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
Return address protections
ASLR and counterattacks
W_X (DEP)
Announcements
Return-oriented programming (ROP)
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 → stack unprotected, etc.

GOT hijack (Müller)

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

GOT hijack (Müller)

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

ret2pop (Müller)

Outline

- Return address protections
- ASLR and counterattacks
- W(X) (DEP)
- Announcements
- Return-oriented programming (ROP)
- Control-flow integrity (CFI)
- More modern exploit techniques
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: next

Outline

- Return address protections
- ASLR and counterattacks
- W[X] (DEP)

Announcements

- Return-oriented programming (ROP)
- Control-flow integrity (CFI)
- More 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

First project meetings

- Sent invitations yesterday, for meetings through next Monday
- Will see most of you later this week
- First progress reports due Monday 2/25
Exercise set 1

- Due tomorrow by 11:59pm
- One member of each group should submit PDF or plain text via Canvas

Outline

- Return address protections
- ASLR and counterattacks
- \( W_\Xi X \) (DEP)
- Announcements
- 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
Another partial example

```
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
```

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)
- Unfortunately, exploitable gaps

**Gaps in lightweight anti-ROP**

- Three papers presented at 2014’s USENIX Security
- 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 address protections
- ASLR and counterattacks
- W(X) (DEP)
- Announcements
- 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

Target checking: classic

\[
\begin{align*}
&\text{cmp } [\text{ecx}], 12345678h \\
&\text{jne error_label} \\
&\text{lea ecx, } [\text{ecx+4}] \\
&\text{jmp ecx}
\end{align*}
\]

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

Supporting COTS programs

- 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
- ASLR and counterattacks
- W\textsuperscript{\textvisiblespace}\textsubscript{\textvisiblespace}X (DEP)
- Announcements
- 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<sup>ε</sup>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
```

```
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)

Sandbox 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