Canary in the coal mine

- StackGuard: 0x00 0D 0A FF
  - 0: String functions
  - newline: fgets(), etc.
  - -1: getc()
  - carriage return: similar to newline?
- Doesn't stop: memcpy, custom loops

Adjacent Canary Idea

- Terminator canary
  - Value hard to reproduce because it would tell the copy to stop
  - StackGuard: 0x00 0D 0A FF
    - 0: String functions
    - newline: fgets(), etc.
    - -1: getc()
    - carriage return: similar to newline?
  - Doesn't stop: memcpy, custom loops

Random Canary

- Can't reproduce because attacker can't guess
- For efficiency, usually one per execution
- Ineffective if disclosed

XOR Canary

- Want to protect against non-sequential overwrites
- XOR return address with value c at entry
- XOR again with c before return
- Standard choice for c: see random canary

Further Refinements

- More flexible to do earlier in compiler
- Rearrange buffers after other variables
  - Reduce chance of non-control overwrite
- Skip canaries for functions with only small variables
  - Who has an overflow bug in an 8-byte array?
What's usually not protected?
- Backwards overflows
- Function pointers
- Adjacent structure fields
- Adjacent static data objects

Where to keep canary value
- Fast to access
- Buggy code/attacker can't read or write
- Linux/x86: %gs:0x14

Complex anti-canary attack
- Canary not updated on fork in server
- Attacker controls number of bytes overwritten

Shadow return stack
- Suppose you have a safe place to store the canary
- Why not just store the return address there?
- Needs to be a separate stack
- Ultimate return address protection

Outline
- Return address protections
- Announcements intermission
- ASLR and counterattacks
- Testing and fuzzing

Brief announcements
- Problem set 1 is available on the public web page now
  - Due Friday, 2/17
- The first midterm exam will be next Tuesday (2/21) in class
  - Open book, open notes, no electronics
  - You will have the whole class period
  - Topics will be memory safety bugs and attacks, and threat modeling
  - Similar concepts, but less depth, than labs and p-set

Note to early readers
- This is the section of the slides most likely to change in the final version
- If class has already happened, make sure you have the latest slides for announcements
Outline
Return address protections
Announcements intermission
ASLR and counterattacks
Testing and fuzzing

Basic idea
"Address Space Layout Randomization"
Move memory areas around randomly so attackers can't predict addresses
Keep internal structure unchanged
E.g., whole stack moves together

Code and data locations
Execution of code depends on memory location
E.g., on x86-64:
Direct jumps are relative
Function pointers are absolute
Data can be relative (%rip-based addressing)

Relocation (Windows)
Extension of technique already used in compilation
Keep table of absolute addresses, instructions on how to update
Disadvantage: code modifications take time on load, prevent sharing

PIC/PIE (GNU/Linux)
"Position-Independent Code / Executable"
Keep code unchanged, use register to point to data area
Disadvantage: code complexity, register pressure hurt performance (especially 32-bit)

What’s not covered
Main executable (Linux PIC)
Incompatible DLLs (Windows)
Relative locations within a module/area

Entropy limitations
Intuitively, entropy measures amount of randomness, in bits
Random 32-bit int: 32 bits of entropy
ASLR page aligned, so at most $32 - 12 = 20$ bits of entropy on x86-32
Other constraints further reduce possibilities

Leakage limitations
If an attacker learns the randomized base address, can reconstruct other locations
Any stack address → stack unprotected, etc.
Testing and security

"Testing shows the presence, not the absence of bugs" – Dijkstra

Easy versions of some bugs can be found by targeted tests:
- Buffer overflows: long strings
- Integer overflows: large numbers
- Format string vulnerabilities: %x

Random or fuzz testing

Random testing can also sometimes reveal bugs
- Original 'fuzz' (Miller): program </dev/urandom
- Even this was surprisingly effective

Mutational fuzzing

Instead of totally random inputs, make small random changes to normal inputs
- Changes are called mutations
- Benign starting inputs are called seeds
- Good seeds help in exercising interesting/deep behavior

Grammar-based fuzzing

Observation: it helps to know what correct inputs look like
- Grammar specifies legal patterns, run backwards with random choices to generate
- Generated inputs can again be basis for mutation
- Most commonly used for standard input formats
  - Network protocols, JavaScript, etc.

What if you don’t have a grammar?

- Input format may be unknown, or buggy and limited
- Writing a grammar may be too much manual work
- Can the structure of interesting inputs be figured out automatically?

Coverage-driven fuzzing

Instrument code to record what code is executed
- An input is interesting if it executes code that was not executed before
- Only interesting inputs are used as basis for future mutation

AFL

Best known open-source tool, pioneered coverage-driven fuzzing
- American Fuzzy Lop, a breed of rabbits
- Stores coverage information in a compact hash table
- Compiler-based or binary-level instrumentation
- Has a number of other optimizations