Buffer Overflows

- Buggy C code can write beyond end of buffer / array
- Ideal solution is to use a language that enforces bounds (i.e. Python / Java)
  - C is used for many valuable applications / libraries so we can't abandon it and we can't avoid writing new C code easily
- Perfect programming would check bounds 100% of the time, but no one is perfect
- We need defenses that make buffer overflows hard to exploit

Classic Stack Buffer Overflows

- Attack
  - Write instructions into buffer on stack
  - Overwrite return PC to point to attacker's instructions
- Defense:
  - OS tells hardware not to execute on the stack
  - I.e. OS can set no execute bit on each bit of the stack page
- There's way to get around non-executable stack (i.e. find a ROP gadget like we did previously)
- Another defense: stack canaries
  - Detects modification of return PC on stack before it is used by RET
  - Compiler generates code that pushes a canary value on stack and checks if that value has been modified before using the RET
  - Sits between variables and the return address
  - We should use a random number chosen at program start for the canary so attacker can't just learn to avoid it
- Another defense: address space layour randomization (ASLR)
  - Place process memory at random addresses
  - Adversary does not know precise address of application stack / code / heap
- Buffer overflows can still:
  - SOmehow get access to the canary
  - Overwrite other variables on the stack
  - Overflow global variables into one another
  - Overflow a heap-allocated buffer

Heap Attacks

- Can attacker use overflowing heap-allocated buffers?
  - They would have to predict what is after p in memory, which requires knowing how free / used malloc blocks work
- If the attacker overflows a malloc()ed free block:
  - Attacker can modify the next / prev pointers in the next block to be prev = x, next = y
  - When malloc allocates a free block b, it does b->next->prev = b->prev and b->prev->next = b->next
  - If this happens to choose the block we just contaminated, we now have b->next->prev = b->prev
    - Accessing ->prev essentially just accesses at offset 0 (if prev is the first pointer listed in the block)
    - Therefore this is just doing `*y = x` which is very powerful since we can use this to overwrite return PC

Attackers

- Attackers have to:
  - Find a buffer overflow bug
  - Find a way to get the program to execute the buggy code so attacker bytes overflow buffer
  - Understand malloc() implementation

    - Find a code pointer and guess its address
    - Guess the address of the buffer to run the attacker instructions
- Attacks require effort / skill and understand corner pieces and have to assembly a lot of fragile pieces
- But clever enough attackers with enough small bugs can exploit programs

Automatic Array Bounds Checking

- Paper: shows how one can retrofit automatic array bounds checking to existing C programs
    - We need to modify the compiler, recompile applications, and perhaps modify the applications

Fat Pointers

- Straightforward, but not practical
- Each pointer contains start / end / current pointer value
- Malloc has to be modified to set these and during dereference, have to check these bounds upon dereference
- Problems:
    - Checks can be expensive
    - Not compatible with unmodified libraries
    - Changes data structure sizes
    - Updates to fat pointers are no longer atomic since they span multiple words (32 bits)

Separate Table

- Keep bounds information in a separate table so we don't have to change pointer representation
- For each allocated pointer, store start and size of the object
    - malloc() or compiler generated code keeps track of this table
    - If the pointer leaves the object, set an OOB bit in the pointer
    - If the pointer is dereferenced with OOB bit set, then the compiler panics
- Challenges:
    - We have to be able to index into this table with both the original pointer start p and also other valid locations in that pointer (i.e. $p + 10$)
    - Is arithmetic modifies p to return back to its original object, we have to detect that and clear the OOB
        - After it leaves the object, we have to know what the original object was
    - Prevent table from being huge / expensive to access

Baggy Bounds

- Uses power-of-two trick to optimize the bounds table
- Each table entry holds just a size in it as a log base 2, so it can fit in a single byte
    - I.e. storing 20 in a table entry means there is an allocation of 1 megabyte
- To make this work, we only allocate power-of-two sizes on that same power-of-two boundaries
    - I.e. if we allocate something of size 256, it has to go on a 256 boundary
    - Compiled code computes start an object from the pointer and table entry by just clearing the the lower $\log_2(size)$ bits
- Each table entry covers slot_size = 16 bytes, so the table isn't huge
    - A 64 byte object at address x uses 4 table entries starting at table[x / 16]
- To make OOB pointers crash on dereference, we make the MSB the OOB bit
    - We then make pages in the upper half of the address space inaccessible
- The OOB mark means that the ptr is within slot_size / 2 of the oriignal object
    - If we move more than slot_size / 2 away, then we panic as well
    - This way, we can always recover what the actual valid region is because we just check if we are OOB and if we are, we look at which border we are closest to

Problems

- Baggy bounds can panic even when the code is legal
  - If C code casts pointers to integers, then this can break due to the OOB bit
- Overflow bugs baggy bounds might not catch:
  - Overflow of array inside a larger malloc()'d struct
  - Cast pointer to int, modify, and then cast back
  - Application might implement its own allocator
  - Dangling pointers to re-allocated memory (use-after-free)