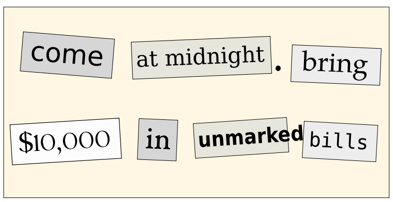
CSci 5271
Introduction to Computer Security
Day 6: Low-level defenses and counterattacks, part 2

Stephen McCamant
University of Minnesota, Computer Science & Engineering

Outline

Pop culture analogy: ransom note trope



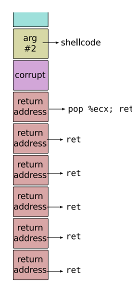
Basic new idea

- Treat the stack like a new instruction set
- "Opcodes" are pointers to existing code
- Generalizes return-to-libc with more programmability

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

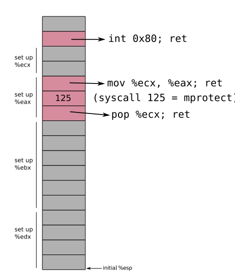
ret2pop (Müller)



Gadgets

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

Another partial example



Overlapping x86 instructions

```
        push %esi
    mov $0x56,%dh | sbb $0xff,%al | inc %eax | or %al,%dh
movzbl 0x1c(%esi),%edx | incl 0x8(%eax) | ...
0f b6   56   1c   ff   40   08   c6
```

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

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

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
- 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

Anti-ROP: lightweight

- Check stack sanity in critical functions
- Check hardware-maintained log of recent indirect jumps (kBouncer)
- Unfortunately, exploitable gaps

Gaps in lightweight anti-ROP

- Hide / flush jump history
- Very long loop → context switch
- Long "non-gadget" fragment
- (Later: call-preceded gadgets)

Anti-ROP: still research

- Modify binary to break gadgets
- Fine-grained code randomization
- Beware of adaptive attackers ("JIT-ROP")
- Next up: control-flow integrity

Outline

- Return-oriented programming (ROP)
- **Announcements**
- BCECHO
- Control-flow integrity (CFI)
- Additional modern exploit techniques

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
- In particular, the BCMTA vulnerability announcement is embargoed

Outline

- Return-oriented programming (ROP)
- Announcements
- **BCECHO**
- Control-flow integrity (CFI)
- Additional modern exploit techniques

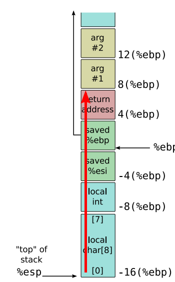
BCECHO code

```
void print_arg(char *str) {
  char buf[20]; int len;
  int buf_sz = (sizeof(buf)-sizeof(NULL))
               * sizeof(char *);
  len = strlcpy(buf, str, buf_sz);
  if (len > buf_sz) {
    fprintf(stderr,"Trucation occured "
            "when printing %s\n", str);
  }
  fwrite(buf, sizeof(char), len, stdout);
}
```

Attack planning

- Looks like candidate for classic stack-smash
- Where to put the attack value?
  - Via disassembly inspection
  - Via GDB
  - Via experimentation

Overwriting the return address



Shellcode concept

```
fd = open("/etc/passwd",
          O_WRONLY|O_APPEND);
write(fd, "pwned\n", 6);
```

Outline

- Return-oriented programming (ROP)
- Announcements
- BCECHO
- **Control-flow integrity (CFI)**
- Additional modern exploit techniques

Some philosophy

- Remember allow-list vs. deny-list?
- 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

How to support COTS binaries

- "Commercial off-the-shelf" binaries
- CCFIR (Berkeley+PKU, Oakland'13)
  - Use Windows ASLR info. to find targets
- CFI for COTS Binaries (Stony Brook, USENIX'13)
  - Keep copy of original code, build translation table

Control-Flow Guard

- CFI-style defense now available in Windows
- 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-oriented programming (ROP)

Announcements

BCECHO

Control-flow integrity (CFI)

Additional 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

Next time

- Defensive design and programming
- Make your code less vulnerable the first time