Outline

- Threat modeling: printer manager (cont’d)
- Return-oriented programming (ROP)
- Announcements intermission
- ROP shellcoding exercise
- Testing and fuzzing

STRIDE threat brainstorming

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

STRIDE threat taxonomy

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

Outline

Threat modeling: printer manager (cont’d)
Return-oriented programming (ROP)
Announcements intermission
ROP shellcoding exercise
Testing and fuzzing

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
Pop culture analogy: ransom note trope

```
come  at midnight. bring
$10,000  in  unmarked  bills
```

Basic new idea

- Treat the stack like a new instruction set
- "Opcodes" are pointers to existing code
- Generalizes return-to-libc with more programmability
- Academic introduction and source of name: Hovav Shacham, ACM CCS 2007

ret2pop (Nergal, 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

Gadgets

- Basic code unit in ROP
- Any existing instruction sequence that ends in a return
- Found by (possibly automated) search

Another partial example

```
... 0f b6 56 1c ff 40 08 c6 ...
```

Where gadgets come from

- Possibilities:
  - Entirely intended instructions
  - Entirely unaligned bytes
  - Fall through from unaligned to intended
  - Standard x86 return is only one byte, 0xc3

Overlapping x86 instructions

```
push %esi
mov $0x56,%dh sbb $0xff,%al inc %eax or %al,%dh
movzbl 0x1c(%esi),%edx incl 0x8(%eax) ...
```

- Variable length instructions can start at any byte
- Usually only one intended stream
Building instructions

- String together gadgets into manageable units of functionality
- Examples:
  - Loads and stores
  - Arithmetic
  - Unconditional jumps
- Must work around limitations of available gadgets

Hardest case: conditional branch

- Existing jCC instructions not useful
- But carry flag CF is useful
- Three steps:
  1. Do operation that sets CF
  2. Transfer CF to general-purpose register
  3. Add variable amount to %esp

Further advances in ROP

- Can also use other indirect jumps, overlapping not required
- Automation in gadget finding and compilers
- In practice: minimal ROP code to allow transfer to other shellcode

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

Setup

- Key motivation for ROP is to disable W ⊕ 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

- Threat modeling: printer manager (cont’d)
- Return-oriented programming (ROP)
- Announcements intermission
- ROP shellcoding exercise
- Testing and fuzzing
**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