CSci 4271W
Development of Secure Software Systems
Day 9: Threat modeling, defenses
Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline
- ROP exercise final followup
- Threat modeling: printer manager
- Return address protections
- ASLR and counterattacks

ROP mprotect example
I’ll show this in Inkscape

Setting: shared lab with printer
- Imagine a scenario similar to CSE Labs
  - Computer labs used by many people, with administrators
- Target for modeling: software system used to manage printing
  - Similar to real system, but use your imagination for unknown details

Data flow diagram
- Show structure of users, software/hardware components, data flows, and trust boundaries
- For this exercise, can mix software, OS, and network perspectives
- Include details relevant to security design decisions
- Take 15 minutes to draw with your neighbors

STRIDE threat brainstorming
- Think about possible threats using the STRIDE classification
- Are all six types applicable in this example?
- Take 10 minutes to brainstorm with your neighbors

Outline
- ROP exercise final followup
- Threat modeling: printer manager
- Return address protections
- ASLR and counterattacks
Canary in the coal mine

Adjacent canary idea

Terminator canary
- Value hard to reproduce because it would tell the copy to stop
- StackGuard: 0x00 OD 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

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
- Other constraints further reduce possibilities

Leakage limitations

- If an attacker learns the randomized base address, can reconstruct other locations
- Any stack address $\rightarrow$ stack unprotected, etc.