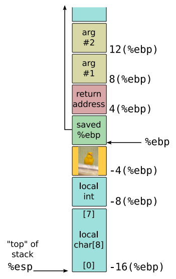
# CSci 4271W
# Development of Secure Software Systems
# Day 27: More low-level defenses

Stephen McCamant
University of Minnesota, Computer Science & Engineering

## Outline

## 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: `%gs:0x14`

## 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

## Outline

## Why is this important?

- This is a relatively new class: help us figure out what we should do differently next time
- Which things worked well, which things should be different?
- What should there be more of, and what less of?
- How do the topics compare with what you expected?

## SRT logistics

- All online this semester
- Requested but not required; can't affect your grade one way or the other
- Primary evaluation combines Prof. McCamant and the course
- Please also evaluate Saugata separately if you have comments or suggestions about his performance
- Open through the last regular class day

## SRT URL

- `https://srt.umn.edu/blue`
- We'll have a 15-minute break in class material that we request you use for filling out the evaluation

## Outline

- Return address protections
- Intermission for SRT
- Report revision suggestions
- ASLR and counterattacks
- Control-flow integrity (CFI)
- More modern exploit techniques

## Logistics reminders

- Two components: fixing patch and revised report
- Take advantage of sample attacks posted on Piazza
- Page limit increased to 6 pages, may need to reduce some old material
  - Still need to decide what's most important
- Due on Canvas by Wednesday night

## Big picture

- If you didn't follow the requirements the first time, do this time
- Don't spend too much time describing the program
- Your attack understanding should be supported by concrete details
  - Use exemption of figures from length limit

## Writing reminders

- Use complete sentences (e.g., avoid comma splices)
- Avoid being too "editorial" (facts over opinions)

## Outline

- Return address protections
- Intermission for SRT
- Report revision suggestions
- ASLR and counterattacks
- Control-flow integrity (CFI)
- More modern exploit techniques

## 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 32-bit x86:
  - Direct jumps are relative
  - Function pointers are absolute
  - Data must be absolute

## 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

## What's not covered

- Main executable (Linux 32-bit 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.

## Outline

## Some philosophy

- Remember allowlist vs. denylist?
- Rather than specific attacks, tighten behavior
  - Compare: type system; garbage collector vs. use-after-free
- CFI: apply to control-flow attacks

## Basic CFI principle

- Each indirect jump should only go to a programmer-intended (or compiler-intended) target
- I.e., enforce call graph
- Often: identify disjoint target sets

## Approximating the call graph

- One set: all legal indirect targets
- Two sets: indirect calls and return points
- $n$ sets: needs possibly-difficult points-to analysis

## Target checking: classic

- Identifier is a unique 32-bit value
- Can embed in effectively-nop instruction
- Check value at target before jump
- Optionally add shadow stack

## Target checking: classic

```
cmp [ecx], 12345678h
jne error_label
lea ecx, [ecx+4]
jmp ecx
```

## Challenge 1: performance

- In CCS'05 paper: 16% avg., 45% max.
  - Widely varying by program
  - Probably too much for on-by-default
- Improved in later research
  - Common alternative: use tables of legal targets

## Challenge 2: compatibility

- Compilation information required
- Must transform entire program together
- Can't inter-operate with untransformed code

## Recent advances: COTS

- Commercial off-the-shelf binaries
- CCFIR (Berkeley+PKU, Oakland'13): Windows
- CFI for COTS Binaries (Stony Brook, USENIX'13): Linux

## COTS techniques

- CCFIR: use Windows ASLR information to find targets
- Linux paper: keep copy of original binary, build translation table

## Control-Flow Guard

- CFI-style defense now in latest Windows systems
- Compiler generates tables of legal targets
- At runtime, table managed by kernel, read-only to user-space

## Coarse-grained counter-attack

- "Out of Control" paper, Oakland'14
- Limit to gadgets allowed by coarse policy
  - Indirect call to function entry
  - Return to point after call site ("call-preceded")
- Use existing direct calls to `VirtualProtect`
- Also used against kBouncer

## Control-flow bending counter-attack

- Control-flow attacks that still respect the CFG
- Especially easy without a shadow stack
- Printf-oriented programming generalizes format-string attacks

## Outline

Return address protections

Intermission for SRT

Report revision suggestions

ASLR and counterattacks

Control-flow integrity (CFI)

More modern exploit techniques

## Target #1: web browsers

- Widely used on desktop and mobile platforms
- Easily exposed to malicious code
- JavaScript is useful for constructing fancy attacks

## Heap spraying

- How to take advantage of uncontrolled jump?
- Maximize proportion of memory that is a target
- Generalize NOP sled idea, using benign allocator
- Under W$\oplus$X, can't be code directly

## JIT spraying

- Can we use a JIT compiler to make our sleds?
- Exploit unaligned execution:
  - Benign but weird high-level code (bitwise ops. with constants)
  - Benign but predictable JITted code
  - Becomes sled + exploit when entered unaligned

## JIT spray example

```
25 90 90 90 3c   and $0x3c909090,%eax
25 90 90 90 3c   and $0x3c909090,%eax
25 90 90 90 3c   and $0x3c909090,%eax
25 90 90 90 3c   and $0x3c909090,%eax
```

## JIT spray example

```
90              nop
90              nop
90              nop
3c 25           cmp $0x25,%al
90              nop
90              nop
90              nop
3c 25           cmp $0x25,%al
```

## Use-after-free

- Low-level memory error of choice in web browsers
- Not as easily audited as buffer overflows
- Can lurk in attacker-controlled corner cases
- JavaScript and Document Object Model (DOM)

## Sandboxes and escape

- Chrome NaCl: run untrusted native code with SFI
  - Extra instruction-level checks somewhat like CFI
- Each web page rendered in own, less-trusted process
- But not easy to make sandboxes secure
  - While allowing functionality

## Chained bugs in Pwnium 1

- Google-run contest for complete Chrome exploits
  - First edition in spring 2012
- Winner 1: 6 vulnerabilities
- Winner 2: 14 bugs and "missed hardening opportunities"
- Each got $60k, bugs promptly fixed