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: \( \neg (W \land X) \)

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: coming up next

Outline

W X (DEP)
Announcements intermission
Return-oriented programming (ROP)
Control-flow integrity (CFI)
More modern exploit techniques
HW1 VMs available

- Hosted across CSE Labs cluster
- Send list of members to geddes@cs.umn.edu
- Include name(s) and UMN id(s)/login name(s)

HW1 early submission

- Due 11:55pm this Friday, .tar.gz on Moodle
- Outline three of your attacks, including specific vulnerabilities
- Primarily for your benefit: take advantage!

Reminder: exercise set 1

- Due 11:55pm this Thursday
- Up to groups of 3
- Acknowledge sources
- Submit plain-text or PDF via Moodle

Scheduling first project meetings

- Plan to start sending invitations tonight for meetings starting this Wednesday
- I’ve requested more times from three groups

Outline

- W X (DEP)
- Announcements intermission
- Return-oriented programming (ROP)
- Control-flow integrity (CFI)
- More modern exploit techniques

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

**Gadgets**

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

**Overlapping x86 instructions**

- 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)
- In theory, these have gaps
  - Let’s see what happens once they’re deployed

Anti-ROP: still research

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

Outline

W+X (DEP)
Announcements intermission
Return-oriented programming (ROP)
Control-flow integrity (CFI)
More modern exploit techniques

Some philosophy

- Remember whitelist vs. blacklist?
- 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

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

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

Approximating the call graph: CCFIR

- One set: all legal indirect targets
- Two sets: indirect calls and return points
- Three sets: segregate returns into sensitive functions
- $n$ sets: needs possibly-difficult points-to analysis

Outline

- W+X (DEP)
- Announcements intermission
- Return-oriented programming (ROP)
- 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+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
```

**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)

**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

**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
Next time

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