CSci 4271W
Development of Secure Software Systems
Day 9: Auditing, more countermeasures

Stephen McCamant
University of Minnesota, Computer Science & Engineering

## Outline

ROP exercise debrief
Advice on code auditing
Announcements intermission
ASLR and counterattacks
Return address protections

## Setup

- Key motivation for ROP is to disable $W \oplus X$
- Can be done with a single syscall, similar to `execve` shellcode
- Your exercise: put together such shellcode from a limited gadget set
- Puzzle/planning aspect: order to avoid overwriting

## Outline

ROP exercise debrief
Advice on code auditing
Announcements intermission
ASLR and counterattacks
Return address protections

## Main source for this advice

- Chapter 4 of *The Art of Software Security Assessment*, by Mark Dowd, John McDonald, and Justin Schuh
- The reading has more explanations and details
- Course-only chapter copy on the Canvas page
- They call this topic “application review”

## The context of auditing

- Any process should be result-driven
- Plan the scope of what you're going to do before diving in
- Be prepared to spend time afterwards explaining your result, and maybe helping fix the problems

## Structure based on design info

- The structure of the process depends on reliable design information
  - E.g., from threat modeling
- If you have it, top-down is most efficient
- Bottom-up helps you learn the design, but is slower
- A hybrid is also possible

## Planning and iteration

- Choose goals and scope (e.g., based on business context)
- Budget enough time
  - 100 to 1,000 LOC/hr for a professional
- Work for a while with one goal/strategy, periodically reassess and maybe change

## Notes and collaboration

- Several reasons to keep notes as you go:
  - "Ideas list" of leads to explore later
  - Preparing to produce documentation as an end product
- Ease of coordination depends on software modularity
  - For Project 0.5, could be independent or pair-programming

## Tracing code and data flow

- Control-flow tracing: what calls what, under what circumstances?
- Data-flow tracing: how does information go from one place to another?
- Can be forward: from an entry point
- Or backwards from a *candidate point*
  - E.g., risky operation

## Or not tracing

- Often, following long flows and remembering a deep context won't be the best use of your time
- Aim to mostly be looking at one function at a time

## Three kinds of strategies

- How can you organize your auditing work?
- Based on code comprehension
- Based on candidate points
- Based on design generalization

## Code comprehension strategies

- CC1: Trace malicious input
- CC2: Analyze a module
- CC3: Analyze an algorithm
- CC4: Analyze a class or object
- CC5: Trace black box hits

## Candidate point strategies

- CP1: General candidate point approach
- CP2: Automated source analysis tool
- CP3: Simple lexical candidate points
- CP4: Simple binary candidate points
- CP5: Black-box-generated candidate points
- CP6: Application-specific candidate points

## Design generalization strategies

- DG1: Model the system
- DG2: Hypothesis testing
- DG3: Deriving purpose and function
- DG4: Design conformity check

## Testing and desk-checking

- Testing can be used to confirm or disprove a theory
  - Sometimes you can test all the code at once
  - Other times, isolate a smaller code unit to test, maybe with a debugger
- A desk-check is manually walking through a test case on a piece of code
  - Construct a table of values over time
  - Can be valuable *because* it makes you slow down

## Constraints and data operations

- When testing with numeric data, think about the constraints on what values are possible
  - These may come from other places in the code
- For richer data types like strings, design your tests based on how the values are processed
  - E.g., transformation, validation, parsing, system usage

## Outline

ROP exercise debrief

Advice on code auditing

Announcements intermission

ASLR and counterattacks

Return address protections

## Midterm next Tuesday

- The first midterm exam will be next Tuesday (2/20) 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
  - Samples of past midterms on the schedule page

## Outline

ROP exercise debrief

Advice on code auditing

Announcements intermission

ASLR and counterattacks

Return address protections

## 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 $\rightarrow$ stack unprotected, etc.

## Outline

ROP exercise debrief

Advice on code auditing

Announcements intermission

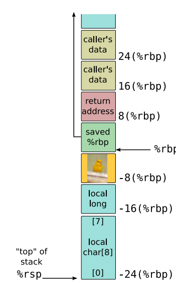
ASLR and counterattacks

Return address protections

## Canary in the coal mine



## 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/x86-64: `%fs:0x28`

## Complex anti-canary attack

- Canary not updated on `fork` in server
- Attacker controls number of bytes overwritten

## Complex anti-canary attack

- Canary not updated on `fork` in server
- Attacker controls number of bytes overwritten
- ANRY BNRY CNRY DNRY ENRY FNRY
- search $2^{32} \longrightarrow$ search $4 \cdot 2^8$

## 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