Outline

Classic code injection attacks
Announcements intermission
Shellcode techniques
Exploiting other vulnerabilities

Overwriting the return address

- Stop the program from crashing early
- ‘Overwrite’ with same value, or another legal one
- Minimize time between overwrite and use

Collateral damage

- Frames of effect
- Change a data pointer used to access a code pointer
- Easiest if there are few other uses
- Common examples
- Frame pointer
- C++ object vtable pointer

Other code injection targets

- Function pointers
- Local, global, on heap
- longjmp buffers
- GOT (PLT) / import tables
- Exception handlers

Indirect overwrites

Non-sequential writes

- E.g. missing bounds check, corrupted pointer
- Can be more flexible and targeted
- E.g. a write-what-where primitive
- More likely needs an absolute location
- May have less control of value written
**Unexpected-size writes**

- Attacks don't need to obey normal conventions
- Overwrite one byte within a pointer
- Use mis-aligned word writes to isolate a byte

**Outline**

- Classic code injection attacks
- Announcements intermission
- Shellcode techniques
- Exploiting other vulnerabilities

**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

**Outline**

- Classic code injection attacks
- Announcements intermission
- Shellcode techniques
- Exploiting other vulnerabilities

**Basic definition**

- Shellcode: attacker supplied instructions implementing malicious functionality
- Name comes from example of starting a shell
- Often requires attention to machine-language encoding

**Classic execve /bin/sh**

- `execve(fname, argv, envp)` system call
- Specialized syscall calling conventions
- Omit unneeded arguments
- Doable in under 25 bytes for Linux/x86

**Avoiding zero bytes**

- Common requirement for shellcode in C string
- Analogy: broken 0 key on keyboard
- May occur in other parts of encoding as well

**More restrictions**

- No newlines
- Only printable characters
- Only alphanumeric characters
- "English Shellcode" (CCS'09)
Transformations
- Fold case, escapes, Latin1 to Unicode, etc.
- Invariant: unchanged by transformation
- Pre-image: becomes shellcode only after transformation

Multi-stage approach
- Initially executable portion unpacks rest from another format
- Improves efficiency in restricted environments
  - But self-modifying code has pitfalls

NOP sleds
- Goal: make the shellcode an easier target to hit
- Long sequence of no-op instructions, real shellcode at the end
  - x86: 0x90 0x90 0x90 0x90 0x90 ... shellcode

Where to put shellcode?
- In overflowed buffer, if big enough
- Anywhere else you can get it
  - Nice to have: predictable location
- Convenient choice of Unix local exploits:

Where to put shellcode?
Environment variables

Code reuse
- If can't get your own shellcode, use existing code
- Classic example: system implementation in C library
  - "Return to libc" attack
- More variations on this later

Outline
- Classic code injection attacks
- Announcements intermission
- Shellcode techniques
  - Exploiting other vulnerabilities

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

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
  - On Windows, interesting address start low
- Step one: add extra integer specifiers, dump stack
  - Already useful for information disclosure

Format string attack layout
-caller frame, other frames
  - spec arg #3
  - spec arg #1
  - format string, ptr
  - return address

Format string attack layout
- caller frame, other frames
  - spec arg #3
  - spec arg #1
  - format string, ptr
  - return address
<table>
<thead>
<tr>
<th>Format string attack: overwrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>%n</code> specifier: store number of chars written so far to pointer arg</td>
</tr>
<tr>
<td>• Advance format arg pointer to other attacker-controlled data</td>
</tr>
<tr>
<td>• Control number of chars written with padding</td>
</tr>
<tr>
<td>• On x86, use unaligned stores to create pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next time</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Defenses and counter-attacks</td>
</tr>
</tbody>
</table>