CMPSC 447
Temporal Errors

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Temporal Errors

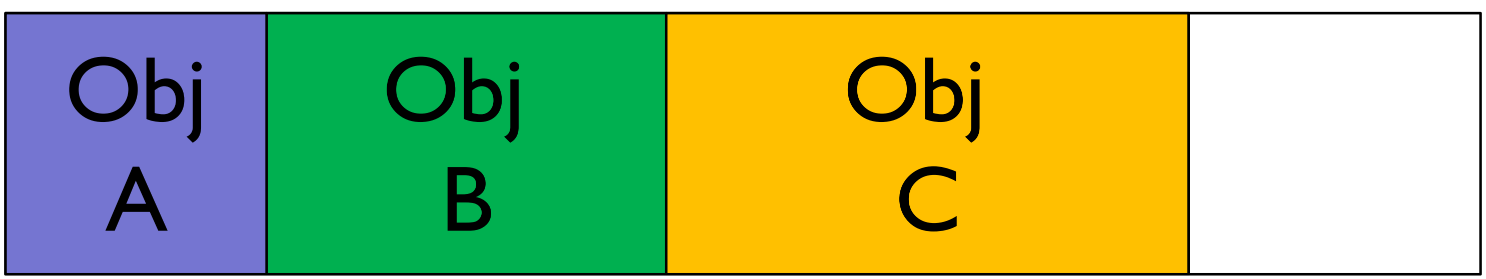
- Errors that permit access to memory outside of the object lifetime
  - These are called temporal errors
  - Access outside the expected "time"
- Most of these errors are permitted by simple programming flaws
  - Of the sort that you are not taught to avoid
  - Let's see how such errors can be avoided
- Some of the changes are rather simple

Temporal Errors

- A few of the exploits that we have discussed are the result of temporal errors

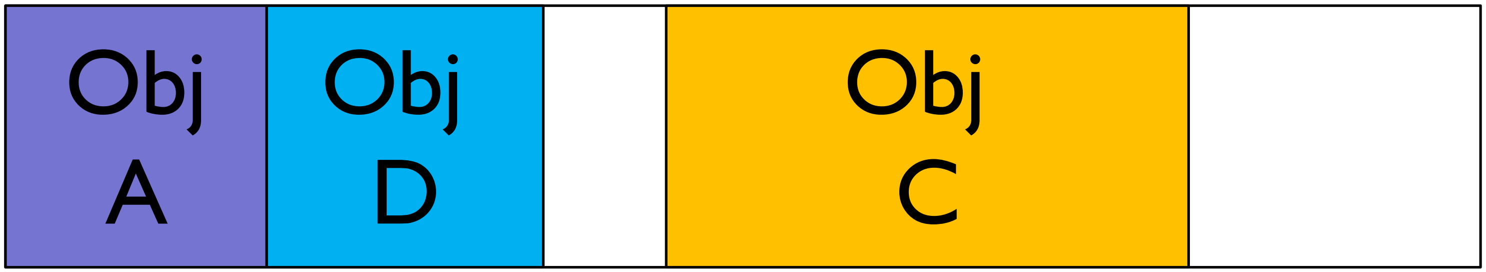
Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say “b”)
- Accessible: Adversary can control data written using the freed pointer
  - memcpy(b, adv-data, size);
- Target: Obtain a “write primitive”



Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say "b")
- Accessible: Adversary can control data written using the freed pointer
  - memcpy(b, adv-data, size);
- Target: Obtain a "write primitive" to new object D



What Is Going Wrong?

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?

What Is Going Wrong?

PENNSTATE

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference a memory region that does not hold a defined (assigned) object
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

What Is Going Wrong?

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference a memory region that does not hold a defined (assigned) object
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

What Is Going Wrong?

PENNSTATE 1855

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference a memory region that does not hold a defined (assigned) object
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object – release memory for reuse

What Is Going Wrong?

PENNSTATE 1855

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does "p" reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object

What Is Going Wrong?

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does "p" reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object
- Called "use before initialization" (UBI)
  - Allows an adversary to use reference value defined at the location used to declare "p" (not an assignment)
  - Could be anywhere

What Is Going Wrong?

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does "p" reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer

What Is Going Wrong?

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does "p" reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer
- Called "use after free" (UAF)
  - Allows an adversary to use reference to memory region that may be allocated a different object
  - Could be anywhere

Only on the Heap?

- Can temporal errors happen for stack objects?

Only on the Heap?

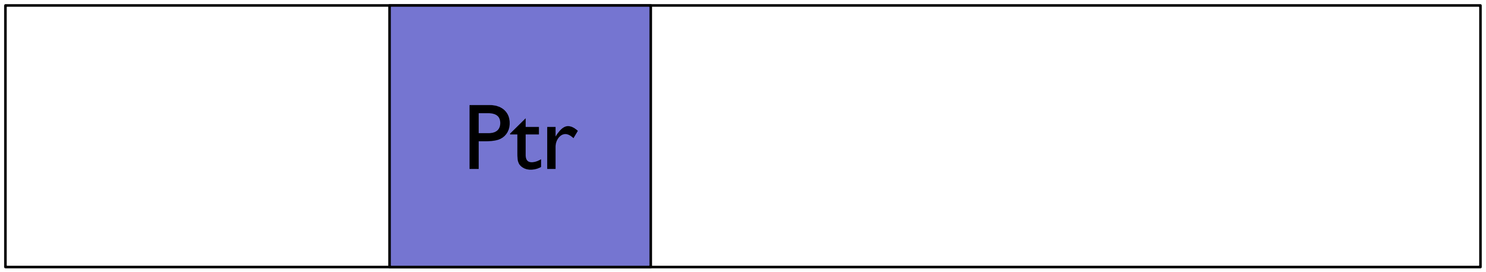
- Can temporal errors happen for stack objects?
  - Yes
- Use before initialization
  - Many references are allocated on the stack (like example)
  - As variables may be uninitialized
  - Do you initialize all variables?

Only on the Heap?

- Can temporal errors happen for stack objects?
  - Yes
- Use after free
  - Typically, exploits the deallocation of heap objects
  - But, stack objects are deallocated too
    - Just automatically by the runtime
  - Can you describe a "use after free" flaw for a stack object?

Exploiting Temporal Errors

- Use before initialization



- Questions to explore
  - Where is the pointer allocated in memory?
    - Can the adversary control what is written to that location
  - What is the pointer's value at initialization?
    - Can this reference a useful target object to attack?

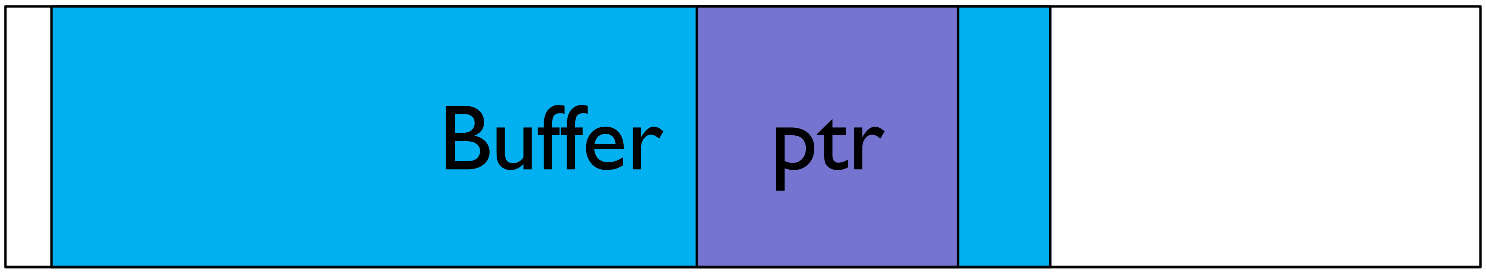
Exploiting Temporal Errors

- Use before initialization

- Assume function “A” calls functions “B” and “C”
  - When function “B” is called, a new stack frame is created
  - Using memory in the stack region
  - Suppose there is a string “buffer” built from adversary input
  - Then, function ”B” returns

Exploiting Temporal Errors

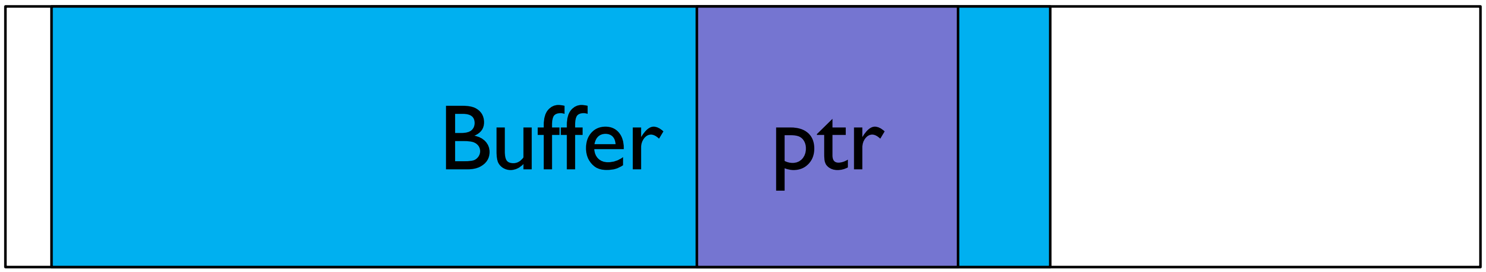
- Use before initialization



- Assume function “A” calls functions “B” and “C”
  - When function “C” is called, a new stack frame is created
  - Using memory in the stack region – used by function “B”
  - Suppose there is a local variable pointer “ptr” declared in function “C”
  - But, ”ptr” is not initialized – what is the value of “ptr”?

Exploiting Temporal Errors

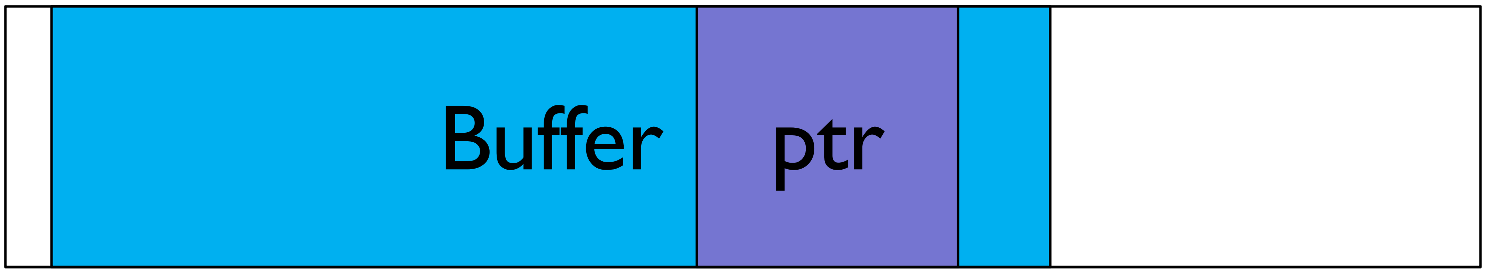
- Use before initialization



- Assume function "A" calls functions "B" and "C"
  - Suppose there is a local variable pointer "ptr" declared in "C"
  - But, "ptr" is not initialized – what is the value of "ptr"?
  - The value of "ptr" is the value of the bytes of "buffer"
  - Suppose "ptr" is used before initialized. Can you exploit this? How?

Exploiting Temporal Errors

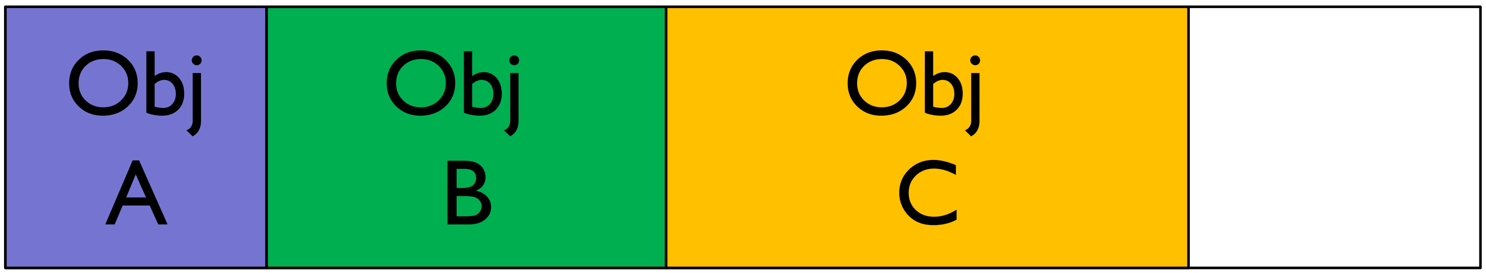
- Use before initialization



- Can you exploit this? How?
  - Use the debugger to determine the relative offset of “buffer” and “ptr”
  - Build filler from the start of the buffer to the start of the pointer “ptr”
  - Then, insert the address of the target object in “ptr”
  - Now, you can access the target via “ptr”

Exploiting Temporal Errors

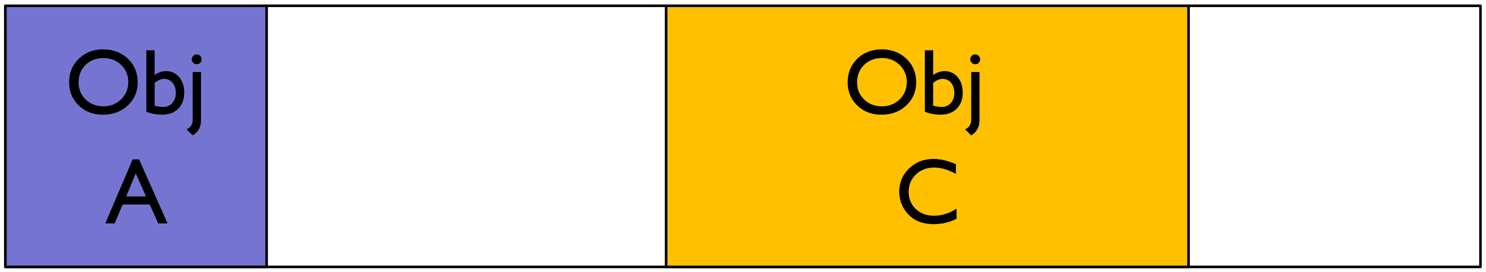
- Use after free



- Assume you have a heap as shown
  - Focus on object ”B”
  - You have a reference to “B” – say pointer “b”

Exploiting Temporal Errors

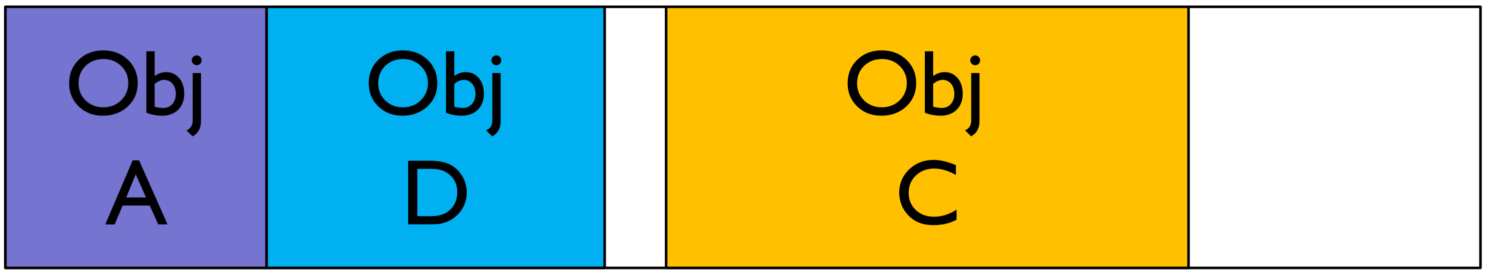
- Use after free



- Assume you have a heap as shown
  - Object "B" is deallocated
  - And you still have a reference to "B" – pointer "b"
  - And, pointer "b" may be have "uses" after the deallocation of object "B"
  - But, the allocator is free to reuse the memory region

Exploiting Temporal Errors

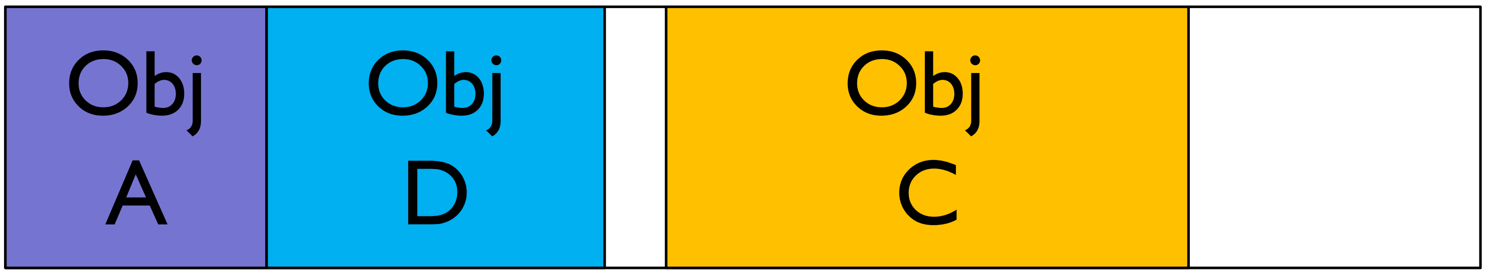
- Use after free



- Assume you have a heap as shown
  - The allocator chooses to use the memory region for object “D”
  - So, a “use” of pointer “b” will access the object ”D” instead
  - If object “B” and object “D” are of different types, you can exploit the differences

Exploiting Temporal Errors

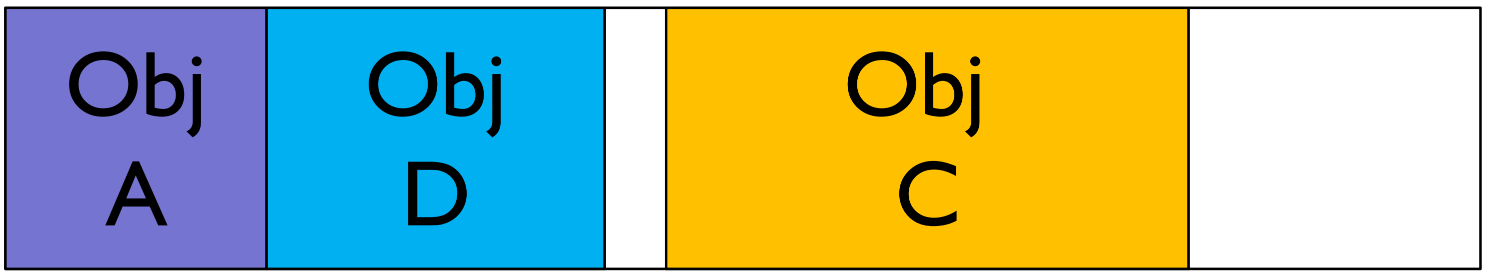
- Use after free



- How exactly do you exploit this?
    - Create and free an object – object “B” - record its location using the debugger
    - With a pointer with a use-after-free flaw – pointer “b”
    - Cause program to allocate instances of the target object “D”
    - Find when a “D” is in the location of the original object “B” using the debugger

Exploiting Temporal Errors

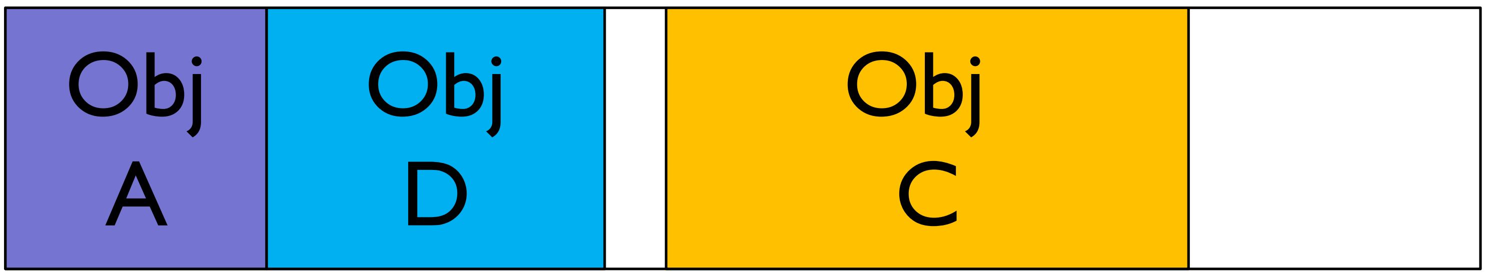
- Use after free



- How exactly do you exploit this?
  - To get new allocation in the same spot
  - Size of Obj “D” <= Obj “B” - Equal only in some cases

Exploiting Temporal Errors

- Use after free



- How exactly do you exploit this?
  - To exploit object ”D”
  - Should be a target field in object “D” that can be modified or read using the stale pointer ”b”
  - Suppose “D” has a pointer field that is aligned with a data field in the type of object “B” that can be modified with “b”

Fundamental Problem?

- What is the fundamental problem that causes temporal errors?

Fundamental Problem?

- What is the fundamental problem that causes temporal errors?
  - We have pointers (references)
  - We have memory regions (objects)
  - We have assignments of pointers to memory regions
- But, the actual relationships may change
  - A pointer is assigned to some value when declared that could be a legal memory region
    - Before assignment – permitting use before initialization
  - Memory regions may be reused for other objects
    - After assignment – permitting use after free

Obvious Solution in C

- So, do you see an "obvious" solution to prevent exploitable temporal errors?

Obvious Solution in C

PENNSTATE 1855

- So, do you see an "obvious" solution to prevent exploitable temporal errors?
    - Shouldn't pointers either reference their assigned and allocated objects or be invalid?

Zeroing Pointers

- Set every pointer value to zero on initialization
  - Assign to zero on the stack
    - char *p = NULL;
  - Zero memory allocated from the heap (including its pointers)
    - obj = (char *) calloc(size, 1);
- As a result, no pointer will refer to any active memory object before it is assigned
  - Prevents use-before-initialization attacks trivially
  - Downside?

Zeroing Pointers

- Downside? Cost of doing extra assignments
  - Can add up
- On the other hand, crashing the program beats an exploit, and such a use before initialization is an error
  - Deserves a trap
- How can you reduce the number of assignments necessary to prevent any exploit of use-before-initialization vulnerabilities?

Zeroing Pointers

PENNSTATE

- How can you reduce the number of assignments necessary to prevent any exploit of use-before-initialization vulnerabilities?
  - Determine which pointers "may" be used before initialization and initialization all of them
  - Can figure the answer to questions like this out with "static analysis"
    - Will discuss a static analysis for detecting use-before-initialization later in the class

Obvious Solution in C

- So, do you see an “obvious” solution to prevent exploitable temporal errors?
  - Would zeroing pointer values also work to prevent the exploit of use-after-free vulnerabilities?

Zeroing Pointers

- Yes! Set every pointer value to zero on deallocation
  - Zero pointers on deallocation from the heap
    - free(p), p = 0;
  - Trickier on the stack
    - In theory, no stack reference should outlive its assignment
    - But, hard to guarantee since deallocation is implicit
- Also, the cost of zeroing on deallocation can be worse
  - Since not done at all normally

Other Ideas

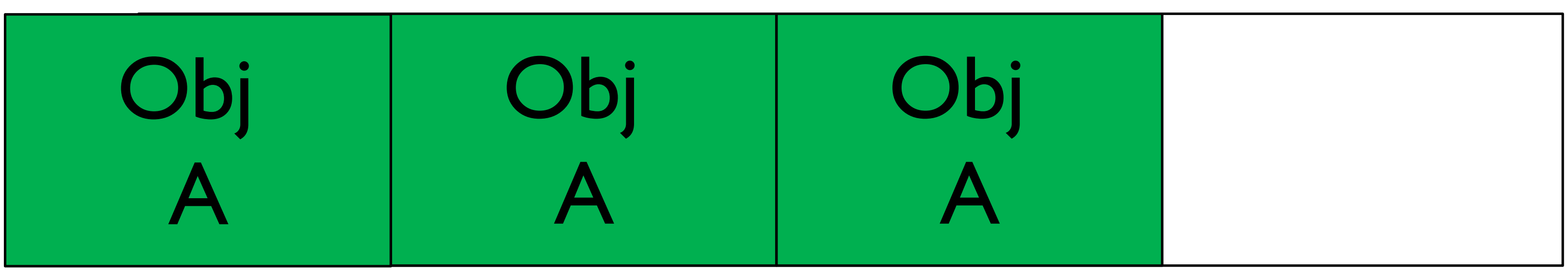
- Can you think of any other ways to prevent use-after-free exploits?
  - May be a little crazy

Alternatives

- Hypothesis: memory is so cheap and abundant, we just do not need to deallocate
  - Will be some cases where this is not going to work
  - But, for others, why risk attack?
- Hypothesis: garbage collection
  - Too expensive for C
- Hypothesis: temporal safety like Rust's "safe" objects
  - Harder to program with lifetimes and ownerships
- Hypothesis: use type-specific allocation
  - All objects and fields are aligned

Type-Specific Pools

- Hypothesis: use type-specific allocation
  - All objects and fields are aligned
- Type-specific pools
  - Allocate an object of type A from a memory region containing only objects of type A
  - Does not prevent use-after-free vulnerabilities, but limits the exploit potential by preventing a reference of one type from exploiting an object of another type



Take Away

PENNSTATE 1855

- Manual (heap) and implicit (stack) memory management in C permits temporal errors
  - So, temporal errors have become common, especially now that defenses for spatial errors have improved
- Exploiting temporal errors involves controlling the relationship of a pointer and the object referenced
  - Set the pointer value or the object at a location
- Preventing temporal errors is trivial conceptually
  - But, a bit more expensive than people will accept yet