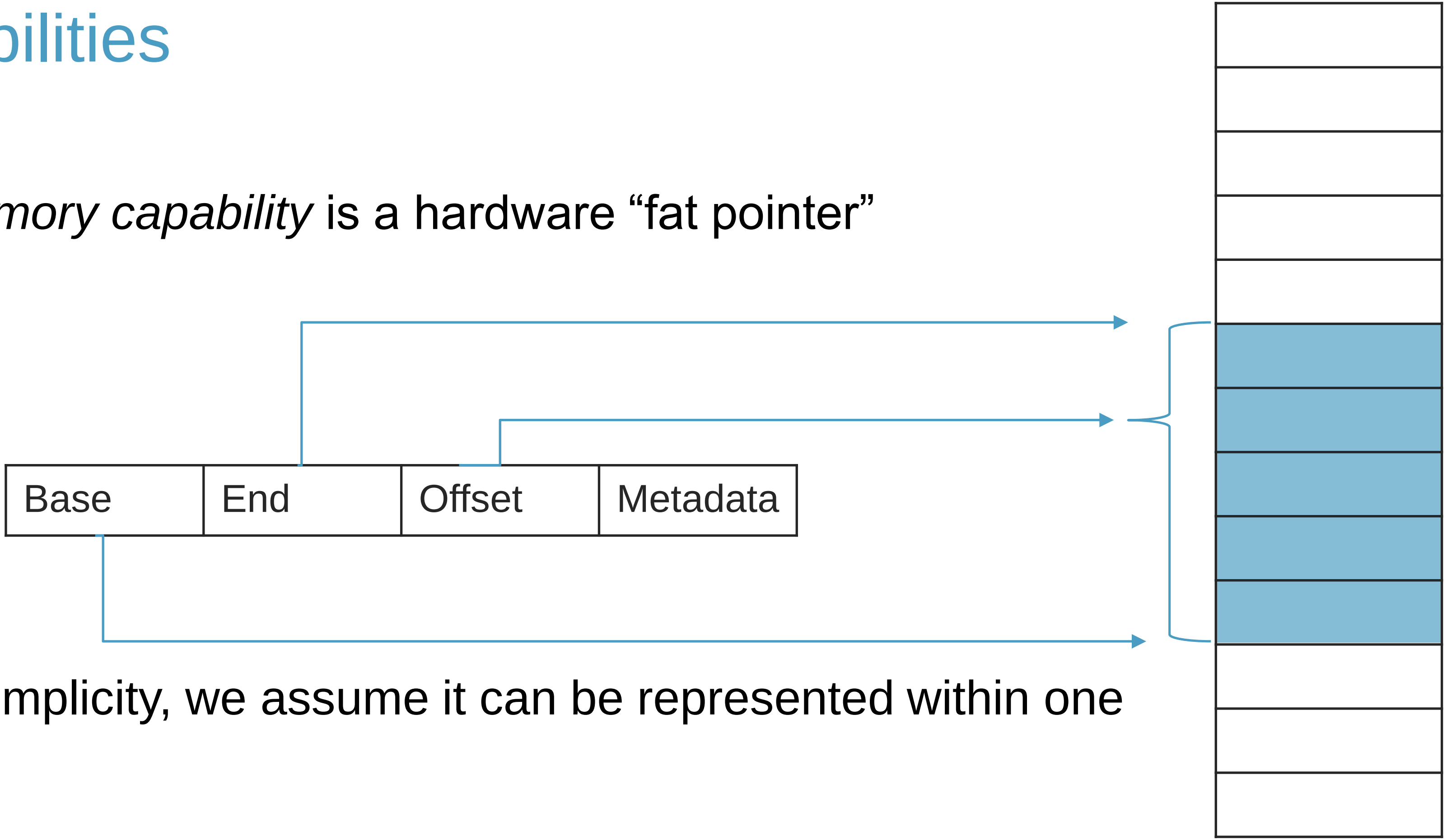
- System model and attacker model
  - Recap of how C-like languages are executed on standard processors
  - Interactive attacker model
- Memory capabilities for run-time security
- Ownership types for compile-time security
- [The next wave of attacks]
- Conclusions

Capability-machines

- Key idea:
  - Pointers (addresses) are NOT integers.
  - Pointers are **capabilities**:
    - They come with a bound on what you can do with that pointer.
    - The entire machine is designed to ensure that capabilities are a **secure** bound on what you can do
- Capabilities are an old security mechanism, studied both at the machine code / OS level, as well as on the PL level
  - We will just discuss the simplest machine-level variant here

Capabilities

- A *memory capability* is a hardware “fat pointer”



- For simplicity, we assume it can be represented within one word

Platform model: CPU extensions/modifications

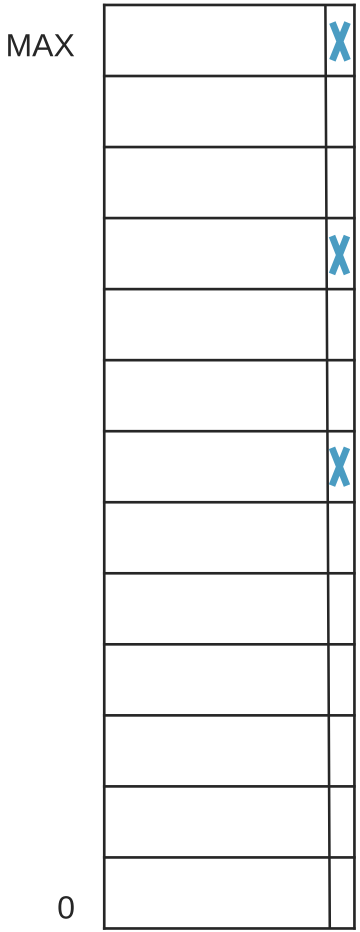
- A CPU with:
  - Standard registers:
    - x0 (=0), x1, x2,…
  - Capability registers:
    - PCC (program counter capability)
    - c0 (=spc), c1 (=rac), c2(=gdc), c3, …
  - Modified and new instructions
    - Memory access must be through a capability
    - Jumps must be to a capability
    - Instructions to compute derived capabilities
      - These must **reduce** the authority of the capability
    - Instructions to inspect capabilities
    - **All instructions check capability constraints**

Example instruction	Semantics
`clw x4,50(c5)`	x4 = M[c5 + 50]
`csw x5,30(c4)`	M[c4+30] = x5
`cjalr 10(c5)`	c1 = PCC+1; PCC = c5 +10
`csetbounds c1,c2,5`	c1.base = c2.offset c1.end = c2.offset + 5 c1.offset = 0 c1.metadata = c2.metadata
`cgetbase x5,c3`	x5 = c3.base

Platform model: memory extensions

- Every memory word has an associated *tag*
  - Set when a capability is stored in that word
  - Cleared whenever a non-capability value is stored

```
csw c5,30(c4)     // store c5 in memory
…
[csw x4,30(c4)]   // OPTIONALLY: store an int
                  // This would clear the tag
…
clw c5,30(c4)     // load c5 from memory again
```



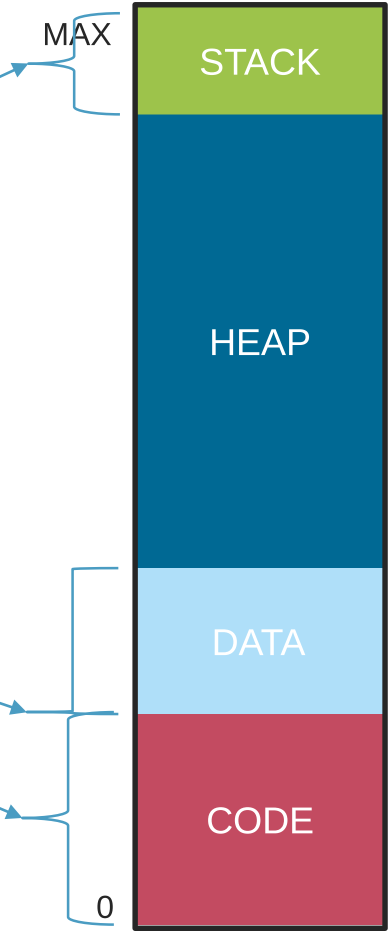
Capabilities are *unforgeable*

- Guarded manipulation
  - Instructions that modify a capability can only reduce their authority
    - Increase base or reduce end
    - Move offset around between base and end
    - Reduce permissions
- Tagged memory
  - Capabilities can be stored in memory and copied around, but are protected by a *tag*

Compilation: Memory layout

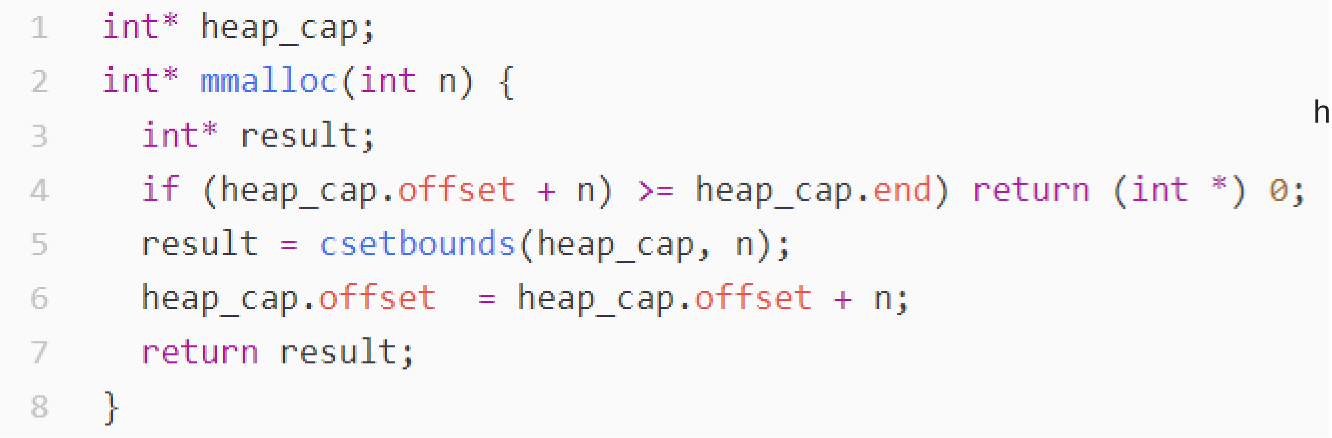
- Dedicated processor registers:
  - spc: stack pointer capability
  - gdc: global data capability
  - pcc: program counter capability
  - malloc() holds a capability to the heap

    and hands out sub-capabilities of appropriate size

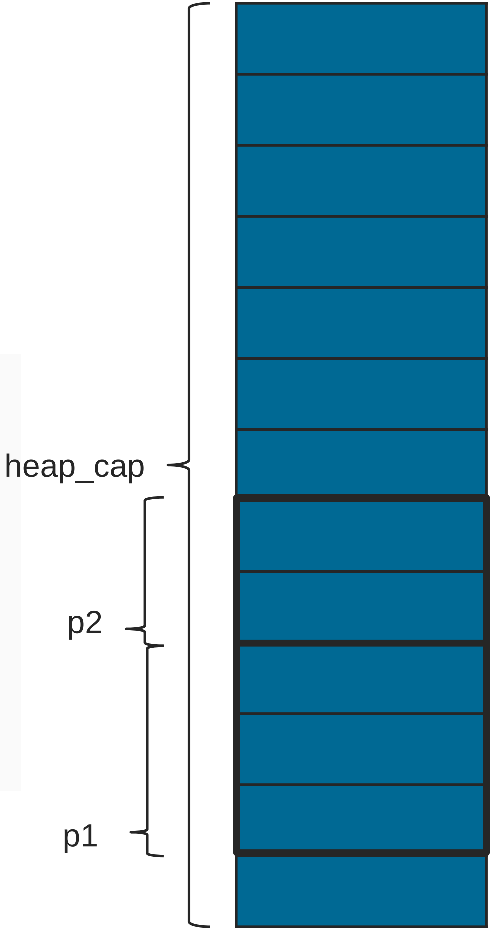


Compilation: malloc() and the heap

- malloc() pseudo-implementation:



```
int* heap_cap;
int* mmalloc(int n) {
  int* result;
  if (heap_cap.offset + n) >= heap_cap.end) return (int *) 0;
  result = csetbounds(heap_cap, n);
  heap_cap.offset  = heap_cap.offset + n;
  return result;
}
```



Example Compilation

```
void sum(int* r) {
  int i,result;

  i = getint();

  result = 0;
  while (i != 0) {
       result += i;
       i = getint();
  }
  r[0] = result;
}
```

```
sum:
  cincoffset csp,csp,-4 // activation record: arg,ra,i,result
  csw c10, 3(csp)       // save argument in activation record
  csw cra, 2(csp)       // save return address in act record

  jal getint            // call getint() (rel jump to PCC)
  csw x10, 1(sp)        // store return value in i

  sub x5, x5, x5        // x5 = 0 (no need to store in memory)

loop:
  beq x10, x0, end      // if (i==o) goto end (rel branch)
  add x5, x5, x10       // x5 += i (still in x10)
  jal getint            // call getint() -> return value in x10
  beq x0,x0,loop        // unconditional jump to start of loop

end:
  clw c6, 3(sp)        // load r in c6 (must be cap register!)
  csw x5, 0(c6)        // r[0] = x5 (store through capability)
  clw cra, 2(sp)       // restore return address
  cincoffset csp,csp,4 // remove activation record
  cjalr cra            // return
```

Memory capabilities for safe compilation of C

- Memory capabilities can represent C pointers, and enforce spatial memory safety at run time
    - Within the interactive attacker model
- This is relatively simple to prove for simplified settings such as the one we considered in this talk
    - CAVEAT: reading uninitialized memory
- For a recent *realistic* prototype, see for instance:
    - David Chisnall, et al. *Beyond the PDP-11: Processor support for a memory-safe C abstract machine*, (ASPLOS 2015)
    - This paper considers many of the challenges involved in bringing this to real-world C
- The main challenge still faced by capability systems is *revocation* (i.e. efficient implementations of free())

Conclusions

- Memory capabilities are a useful hardware primitive to build secure C compilers
- Hardware-supported capabilities are becoming more mainstream
  - The CHERI project: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/
  - Arm is working with the CHERI team to bring these ideas into the Arm architecture:
    - https://community.arm.com/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity
- And we are not even using the full power of capabilities yet
  - In particular, a capability processor satisfies a *monotonicity* property:
    - The set of memory addresses that a program has access to (directly or indirectly), can only shrink over time.
  - This makes it possible to *share* a memory address space between distrusting program components providing strong compartmentalization guarantees

Overview of the rest of the talk

- System model and attacker model
  - Recap of how C-like languages are executed on standard processors
  - Interactive attacker model
- Memory capabilities for run-time security
- Ownership types for compile-time security
- [The next wave of attacks]
- Conclusions

Memory safety

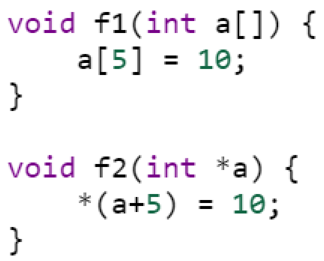
- An important reason why C programs have exploitable security vulnerabilities is because of unsafe memory accesses
  - The program contains a bug (e.g. missing bounds check) such that the compiled program performs a memory access (read or write) that an attacker can control

Essentially, only 4 ways things can go wrong

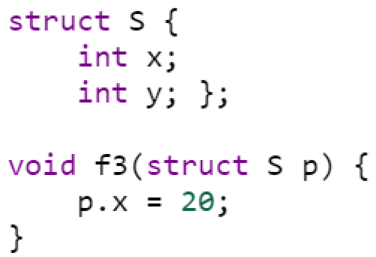
- **Spatial memory safety errors**: a blob of allocated memory is accessed out of bounds
- **Temporal memory safety errors**: a blob of memory is accessed after it has been deallocated
- **Pointer forging**: creating an invalid pointer value
    - By invalid casts
    - By use of uninitialized memory
- **Unsafe primitive API functions**:
    - Like C’s printf() function

Spatial memory safety

- Examples: indexing an array, indexing a struct, pointer arithmetic



```
void f1(int a[]) {
    a[5] = 10;
}

void f2(int *a) {
    *(a+5) = 10;
}
```



```
struct S {
    int x;
    int y; };

void f3(struct S p) {
    p.x = 20;
}
```

- How could the compiler protect against spatial memory safety errors?

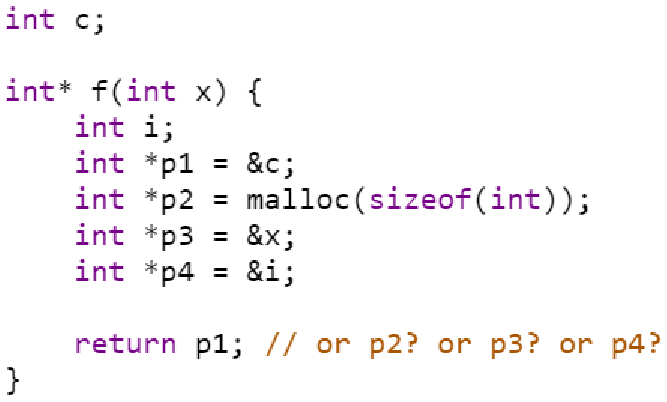
Enforcing spatial memory safety

- Through type checking for structs and arrays with statically known bounds
  - E.g. Java type system will make sure that you can not access a non-existing field of an object
- Through run-time bounds checking otherwise
  - E.g. Java throws ArrayIndexOutOfBoundsException
  - E.g. “Fat” pointers in C or C++

Temporal memory safety

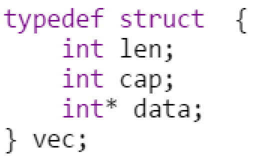
- How long are pointers valid?

  This depends on how the pointer is created.

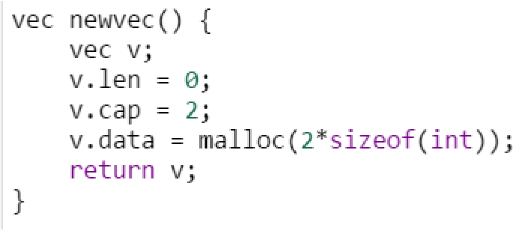


```
int c;

int* f(int x) {
    int i;
    int *p1 = &c;
    int *p2 = malloc(sizeof(int));
    int *p3 = &x;
    int *p4 = &i;

    return p1; // or p2? or p3? or p4?
}
```

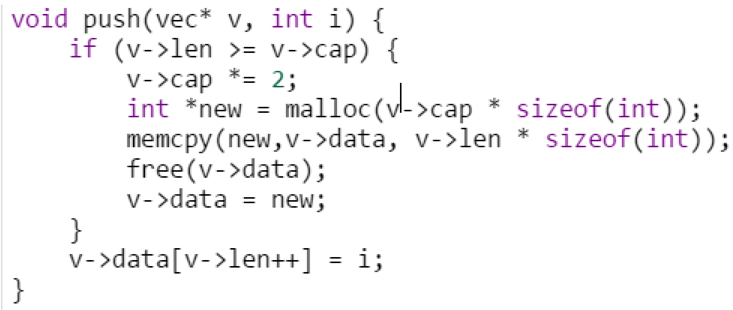
A simple example



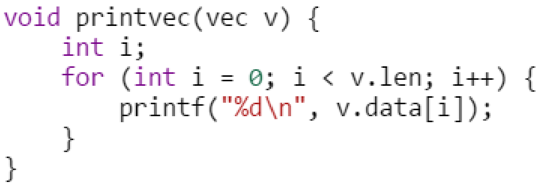
```
typedef struct  {
    int len;
    int cap;
    int* data;
} vec;
```



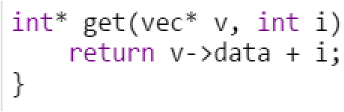
```
vec newvec() {
    vec v;
    v.len = 0;
    v.cap = 2;
    v.data = malloc(2*sizeof(int));
    return v;
}
```



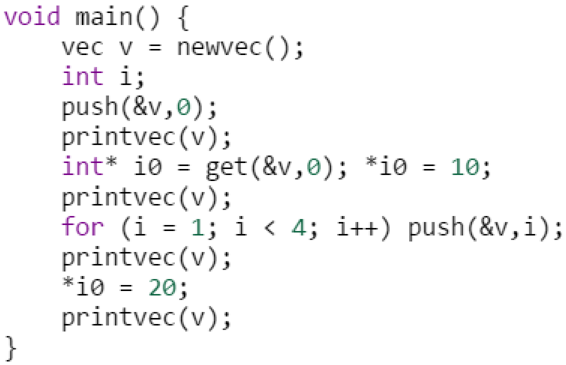
```
void push(vec* v, int i) {
    if (v->len >= v->cap) {
        v->cap *= 2;
        int *new = malloc(v->cap * sizeof(int));
        memcpy(new,v->data, v->len * sizeof(int));
        free(v->data);
        v->data = new;
    }
    v->data[v->len++] = i;
}
```



```
void printvec(vec v) {
    int i;
    for (int i = 0; i < v.len; i++) {
        printf("%d\n", v.data[i]);
    }
}
```



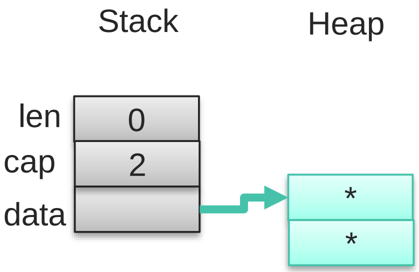
```
int* get(vec* v, int i) {
    return v->data + i;
}
```



```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

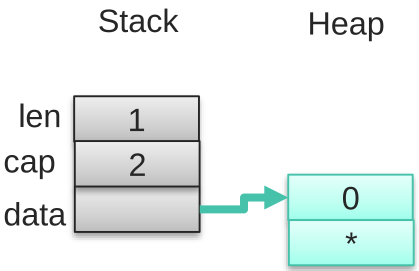
A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



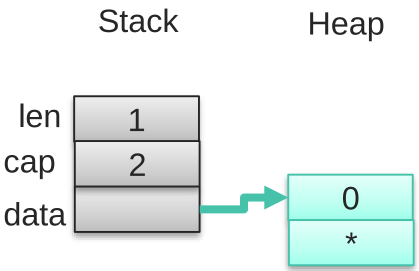
A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



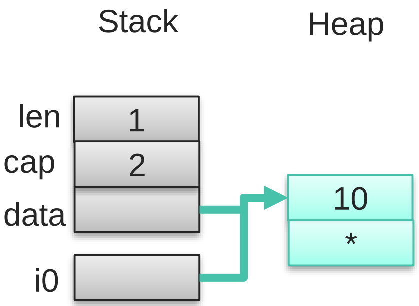
Output:

```
0
```

A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

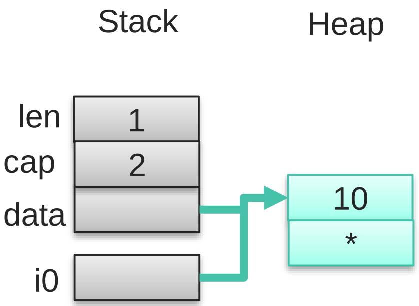
Output:
0



A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

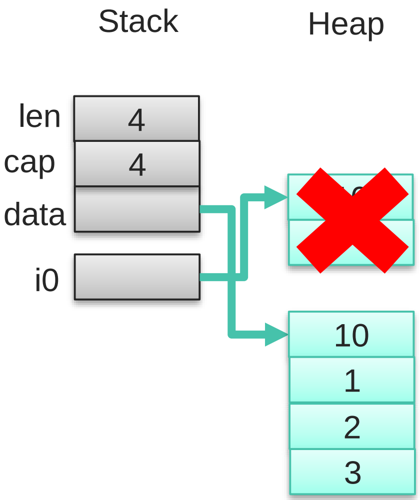
Output:
0
10



A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:
0
10

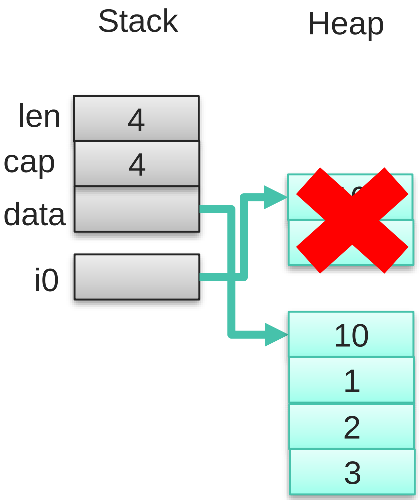


A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:

```
0
10
10
1
2
3
```



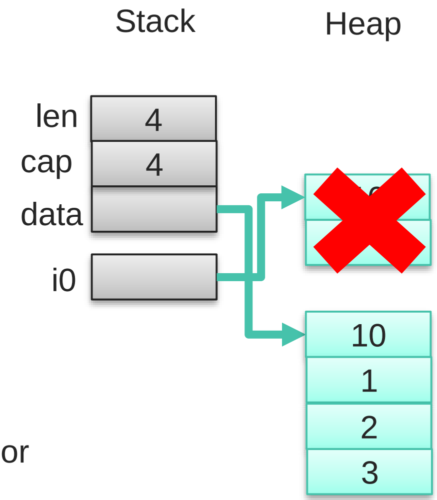
A simple example

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0); *i0 = 10;
    printvec(v);
    for (i = 1; i < 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

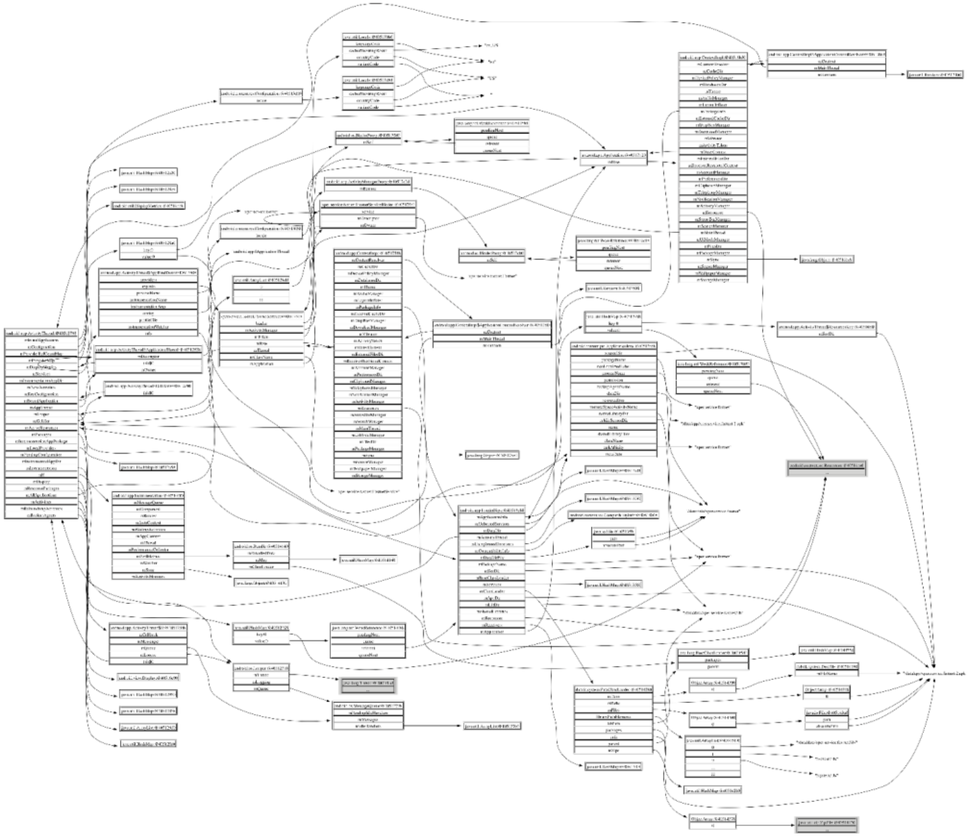
Output:

```
0
10
10
1
2
3
```

Temporal memory safety error



Real heap looks more complicated…



Enforcing temporal memory safety

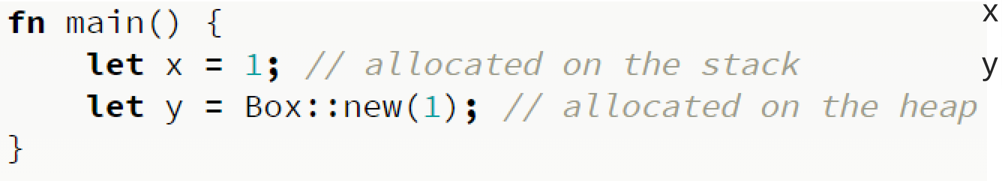
- Allocate everything on the heap, and do **garbage collection**:
  - Programmer can not do explicit deallocation
    - I.e. no free()
  - At regular intervals, the program will be halted and the run-time system will clean up unused memory
    - Basic idea: check what memory is reachable from the current program state, and deallocate all the rest
    - Many different strategies to implement this with different pros and cons
- Important disadvantages for systems programming:
  - Less precise control over memory
  - Unpredictable timing

Enforcing temporal memory safety

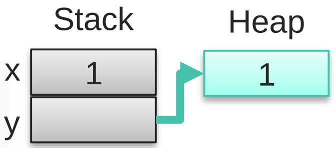
- New approach: **ownership types** and **borrowing**
- Basic idea:
  - There is at all times a unique **owning** pointer to each allocated blob of memory
  - Memory is deallocated when the owning pointer disappears
    - Because it goes out of scope
    - Or because it is overwritten
    - Or because it was part of a data structure that is being deallocated
- We discuss the implementation of this idea in **Rust**

Memory management in Rust

- Programmer controls:
  - At what time memory is allocated
  - And where it is allocated (stack / heap)
- Deallocated when owner goes out of scope

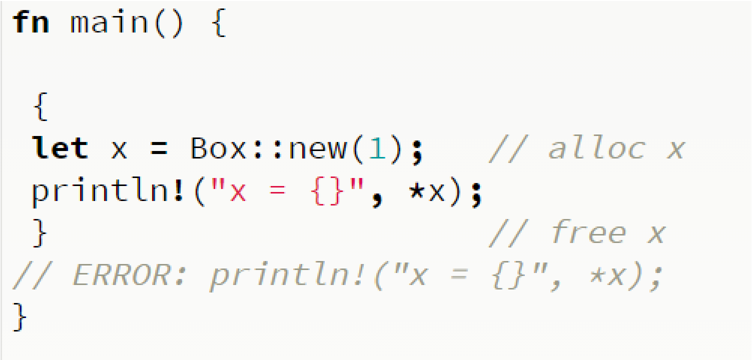


```
fn main() {
    let x = 1; // allocated on the stack
    let y = Box::new(1); // allocated on the heap
}
```



No use after free is possible

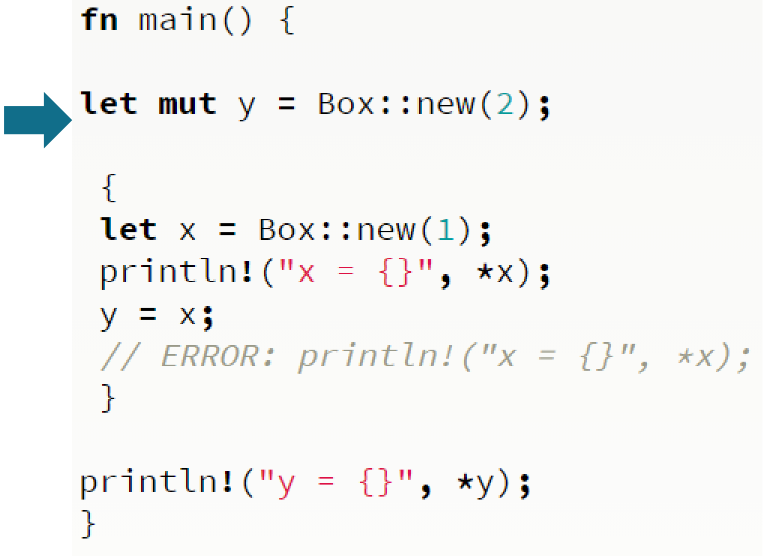
- There was only a single pointer, and it has gone out of scope



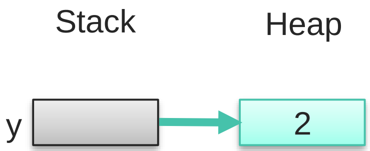
```
fn main() {

 {
 let x = Box::new(1);    // alloc x
 println!("x = {}", *x);
 }                       // free x
// ERROR: println!("x = {}", *x);
}
```

Move semantics

- Pointers are not copied but **moved**

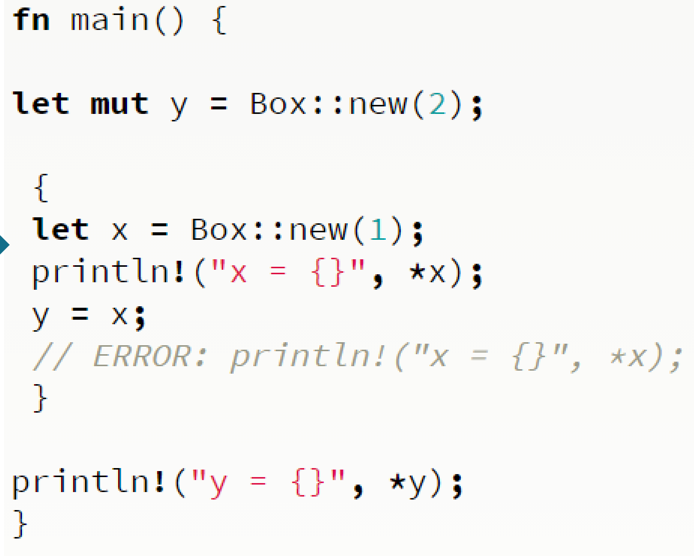


```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```

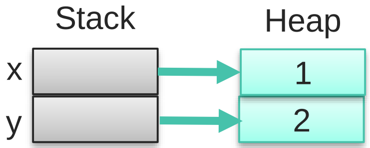


Move semantics

- Pointers are not copied but **moved**

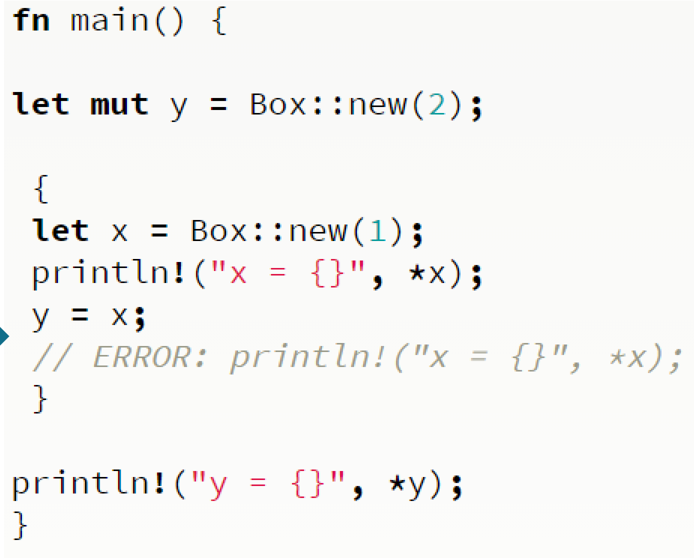


```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```

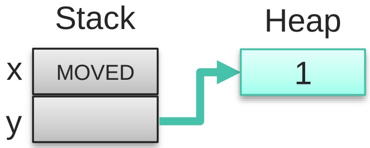


Move semantics

- Pointers are not copied but **moved**

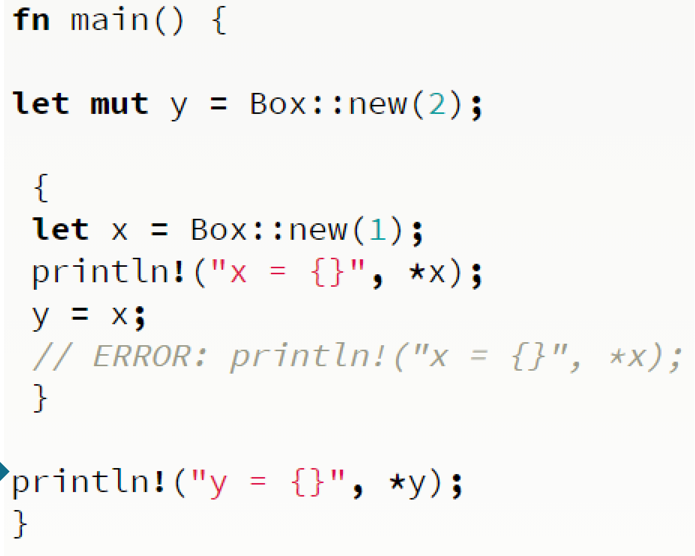


```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```

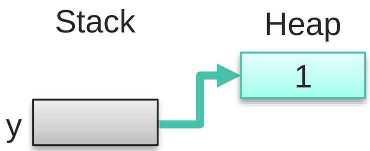


Move semantics

- Pointers are not copied but **moved**
  - Hence: there is always a **unique** owning pointer

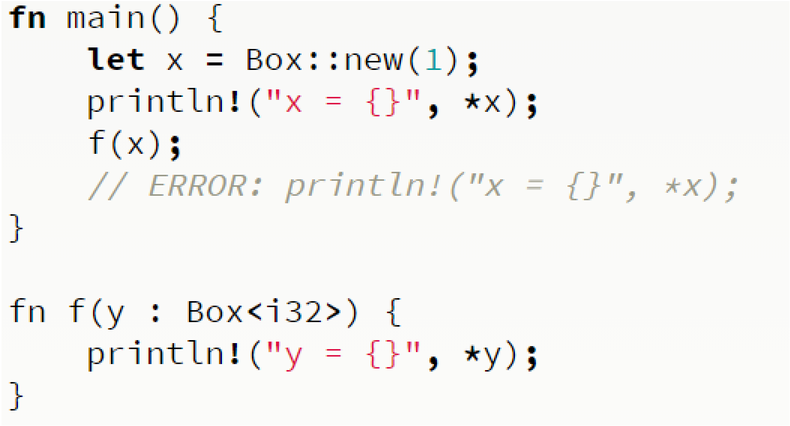


```
fn main() {

let mut y = Box::new(2);

 {
 let x = Box::new(1);
 println!("x = {}", *x);
 y = x;
 // ERROR: println!("x = {}", *x);
 }

println!("y = {}", *y);
}
```

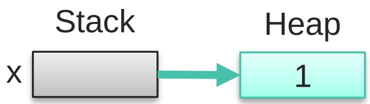


Pointers move into functions too

- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?

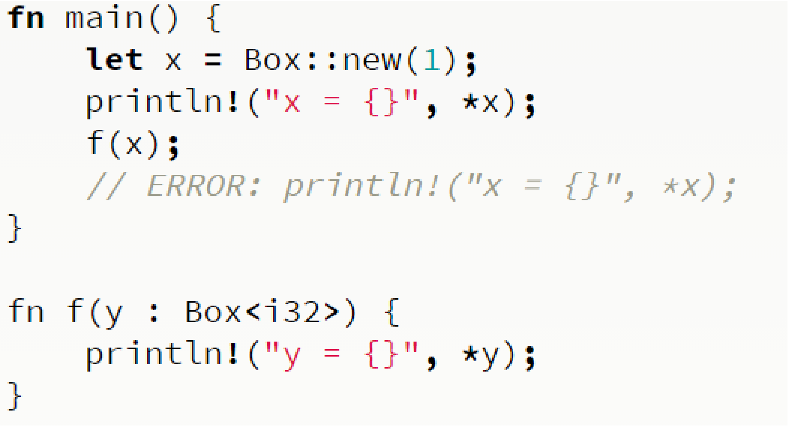


```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```

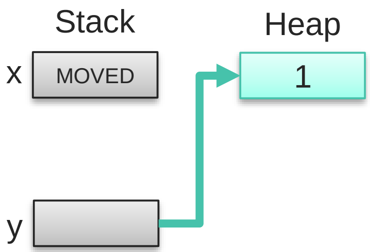


Pointers move into functions too

- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?

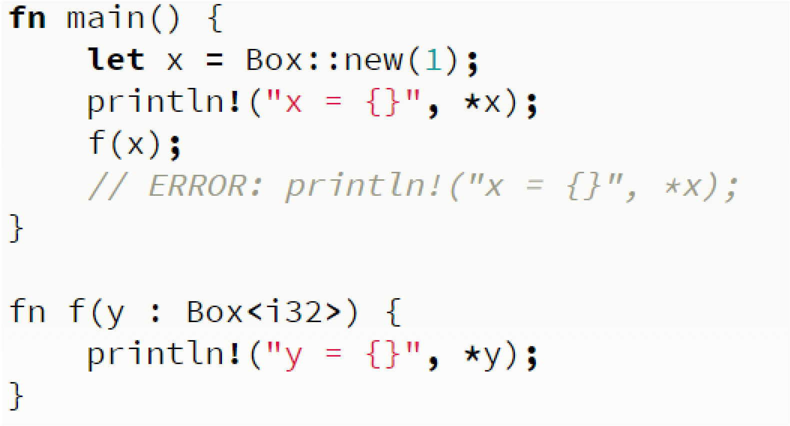


```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```



Pointers move into functions too

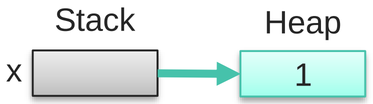
- Ownership moves from argument to formal parameter
- So when is the allocated memory freed in the program below?



```
fn main() {
    let x = Box::new(1);
    println!("x = {}", *x);
    f(x);
    // ERROR: println!("x = {}", *x);
}

fn f(y : Box<i32>) {
    println!("y = {}", *y);
}
```

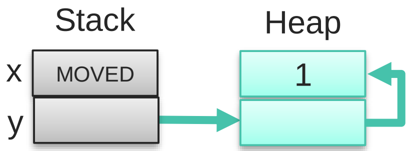
Pointers can also move into Boxes and structs

```
fn main() {
  let mut x = Box::new(1);
  let mut y = Box::new(x);
  let mut z = Box::new(y);
  println!("Val:{}",***z);
}
```



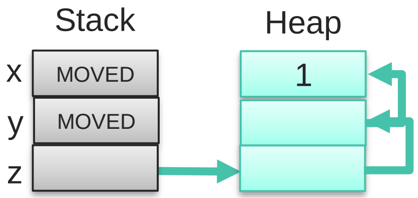
Pointers can also move into Boxes and structs

```
fn main() {
  let mut x = Box::new(1);
  let mut y = Box::new(x);
  let mut z = Box::new(y);
  println!("Val:{}",***z);
}
```

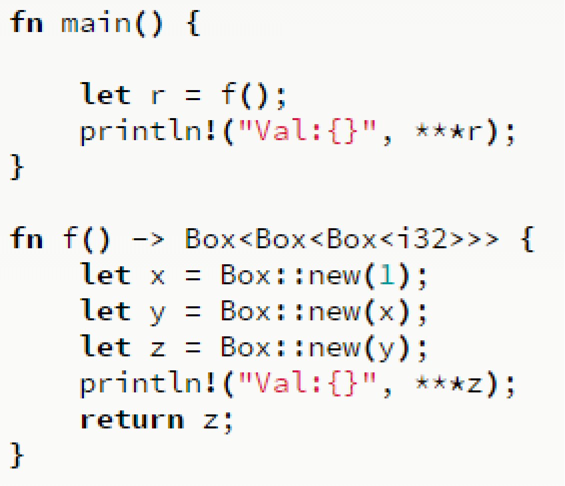


Pointers can also move into Boxes and structs

```
fn main() {
  let mut x = Box::new(1);
  let mut y = Box::new(x);
  let mut z = Box::new(y);
  println!("Val:{}",***z);
}
```

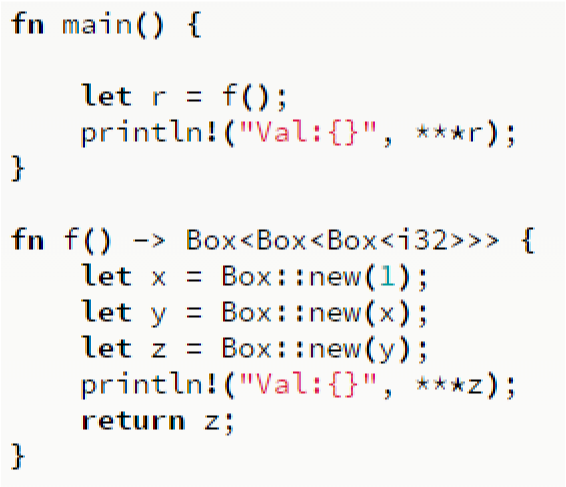


Moving into a box can extend life



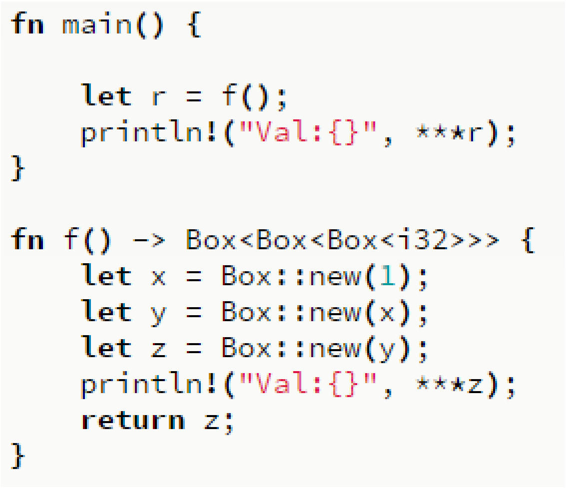
```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```

Moving into a box can extend life

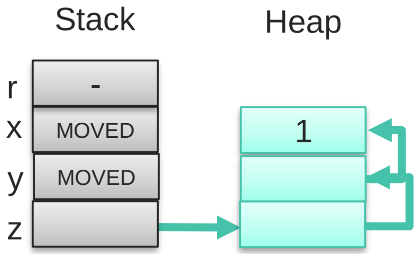


```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```

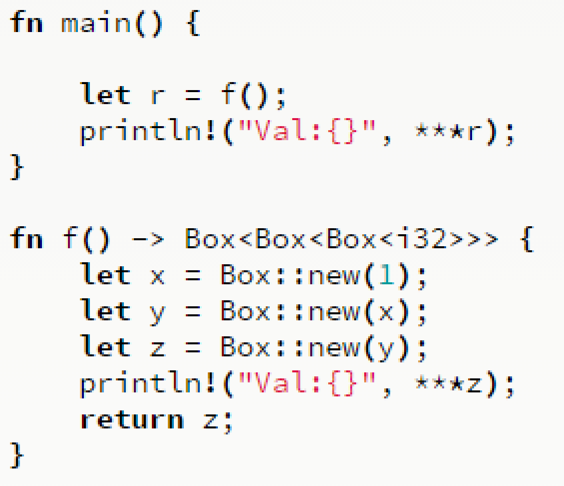
Moving into a box can extend life



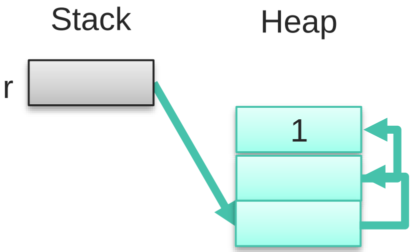
```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```



Moving into a box can extend life



```
fn main() {

    let r = f();
    println!("Val:{}", ***r);
}

fn f() -> Box<Box<Box<i32>>> {
    let x = Box::new(1);
    let y = Box::new(x);
    let z = Box::new(y);
    println!("Val:{}", ***z);
    return z;
}
```

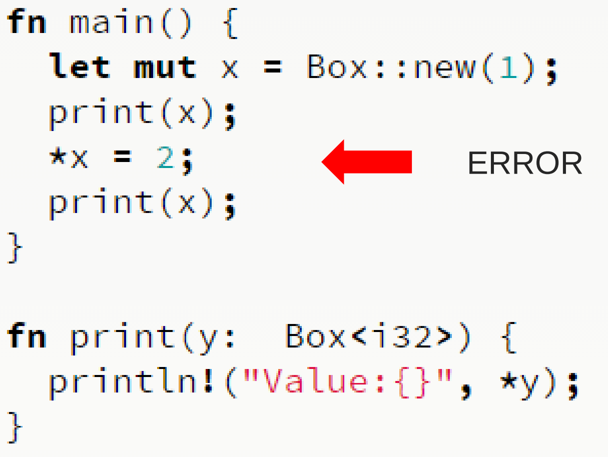


Enforcing unique ownership simplifies the heap

- The heap is a forest (set of trees), with allocated blobs of memory as nodes, and owning references as arrows.
- Roots of the trees are on the stack:
  - local variables of Box type
- If a local variable goes out of scope, that tree gets deallocated
  - We know that there are no other owners, because of uniqueness of ownership
- Uniqueness of ownership is maintained with the move semantics of pointers

Borrowing

- Move semantics is sometimes too limiting / annoying

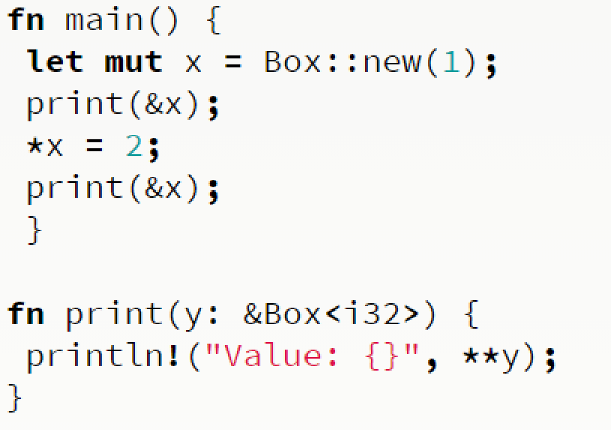


```
fn main() {
  let mut x = Box::new(1);
  print(x);
  *x = 2;
  print(x);
}

fn print(y:  Box<i32>) {
  println!("Value:{}", *y);
}
```

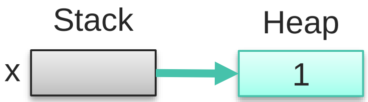
ERROR

- Rust supports “borrowing” of references to address this

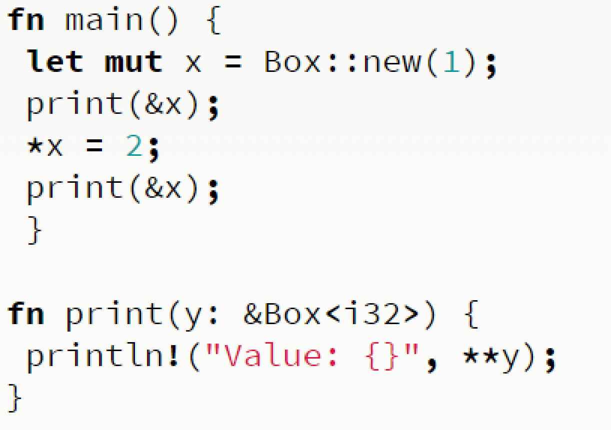
Borrowing



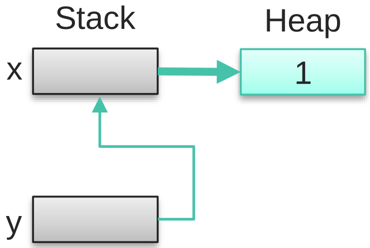
```
fn main() {
 let mut x = Box::new(1);
 print(&x);
 *x = 2;
 print(&x);
 }

fn print(y: &Box<i32>) {
 println!("Value: {}", **y);
}
```



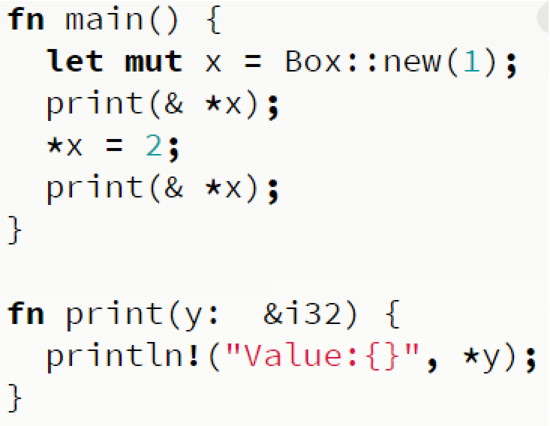
Borrowing



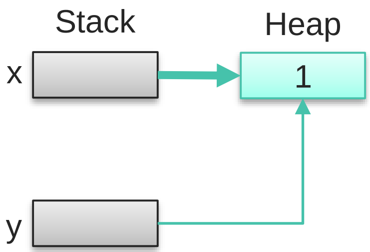
```
fn main() {
 let mut x = Box::new(1);
 print(&x);
 *x = 2;
 print(&x);
 }

fn print(y: &Box<i32>) {
 println!("Value: {}", **y);
}
```



Borrowing

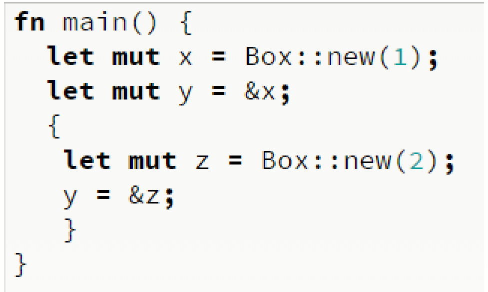


```
fn main() {
  let mut x = Box::new(1);
  print(& *x);
  *x = 2;
  print(& *x);
}

fn print(y:  &i32) {
  println!("Value:{}", *y);
}
```

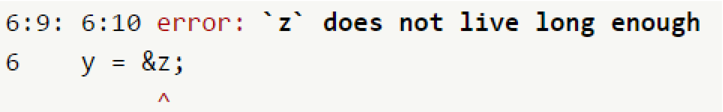


Borrowing rules

- To avoid introducing temporal safety errors, borrowing and ownership follow some rules:
  - The *lifetime* of a borrow should always be included in the lifetime of the owner from which it is borrowed
    - Otherwise, if the owner dies, the borrowed reference would be dangling

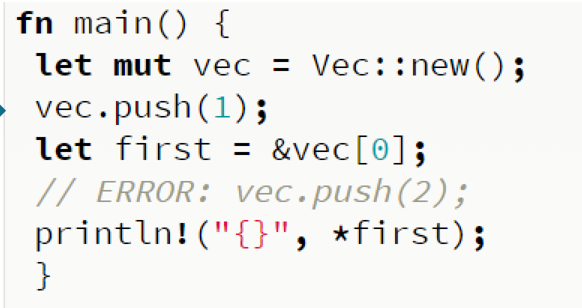


```
fn main() {
  let mut x = Box::new(1);
  let mut y = &x;
  {
   let mut z = Box::new(2);
   y = &z;
   }
}
```

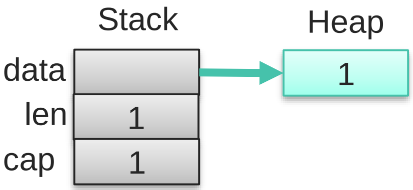


```
6:9: 6:10 error: `z` does not live long enough
6    y = &z;
          ^
```

Borrowing should also forbid mutation

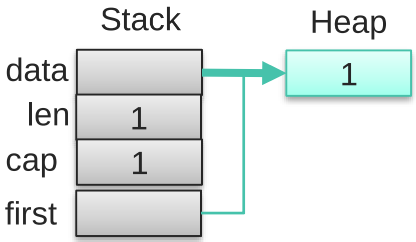


```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```

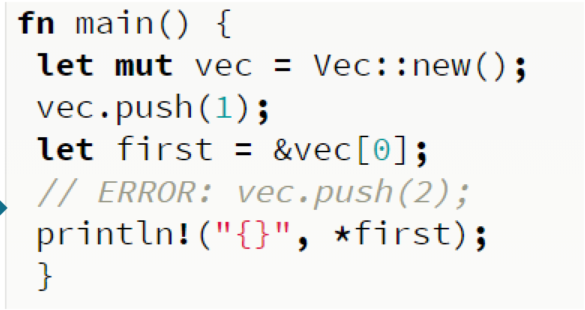


Borrowing should also forbid mutation

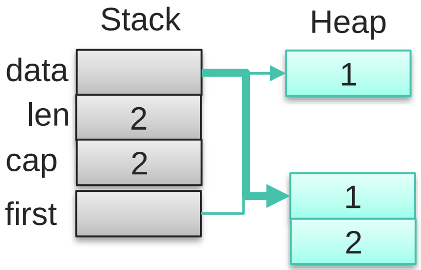
```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



Borrowing should also forbid mutation

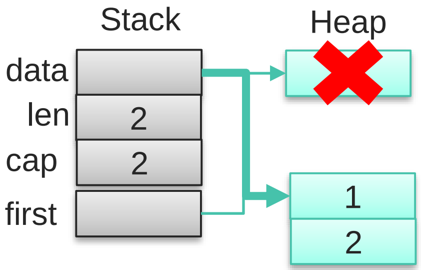


```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



Borrowing should also forbid mutation

```
fn main() {
 let mut vec = Vec::new();
 vec.push(1);
 let first = &vec[0];
 // ERROR: vec.push(2);
 println!("{}", *first);
 }
```



Borrowing rules

- Rust supports borrowing:
  - Either: an arbitrary number of immutable references
  - Or: a single mutable reference
- To ensure safety, Rust ensures:
  - Modification through the owner is disallowed while borrows are outstanding
  - Lifetimes of borrowed references are always strictly included in the lifetime of the owner

Summary: Ownership and borrowing

- Together these concepts:
  - Can guarantee temporal memory safety statically
    - By ruling out simultaneous aliasing + mutation
  - Allow relatively flexible pointer manipulating programs
- Many advantages:
  - No need for a run-time (no garbage collection)
  - Also helps in avoiding data races (concurrency errors)
- Some disadvantages:
  - Non-trivial to use
  - Not as flexible as C

The Rust programming language

- Is one of the fastest growing languages at the moment
- Since Firefox 48 (August 2016), there is Rust code in Firefox
- The language has many other interesting features that we did not discuss
  - Pattern matching
  - Traits
  - Generics
  - …
- See:
  - https://www.rust-lang.org/

Comparison

- Java/C#/JavaScript/…
  - Runtime = virtual machine + JIT compiler + GC + …
  - Garbage collection can induce substantial latency
    - “Stop-the-world”
- Go
  - Runtime = GC
  - Low-latency garbage collection
    - Focus on GC algorithms that can keep the program running
- Rust
  - “Runtime” = just a set of libraries

Overview of the rest of the talk

- System model and attacker model
  - Recap of how C-like languages are executed on standard processors
  - Interactive attacker model
- Memory capabilities for run-time security
- Ownership types for compile-time security
- [The next wave of attacks]
- Conclusions

Introduction

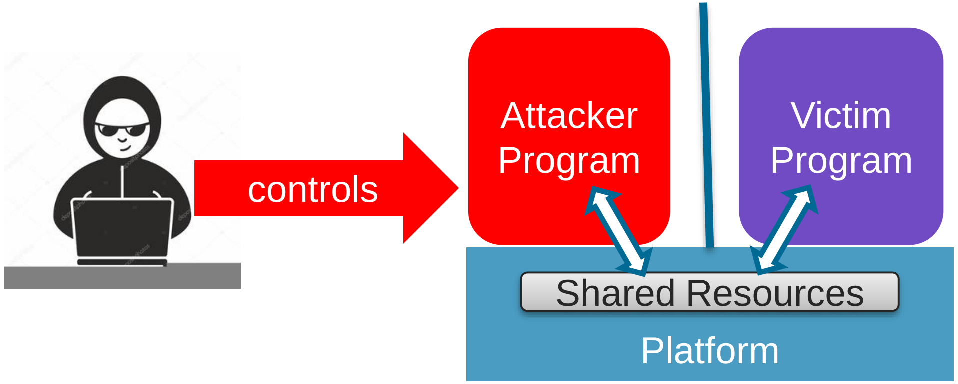
- In 2018, micro-architectural attacks have come of age:
  - Meltdown breaks user/kernel isolation
  - Spectre breaks several isolation boundaries that software security fundamentally relies on
  - Foreshadow breaks SGX enclave isolation
- Hardware and system software vendors are scrambling to address these attacks

References:

Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*, IEEE S&P 2019
Moritz Lipp et al. *Meltdown: Reading Kernel Memory from User Space*, USENIX Security Symposium 2018
Jo Van Bulck et al. *Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution*, USENIX Security Symposium 2018

Attacker model: Shared platform attacker

- The attacker can run code on the same platform where victim code is running.
- The objective of the attacker is to learn more about the victim than what one can learn through intended communication interfaces.



controls

Attacker Program

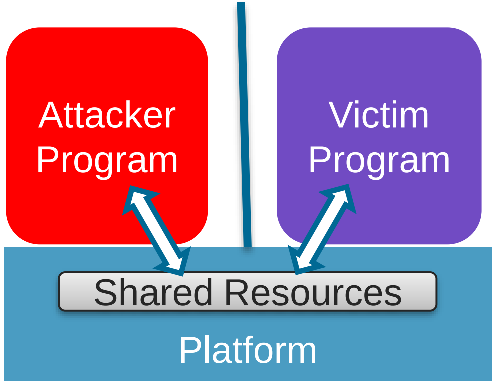
Victim Program

Shared Resources

Platform

Micro-architectural attacks

- The attacker learns information by manipulating and observing the victim program’s use of shared platform resources such as the cache, the branch predictor, …



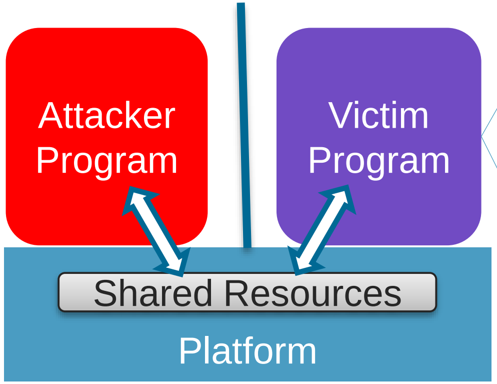
Attacker Program

Victim Program

Shared Resources

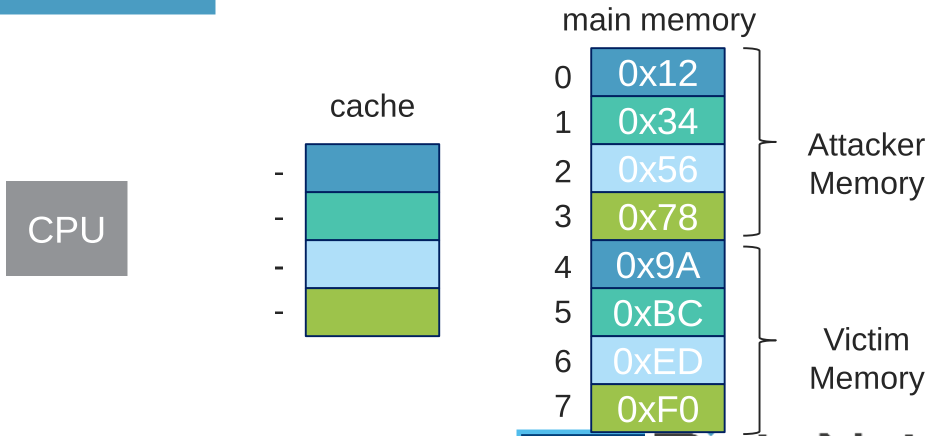
Platform

Side-channels: a simple example of a cache-attack



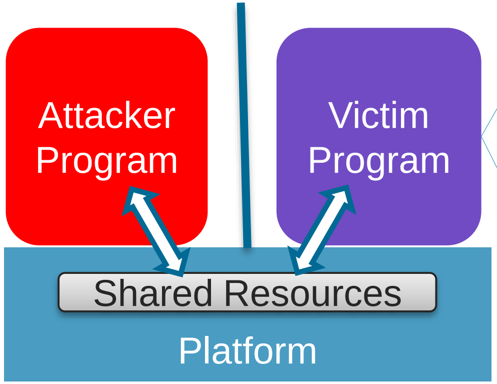
```
…
if secret {
   load address 4
}
else {
   load address 5
}
…
```

- The shared resources between attacker and victim program include a direct-mapped cache



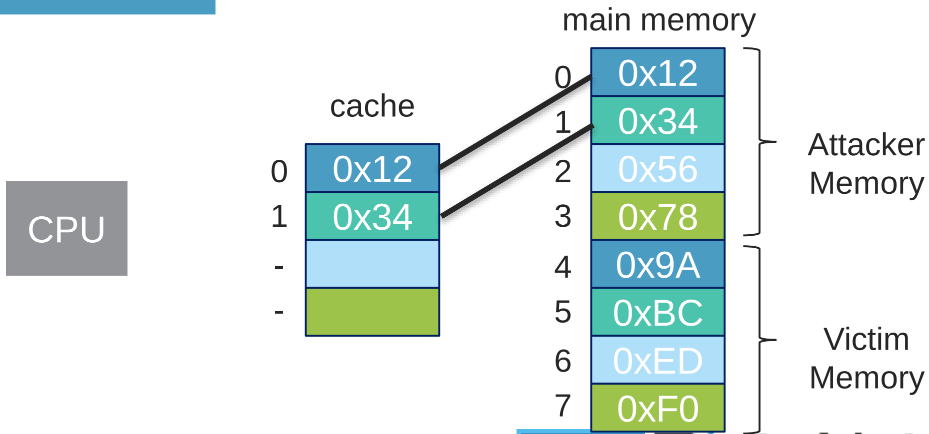
	main memory
0	0x12
1	0x34
2	0x56
3	0x78
4	0x9A
5	0xBC
6	0xED
7	0xF0

Side-channels: a simple example of a cache-attack

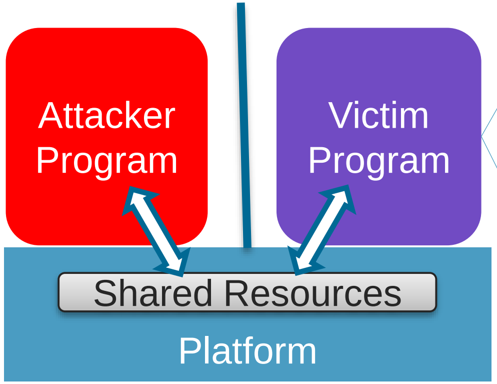


```
…
if secret {
   load address 4
}
else {
   load address 5
}
…
```

- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines

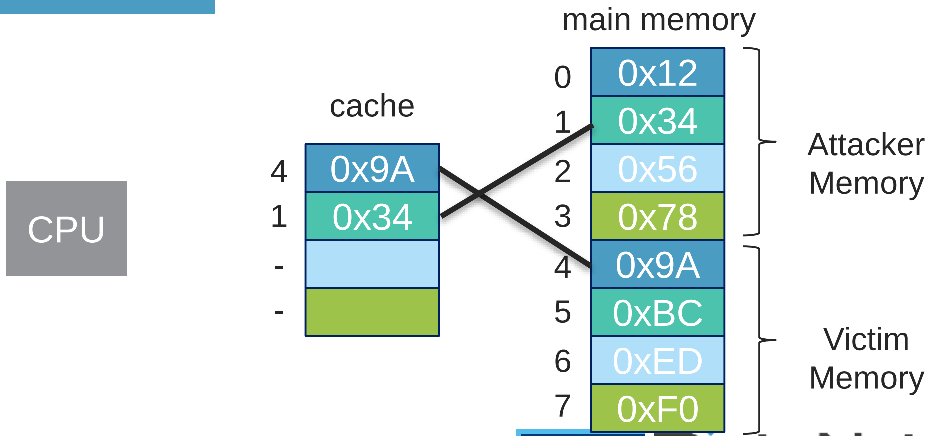


Side-channels: a simple example of a cache-attack



```
…
if secret {
   load address 4
}
else {
   load address 5
}
…
```

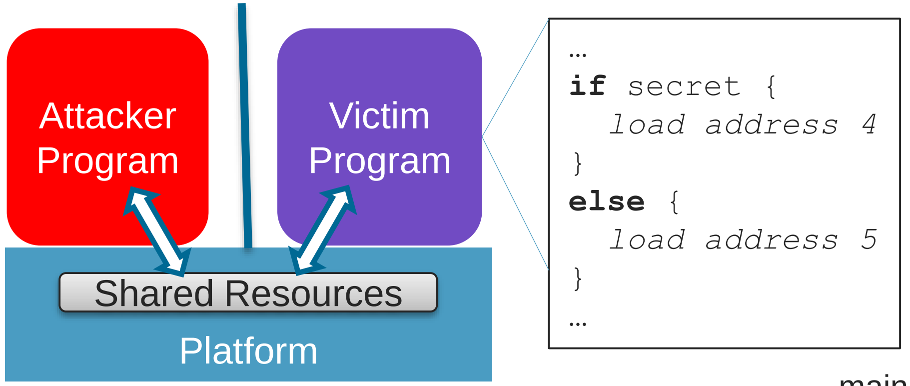
- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines
  - Next the victim program runs and performs **secret-dependent** memory accesses



	cache
4	0x9A
1	0x34
-	
-	

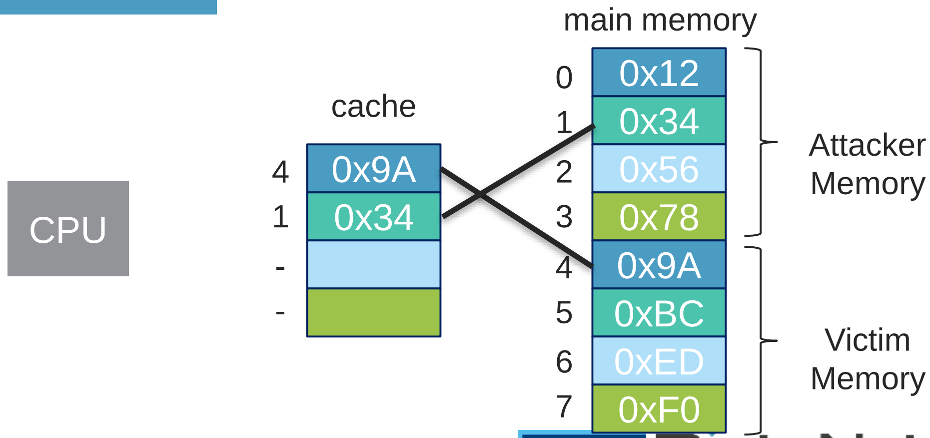
	main memory
0	0x12
1	0x34
2	0x56
3	0x78
4	0x9A
5	0xBC
6	0xED
7	0xF0

Side-channels: a simple example of a cache-attack



```
…
if secret {
  load address 4
}
else {
  load address 5
}
…
```

- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines
  - Next the victim program runs and performs **secret-dependent** memory accesses
  - Finally, attacker measures duration of an access to address 0



cache	
4	0x9A
1	0x34
-	
-	

main memory	
0	0x12
1	0x34
2	0x56
3	0x78
4	0x9A
5	0xBC
6	0xED
7	0xF0

Cache attacks

- Cache-based side-channel attacks have been understood for quite a while
- Countermeasures exist:
  - At the hardware level, e.g. cache partitioning
  - At the software level, e.g. the crypto constant time model

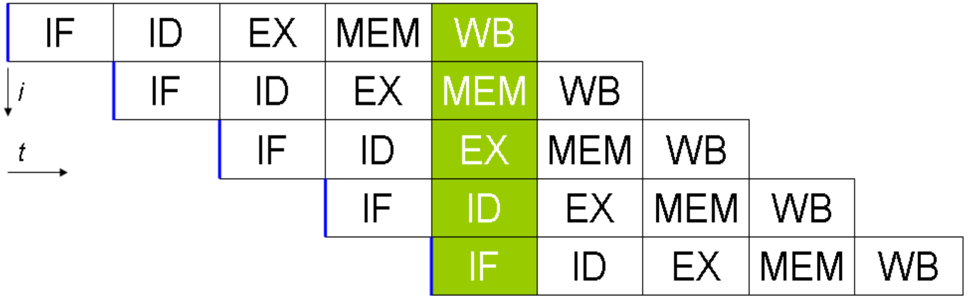
Qian Ge, Yuval Yarom, David Cock, Gernot Heiser: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptographic Engineering (2018)

Speculative execution attacks

- Speculative execution attacks amplify the impact of existing side-channels **by giving the attacker control over the sending side of the channel** too
- The key observations are:
  - Processors are pipelined and sometimes execute instructions *speculatively*
    - No architectural effects are visible until instruction is committed
  - Speculatively executed instructions *also impact the micro-architectural state*
  - The attacker *can influence what instructions get executed speculatively*

Speculative execution

- All major processors support speculative execution
  - Processor implementations are pipelined
  - To keep the hardware busy, instructions are executed *out-of-order* and *speculatively*
  - No visible *architectural* effects of speculatively executed instructions – but there are persistent micro-architectural effects



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

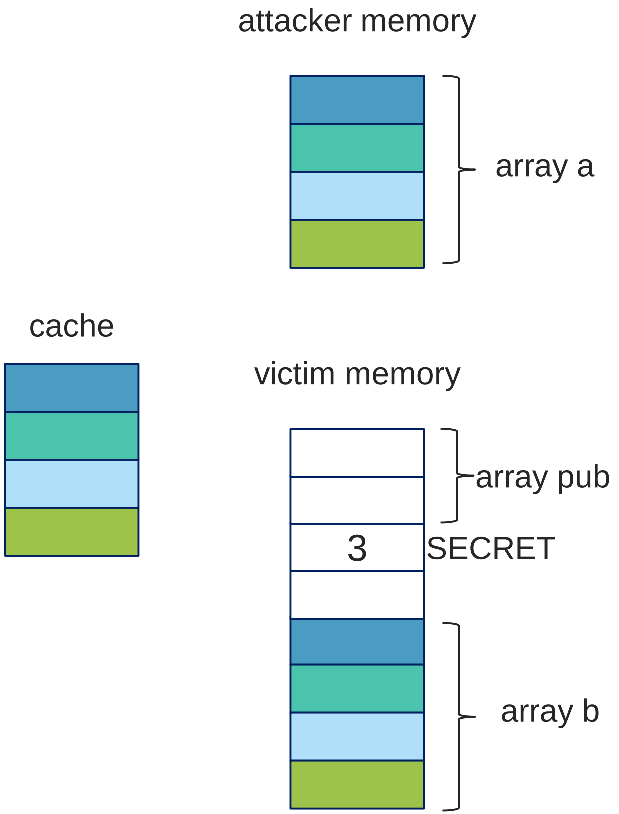
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



A simple example of a speculative execution attack

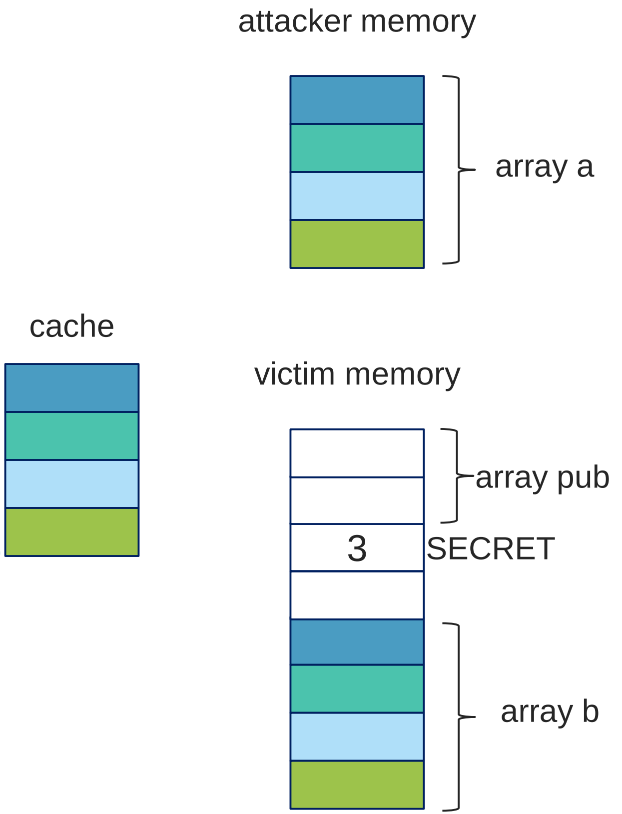
attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```

Branch predictor learns that usually then branch is taken



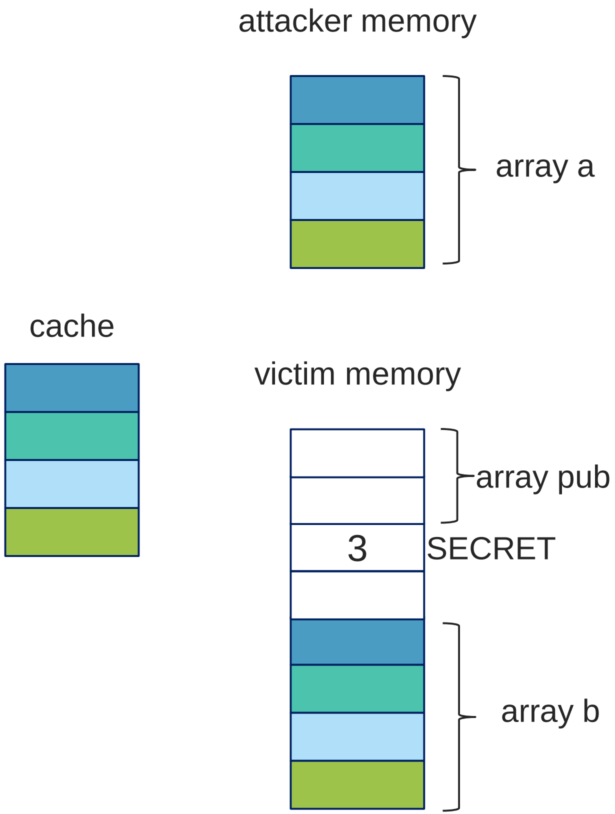
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



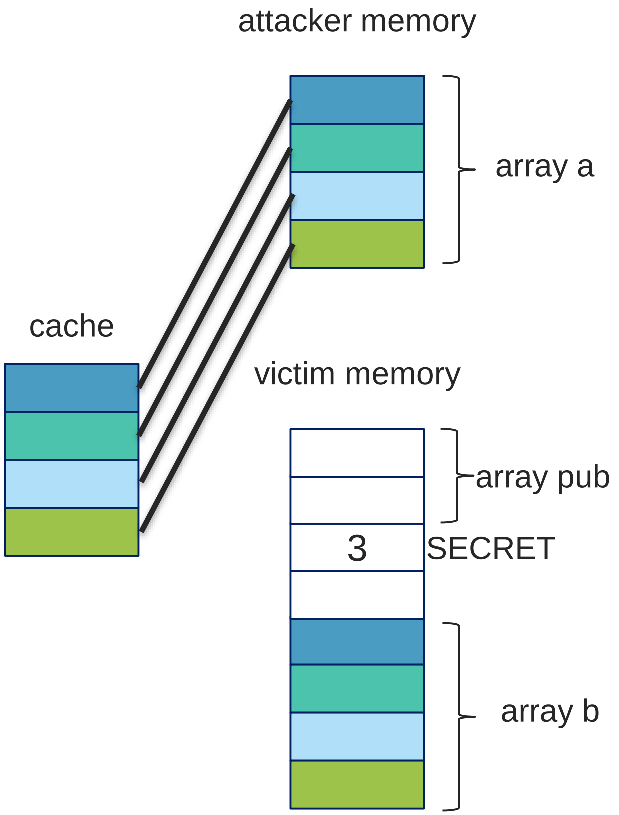
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



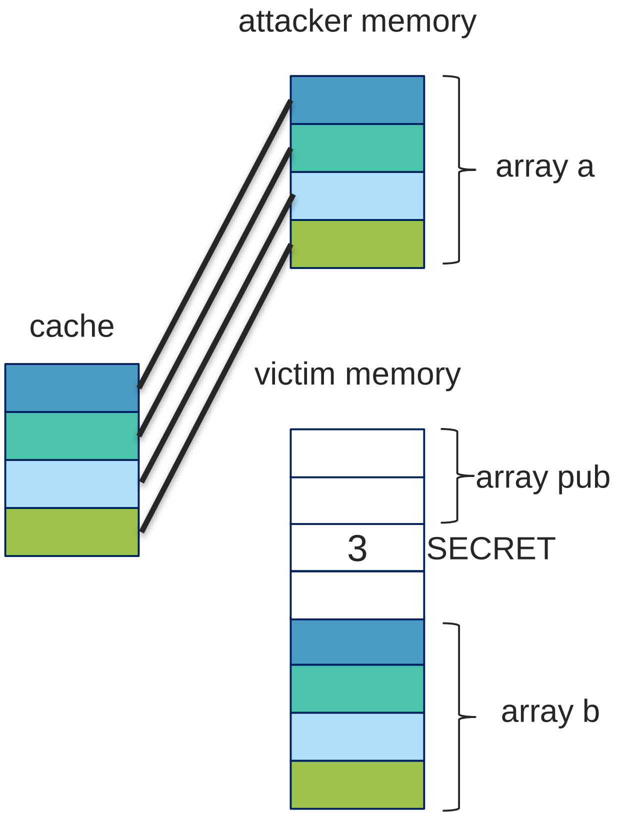
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



A simple example of a speculative execution attack

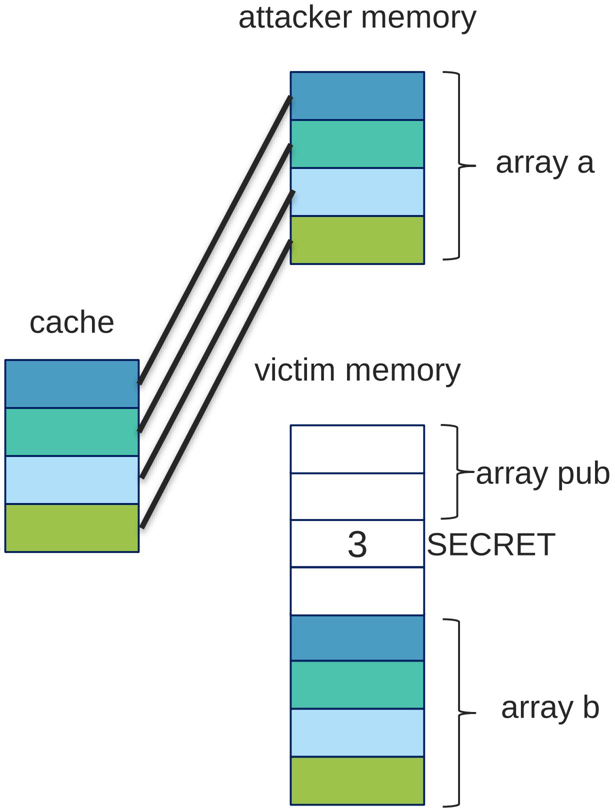
attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```

CPU speculatively executes the then branch



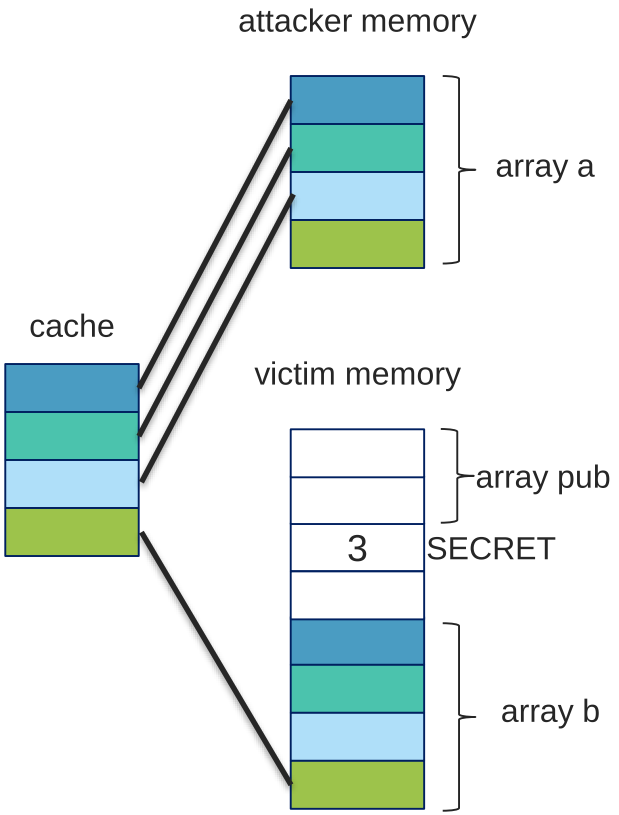
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



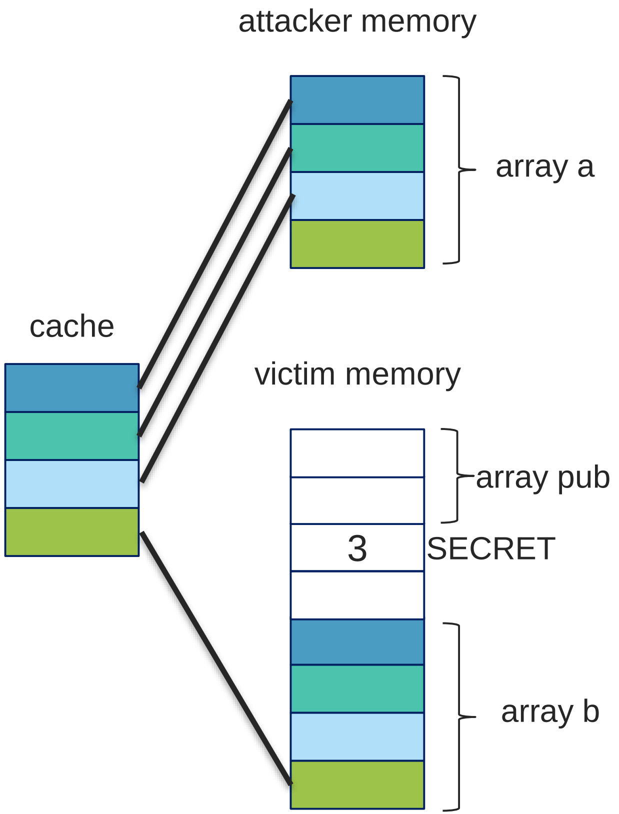
A simple example of a speculative execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



Speculative execution attacks

- This was a simplified Spectre Variant 1 attack
  - Many other variants exist
  - Meltdown/Foreshadow style attacks are similar but rely on the micro-architectural effects of out-of-order code execution that leads to an access control exception
- Note the **devastating** nature of this kind of attack
  - on any kind of software-enforced confidentiality
  - on any kind of hardware-enforced confidentiality where hardware resources are shared over protection boundaries
- Meltdown and Foreshadow are related attacks that exploit the fact that a processor may do speculative execution beyond a faulting instruction

Overview of the rest of the talk

- System model and attacker model
  - Recap of how C-like languages are executed on standard processors
  - Interactive attacker model
- Memory capabilities for run-time security
- Ownership types for compile-time security
- [The next wave of attacks]
- Conclusions

Conclusions

- System software plays a key role in ICT security
  - Vulnerabilities in system software impact all applications on the system
  - The boundaries of system software are fuzzy: your application likely relies on system software libraries
- System software is a clear example of the typical attacker-defender race
  - We are currently witnessing the transition to a new wave of attacks …
  - … as well as significant progress with closing the previous wave