Major Problem Areas for Secure Programming

- Certain areas of programming have proven to be particularly prone to problems
- What are they?
- How do you avoid falling into these traps?

Example Problem Areas

- Buffer overflows and other input verification issues
- Error handling
- Access control issues
- Race conditions
- Use of randomness
- Proper use of cryptography
- Trust
- Variable synchronization
- Variable initialization
- There are others . . .

Buffer Overflows

- The poster child of insecure programming
- One of the most commonly exploited types of programming error
- Technical details of how they occur discussed earlier
- Key problem is language does not check bounds of variables

Preventing Buffer Overflows

- Use a language with bounds checking
  - Most modern languages other than C and C++ (and assembler)
  - Not always a choice
  - Or the right choice
  - Not always entirely free of overflows
- Check bounds carefully yourself
- Avoid constructs that often cause trouble

Problematic Constructs for Buffer Overflows

- Most frequently C system calls:
  - `gets()`, `strcpy()`, `strcat()`, `sprintf()`, `scanf()`, `sscanf()`, `fscanf()`, `vfscanf()`, `vsprintf()`, `vscanf()`, `vsscanf()`, `streadd()`, `strecpy()`
  - There are others that are also risky

Why Are These Calls Risky?

- They copy data into a buffer
- Without checking if the length of the data copied is greater than the buffer
- Allowing overflow of that buffer
- Assumes attacker can put his own data into the buffer
  - Not always true
  - But why take the risk?

What Do You Do Instead?

- Many of the calls have variants that specify how much data is copied
  - If used properly, won't allow the buffer to overflow
- Those without the variants allow precision specifiers
  - Which limit the amount of data handled

Is That All I Have To Do?

- No
- These are automated buffer overflows
- You can easily write your own
- Must carefully check the amount of data you copy if you do
- And beware of integer overflow problems

An Example

- Actual bug in OpenSSH server:

```
u_int nresp;
. . .
nresp = packet_get_int();
If (nresp > 0) {
   response = xmalloc(nresp * sizeof(char *));
   for (i=0; i<nresp;i++)
       response[i] = packet_get_string(NULL);
}
packet_check_eom();
```

Why Is This a Problem?

- `nresp` is provided by the user
  - `nresp = packet_get_int();`
- But we allocate a buffer of `nresp` entries, right?
  - `response = xmalloc(nresp * sizeof(char *));`
- So how can that buffer overflow?
- Due to integer overflow

How Does That Work?

- The argument to `xmalloc()` is an unsigned int
- Its maximum value is 2^{32}-1
  - 4,294,967,295
- `sizeof(char *)` is 4
- What if the user sets `nresp` to 0x40000020?
- Multiplication is modulo 2^{32} . . .
  - So 4 * 0x40000020 is 0x80

What Is the Result?

- There are 128 entries in `response[]`
- And the loop iterates hundreds of millions of times
  - Copying data into the "proper place" in the buffer each time
- A massive buffer overflow

Other Programming Tools for Buffer Overflow Prevention

- Software scanning tools that look for buffer overflows
  - Of varying sophistication
- Use a C compiler that includes bounds checking
  - Typically offered as an option
- Use integrity-checking programs
  - Stackguard, Rational's Purity, etc.

Canary Values

- One method of detecting buffer overflows
- Akin to the "canary in the mine"
- Place random value at end of data structure
- If value is not there later, buffer overflow might have occurred
- Implemented in language or OS

Data Execution Prevention (DEP)

- Buffer overflows typically write executable code somewhere
- DEP prevents this
  - Page is either writable or executable
- So if overflow can write somewhere, can't execute the code
- Present in Windows, Mac OS, etc.
- Doesn't help against some advanced techniques

Randomizing Address Space (ASLR)

- Address Space Layout Randomization
- Randomly move around where things are stored
  - Base address, libraries, heaps, stack
- Making it hard for attacker to write working overflow code
- Used in Windows, Linux, MacOS
- Not always used, not totally effective
  - Several recent Windows problems from programs not using ASLR