Outline

Exploiting other vulnerabilities
Return address protections
Announcements intermission
ASLR and counterattacks
W+X (DEP)

Non-control data overwrite

- Overwrite other security-sensitive data
- No change to program control flow
- Set user ID to 0, set permissions to all, etc.

Heap meta-data

- Boundary tags similar to doubly-linked list
- Overwritten on heap overflow
- Arbitrary write triggered on free
- Simple version stopped by sanity checks

Heap meta-data

Use after free

- Write to new object overwrites old, or vice-versa
- Key issue is what heap object is reused for
- Influence by controlling other heap operations

Integer overflows

- Easiest to use: overflow in small (8-, 16-bit) value, or only overflowed value used
- 2GB write in 100 byte buffer
  - Find some other way to make it stop
- Arbitrary single overwrite
  - Use math to figure out overflowing value

Null pointer dereference

- Add offset to make a predictable pointer
  - On Windows, interesting address start low
- Allocate data on the zero page
  - Most common in user-space to kernel attacks
  - Read more dangerous than a write
Format string attack

- Attacker-controlled format: little interpreter
- Step one: add extra integer specifiers, dump stack
  - Already useful for information disclosure

Format string attack layout

Format string attack layout

Format string attack: overwrite

- \%n specifier: store number of chars written so far to pointer arg
- Advance format arg pointer to other attacker-controlled data
- Control number of chars written with padding
- On x86, use unaligned stores to create pointer

Outline

Exploiting other vulnerabilities
Return address protections
Announcements intermission
ASLR and counterattacks
W<X (DEP)

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

**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

Exploiting other vulnerabilities
Return address protections
Announcements intermission
ASLR and counterattacks
W*\(X\) (DEP)

Pre-proposals due tonight

- Most groups formed?
- One PDF per group, include schedule choices
- Submit via Canvas by 11:59pm

Readings reminders

- For last Wed: buffer overflows and defenses
- For Monday: Attack techniques (under ASLR)
- Now also: academic (ACM) papers, campus/proxy downloads

Exercise 1 submissions via Gradescope

- Draft answers in plain text, submit in online text boxes
- Due a week from tonight, available now
- Bring your questions to Piazza or office hours

Outline

Exploiting other vulnerabilities
Return address protections
Announcements intermission
ASLR and counterattacks
W*\(X\) (DEP)

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.

GOT hijack (Müller)

- Main program fixed, libc randomized
- PLT in main program used to call libc
- Rewrite PLT to call attacker's favorite libc functions
- E.g., turn printf into system

printf@plt: jmp *0x8049678
...
system@plt: jmp *0x804967c
...
0x8049678: <addr of printf in libc>
0x804967c: <addr of system in libc>

ret2pop (Müller)

- Take advantage of shellcode pointer already present on stack
- Rewrite intervening stack to treat the shellcode pointer like a return address
  - A long sequence of chained returns, one pop
Outline
- Exploiting other vulnerabilities
- Return address protections
- Announcements intermission
- ASLR and counterattacks
- \( W \oplus X \) (DEP)

Basic idea
- Traditional shellcode must go in a memory area that is
  - writable, so the shellcode can be inserted
  - executable, so the shellcode can be executed
- But benign code usually does not need this combination
- \( W \oplus X \), really \( \sim (W \land X) \)

Non-writable code, \( X \rightarrow \neg W \)
- E.g., read-only .text section
- Has been standard for a while, especially on Unix
- Lets OS efficiently share code with multiple program instances

Non-executable data, \( W \rightarrow \neg X \)
- Prohibit execution of static data, stack, heap
- Not a problem for most programs
  - Incompatible with some GCC features no one uses
  - Non-executable stack opt-in on Linux, but now near-universal

Implementing \( W \oplus X \)
- Page protection implemented by CPU
  - Some architectures (e.g. SPARC) long supported \( W \oplus X \)
  - x86 historically did not
    - One bit controls both read and execute
    - Partial stop-gap "code segment limit"
  - Eventual obvious solution: add new bit
    - NX (AMD), XD (Intel), XN (ARM)

One important exception
- Remaining important use of self-modifying code: just-in-time (JIT) compilers
  - E.g., all modern JavaScript engines
- Allow code to re-enable execution per-block
  - mprotect, VirtualProtect
  - Now a favorite target of attackers

Counterattack: code reuse
- Attacker can't execute new code
- So, take advantage of instructions already in binary
- There are usually a lot of them
- And no need to obey original structure

Classic return-to-libc (1997)
- Overwrite stack with copies of:
  - Pointer to libc's system function
  - Pointer to "/bin/sh" string (also in libc)
- The system function is especially convenient
- Distinctive feature: return to entry point
Chained return-to-libc

- Shellcode often wants a sequence of actions, e.g.
  - Restore privileges
  - Allow execution of memory area
  - Overwrite system file, etc.
- Can put multiple fake frames on the stack
  - Basic idea present in 1997, further refinements

Beyond return-to-libc

- Can we do more? Oh, yes.
- Classic academic approach: what’s the most we could ask for?
- Here: “Turing completeness”
- How to do it: reading for Monday

Next slides

- Return-oriented programming (ROP)
  - And counter-defenses
- Control-flow integrity (CFI)