CSci 5271
Introduction to Computer Security
Day 4: Low-level attacks

Stephen McCamant

University of Minnesota, Computer Science & Engineering

Outline

Where overflows come from (cont'd)

Non-buffer problems

Announcements intermission

Classic code injection attacks

Shellcode and other targets

Exploiting other vulnerabilities

Last time

- Unsafe/misused library functions
  - `strcpy`
  - `strcat`
  - `sprintf`
- Alternatives have their own problems

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

Outline

Integer overflow

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

Integer overflow example

```
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

```
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(msg)` with untrusted `msg` lets the attacker program it
- Allows:
  - Dumping stack contents
  - Denial of service
  - Arbitrary memory modifications!

Outline

Where overflows come from (cont'd)

Non-buffer problems

Announcements intermission

Classic code injection attacks

Shellcode and other targets

Exploiting other vulnerabilities

HW1 progress

- Makefile posted
- Watch Moodle forum for latest news
- VMs: coming soon
- Getting started without a VM

Pre-proposals due Wednesday

- One page: who, what, why, how, when
- On web site: links to papers
- On web site: possible meeting slots
- Submit on Moodle by 11:55pm

Office hours

- Mondays: Stephen 10-11am 4-225E
- Tuesdays: Stephen 2-3pm 4-225E
- Wednesdays: Mike 2:30-3:30pm 2-209
- Thursdays: John 10-11am 2-209
- Fridays: John 1-2pm 2-209

Grace Hopper in Minneapolis

- Celebration of Women in Computing
- October 2-5 in downtown Minneapolis
- CS&E+CSE providing support + t-shirt
- `http://women.cs.umn.edu/`

Outline

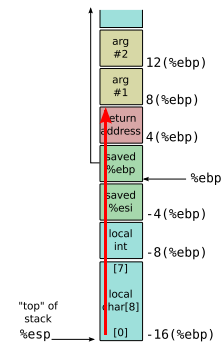Where overflows come from (cont'd)

Non-buffer problems

Announcements intermission
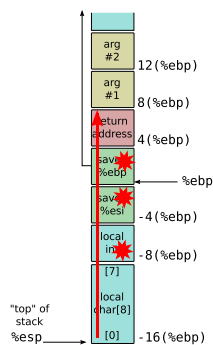
Classic code injection attacks

Shellcode and other targets

Exploiting other vulnerabilities

Overwriting the return address

Collateral damage

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
- 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)

Non-buffer problems

Announcements intermission

Classic code injection attacks

Shellcode and other targets

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
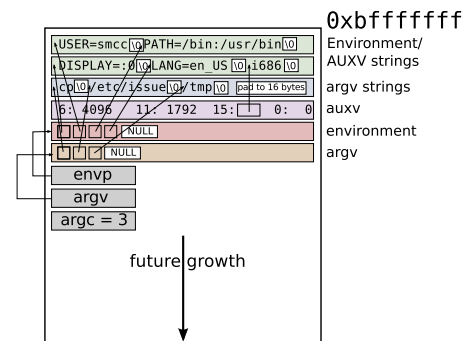- But self-modifying code has pitfalls

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


```
0xbfffffff
USER=smcc \0 PATH=/bin:/usr/bin \0    Environment/
DISPLAY=: \0 \0 LANG=en_US \0 i686 \0  AUXV strings
cp \0 /etc/issue \0 /tmp \0 pad to 16 bytes  argv strings
6:  4096   11  1792  15:     0:  0    auxv
                NULL                  environment
                NULL                  argv
        envp
        argv
        argc = 3

        future growth
```

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)

Non-buffer problems

Announcements intermission

Classic code injection attacks

Shellcode and other targets
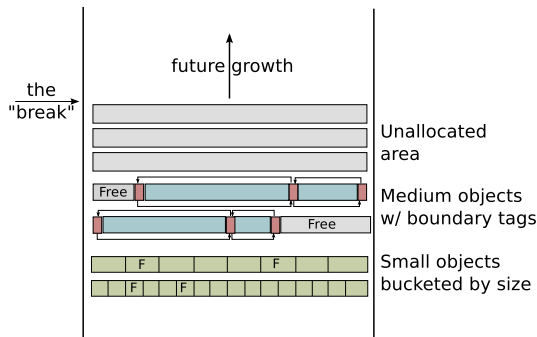
Exploiting other vulnerabilities

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

Heap meta-data

future growth

the "break"

Unallocated area

Free

Medium objects w/ boundary tags

Free

F | F

Small objects bucketed by size

F | F

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
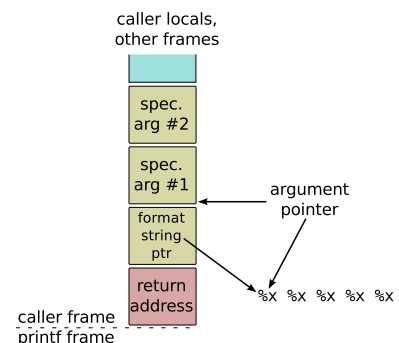 - Use math to figure out overflowing value

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
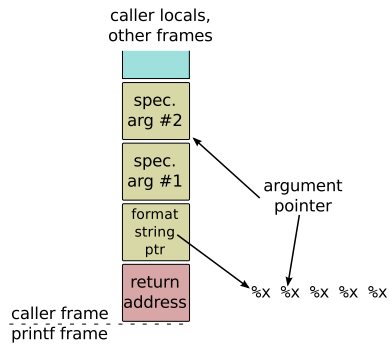 - Read more dangerous than a write

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

caller locals, other frames

spec. arg #2

spec. arg #1

format string ptr

argument pointer

return address

%X %X %X %X %X

caller frame
printf frame

Format string attack layout

caller locals,
other frames

spec.
arg #2

spec.
arg #1

argument
pointer

format
string
ptr

return
address

%X %X %X %X %X

caller frame
printf frame

Format string attack: overwrite

- %n specifier: store number of chars written so far to pointer arg
- Advance format arg pointer to other attacker-controlled data
- Control number of chars written with padding
- On x86, use unaligned stores to create pointer

Next time

- Defenses and counter-attacks