Simple buffer overflow
- Overflow one variable into another

gets(color)
- What if I type “blue 1”?
- (Actually, need to be more clever than this)

More devious examples...
- strcpy(buf, str)
- What if str has more than buf can hold?
  - Problem: strcpy does not check that str is shorter than buf
  - Overwrite EIP, change expected control flow of program (cf. in-class examples)

Even more devious...
- Attacker puts actual assembly instructions into his input string, e.g., binary code of execve("/bin/sh")

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address
**NOP sled**

- If exact address of injected shellcode is unknown, use NOP sled to provide a margin of error

**Severity of attack?**

- Theoretically, attacker can cause machine to execute arbitrary code with the permissions of the program itself

**Heap overflows**

- The examples just described all involved overflowing the stack
- Also possible to overflow buffers on the heap
- More difficult to get arbitrary code to execute, but imagine the effects of overwriting
  - Passwords
  - Usernames
  - Filenames
  - Variables
  - Function pointers (possible to execute arbitrary code)

**Defenses**
Defenses (overview)

- Prevent overflows from existing
  - Safe programming techniques/languages
  - Input validation
  - Static/dynamic analysis
- Prevent overflows from being exploited
  - Intercept function calls (e.g., Libsafe)
  - Canaries (StackGuard)
  - Non-executable stack, data execution prevention (DEP)
  - Address space layout randomization/ASLR

Safe programming techniques

- Use arrays instead of pointers
- Make buffers (slightly) longer than necessary to avoid “off-by-one” errors

Safe programming techniques

- We have seen that strcpy is unsafe
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until \"0\" is encountered, ignoring the size of buf
  - Avoid strcpy(), strcat(), gets(), etc.
    - Use strncpy(), strncat(), instead
    - Even these are not perfect... (e.g., no null termination)
    - Still need to be careful when copying multiple inputs into a buffer

Safe programming languages

- E.g., using Java prevents many of the problems that arise when using C
- C variants have been developed that ensure that certain classes of overflows cannot occur
  - E.g., Cyclone
Input validation

- (This is a general issue, not just in the context of buffer overflows)
- Two approaches
  - Whitelisting
    - Allow input that matches up with approved whitelist
  - Blacklisting
    - Discard input that matches up with disallowed blacklist
- The usual tradeoffs between usability/complexity and security

Length checking

- Beware off-by-one errors
- Even when using strncpy, the programmer must use the right value for the number of bytes to copy

```c
... strncpy(record, user, MAX_STRING_LEN-1);
strcat(record, ":");
strncat(record, fname, MAX_STRING_LEN-1);
```

ASCII printable instructions

- (Partial list)
  - AND EAX
  - SUB EAX
  - PUSH EAX
  - POP EAX
  - PUSH ESP
  - POP ESP

Input validation

- Blacklisting can often be easily circumvented
  - E.g., checking for a NOP sled
    - Replace NOP with a sequence of instructions whose net effect is to do nothing (inc eax, dec eax, ...)
- Whitelisting can be more difficult to bypass
  - E.g., checking for printable ASCII
    - There are legal instructions that fall in this range!
What can be done?

- AND EAX val1, AND EAX val2
  - If val1, val2 are ASCII-printable strings that are complements of each other, this zeros EAX
- SUB EAX v1, SUB EAX v2, SUB EAX v3
  - If v1, v2, v3 ASCII-printable strings chosen appropriately, wrap around occurs and the net effect is an addition
- PUSH EAX, POP ESP
  - Copies EAX into ESP

In principle, could build exploit using these instructions (very hard)

Alternative: inject loader code that generates shellcode on the stack

Defenses (overview)

- Prevent overflows from existing
  - Safe programming techniques/languages
  - Input validation
  - Static/dynamic analysis
- Prevent overflows from being exploited
  - Intercept function calls (e.g., Libsafe)
  - Canaries (StackGuard)
  - Non-executable stack, data execution prevention (DEP)
  - Address space layout randomization/ASLR

Development/compile time

compile time

O/S
Static analysis

- Static analysis: analyze source code (or object code) without actually running the program
- Can vary in complexity
  - flawfinder – checks code for instances of problematic functions (gets, strcpy, etc.)
  - Fortify
  - SPLINT – programmer annotates their code with pre-/post-conditions that are verified by the tool
    - E.g., buf[i]=x /*@requires i<max_size@*/
- (Provably) impossible to be perfect
  - False positives/negatives, with the usual tradeoffs...
    - Too many false negatives will make programmers less likely to use the tool

Dynamic analysis

- "Run" program (actually or symbolically) in an attempt to find security holes
  - Fuzzing can also be viewed as a type of dynamic analysis

Comparison?

- Static analysis may require annotation of source
- Dynamic analysis can be better, but slower

Libsafe

- Linked to code at compilation time
- Intercepts all calls to unsafe functions, and replaces them with safe versions
  - e.g., gets(dest) changed to fgets(dest, maxSize, stdin), where maxSize is computed from dest
- Drawbacks
  - Adds overhead
  - Only protects against a limited class of exploits
  - Requires linking to it at compile time
  - Only available for x86 processors
  - Incompatible with other techniques (e.g., StackGuard)

StackGuard

- Embed random "canaries" in stack frames and verify their integrity prior to function return
- Some possibilities:
  - NULL canaries
  - Random canaries
**Drawbacks of StackGuard**

- Adds overhead
- Only addresses stack overflows
- NULL canaries can potentially be overwritten
- Random canary values can potentially be learned (e.g., using format string vulnerabilities)

**Non-executable stack**

- A.K.A. data execution prevention (DEP)
  - Separate memory space for instructions from memory space allocated to the stack
- Drawbacks
  - Does not address general buffer/heap overflows, just execution of injected code from the stack
  - Does not address 'return-to-libc' exploits
    - Overflow sets ret-addr to address of libc function
    - E.g., `system("/bin/sh");`

**ASLR**

- Address-space layout randomization (ASLR)
- Idea: randomize the location of the stack
  - Prevents setting the EIP to point to the stack
- Countermeasures
  - Inject shellcode `below` the stored EIP; set EIP to point to a 'JMP ESP' instruction
  - Use huge NOP sled and try several times; stack randomization uses relatively little entropy
  - Use other vulnerabilities (e.g., format string vulnerabilities) to learn stack layout

**Further reading**

- *Hacking: The Art of Exploitation* 2nd Edition
  - Jon Erickson
More input validation flaws

Input validation

- Buffer overflows can be viewed as an example of problems caused by improper input validation
- There are many other important examples as well

Validating input

- Filenames
  - Disallow *, .., etc.
- Command-line arguments
  - Even argv[0]...
- Commands
  - E.g., SQL
- Integer values
  - Check for negative inputs
  - Check for large inputs that might cause overflow!
- Web attacks
  - E.g., cross site scripting, more
Input validation

- Buffer overflows can be viewed as an example of problems caused by improper input validation
- There are many other important examples as well

Need to validate input

- Filenames
  - Disallow *, /*/Alice/.../Bob, etc.
- Integer values
  - Check for negative inputs
  - Check for large inputs that might cause overflow!
- Command-line arguments
  - Even argv[0]...
- Commands
  - E.g., SQL
- Web attacks
  - E.g., cross site scripting, more

Format string vulnerabilities

- What is the difference between
  printf(buf);
  and
  printf("%s", buf);
  ?
- What if buf holds %x ?
- Look at memory, and what printf expects...

What happens?

- printf("%x") expects an additional argument...
  "%x"
   ebp buf
   eip
   Frame of the calling function
   "%x"
  will print the value sitting here

  - What if we could write that value instead?