Off-by-one bugs

- `strlen` does not include the terminator
- Comparison with < vs. <=
- Length vs. last index
- `x++` vs. `++x`

Even more buffer/size mistakes

- Inconsistent code changes (use `sizeof`)
- Misuse of `sizeof` (e.g., on pointer)
- Bytes vs. wide chars (UCS-2) vs. multibyte chars (UTF-8)
- OS length limits (or lack thereof)

Other array problems

- Missing/wrong bounds check
  - One unsigned comparison suffices
  - Two signed comparisons needed
- Beware of clever loops
  - Premature optimization

Integer overflow

- Fixed size result ≠ math result
- Sum of two positive ints negative or less than addend
- Also multiplication, left shift, etc.
- Negation of most-negative value
- `(low + high)/2`

Integer overflow example

```c
int n = read_int();
obj *p = malloc(n * sizeof(obj));
for (i = 0; i < n; i++)
  p[i] = read_obj();
```
Signed and unsigned

- Unsigned gives more range for, e.g., `size_t`
- At machine level, many but not all operations are the same
- Most important difference: ordering
- In C, signed overflow is undefined behavior

Mixing integer sizes

- Complicated rules for implicit conversions
  - Also includes signed vs. unsigned
- Generally, convert before operation:
  - E.g., `1ULL << 63`
- Sign-extend vs. zero-extend
  - `char c = 0xff; (int)c`

Null pointers

- Vanilla null dereference is usually non-exploitable (just a DoS)
- But not if there could be an offset (e.g., field of struct)
- And not in the kernel if an untrusted user has allocated the zero page

Undefined behavior

- C standard "undefined behavior": anything could happen
- Can be unexpectedly bad for security
- Most common problem: compiler optimizes assuming undefined behavior cannot happen

Linux kernel example

```c
struct sock *sk = tun->sk;
// ...
if (!tun)
    return POLLERR;
// more uses of tun and sk
```

Format strings

- `printf` format strings are a little interpreter
- `printf(fmt)` with untrusted `fmt` lets the attacker program it
- Allows:
  - Dumping stack contents
  - Denial of service
  - Arbitrary memory modifications!

Outline

Where overflows come from, cont'd
- More problems
- Classic code injection attacks
- Announcements intermission
- Shellcode techniques
- Exploiting other vulnerabilities

Overwriting the return address
Collateral damage

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

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

Indirect overwrites
- 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

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
Where overflows come from, cont’d
More problems
Classic code injection attacks
Announcements intermission
Shellcode techniques
Exploiting other vulnerabilities

Project pre-proposal
- PDF submission via Canvas, one per group
- Due this Wednesday evening (11:59pm)
## Project meeting scheduling
- For pre-proposal due Wednesday night:
- Will pick a half-hour meeting slot, use for three different meetings
- List of about 65 slots on the web page
- Choose ordered list in pre-proposal, length inverse to popularity

## Exercise set 1
- Questions PDF available on website
- Due Wednesday, September 29th, on Gradescope
- Groups of 1-3, turn in one online submission

## Updated instructor office hours
- Mondays 1-2pm in 4-225E Keller (unchanged)
- Wednesdays 10:30-11:30am on Zoom (moved earlier)
- Or email for an appointment

## Outline
- Where overflows come from, cont'd
- More problems
- 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

- Where overflows come from, cont'd
- More problems
- 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
- Step one: add extra integer specifiers, dump stack
  - Already useful for information disclosure

Format string attack layout
<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>