Goals for Today

- Software security
  - Software lifecycle
  - Buffer overflow attacks
  - Other software security issues

- Turn in Ethics Form
- Project 1 online

- Really impressed with all the activity on the blog!

Software Lifecycle (Simplified)

- Requirements
- Design
- Implementation
- Testing
- Use

Software problems are ubiquitous
Software problems are ubiquitous

1957–1958 – Therac-25 medical accelerator. A radiation therapy device malfunctioned and delivered lethal radiation doses at several medical facilities. Based upon a previous design, the Therac-25 was an “improved” therapy system that could deliver two different types of radiation treatments. The failure was caused by a missing overcurrent trip that caused the patient to receive a lethal dose of radiation.

The Therac-25 X-rays were generated by shining high-power x-rays into a metal target positioned between the electron gun and the patient. A second “improvement” was the replacement of the older Therac-25’s electronic safety interlocks with software control. A decision made because software was perceived to be more reliable.

What engineers didn’t know was that both the software and the X-ray beam were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called “race condition” involving the timing of the system’s interrupts, the Therac-25’s electronic ‘check’ interlock system had the potential to not ‘fire’ on high-power x-rays, with the result being some patients were seriously injured.

NASA Mars Lander
• Bug in translation between English and metric units
• Cost taxpayers $165 million

Denver Airport baggage system
• Bug caused baggage carts to become out of “sync,” overloaded, etc.
• Delayed opening for 11 months, at $1 million per day

Other fatal or potentially fatal bugs
• US Vicennes tracking software
• MV-22 Osprey
• Medtronic Model 8870 Software Application Card

From Exploiting Software and http://www.fda.gov/cdrh/recalls/recall-082404b-pressrelease.html

Software problems are ubiquitous

January 15, 1990 – AT&T Network Outage. A bug in a new release of the software that controls AT&T’s 24-hour long distance switches causes these million-dollar computers to crash when they receive a specific message from one of their neighboring machines.

One day a switch in New York crashes and reboot, causing its neighboring switches to crash, then their neighbors’ neighbors, and on and on. This cascading crash switches AT&T’s nationwide long-distance network into a complete neighborhood in less than a few seconds.

The AT&T engineers had to roll back the previous software release.


Software problems are ubiquitous

MSN Mars Lander
• Bug in translation between English and metric units
• Cost taxpayers $165 million

Denver Airport baggage system
• Bug caused baggage carts to become out of “sync,” overloaded, etc.
• Delayed opening for 11 months, at $1 million per day

Other fatal or potentially fatal bugs
• US Vicennes tracking software
• MV-22 Osprey
• Medtronic Model 8870 Software Application Card

From Exploiting Software and http://www.fda.gov/cdrh/recalls/recall-082404b-pressrelease.html

Adversarial Failures

Software bugs are bad
• Consequences can be serious
• Even worse when an intelligent adversary wishes to exploit them!
• Intelligent adversaries: Force bugs into “worst possible” conditions/states
• Intelligent adversaries: Pick their targets

Buffer overflows bugs: Big class of bugs
• Normal conditions: Can sometimes cause systems to fail
• Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, …)
A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now an EECS professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
  - Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy)

Buffer Overflow These Days

- Very common cause of Internet attacks
  - In 1998, over 50% of advisories published by CERT (Computer Security Incident Report Team) were caused by buffer overflows
- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in 10 minutes

Attacks on Memory Buffers

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
    - If more data is stuffed into it, it spills into adjacent memory
    - If executable code is supplied as "data", victim's machine may be fooled into executing it — we'll see how
  - Code will self-propagate or give attacker control over machine
- First generation exploits: stack smashing
- Second gen: heaps, function pointers, off-by-one
- Third generation: format strings and heap management structures
### Stack Buffers

- Suppose Web server contains this function
  ```c
  void func(char *str) {
    char buf[126];
    ...
    strcpy(buf, str);
    ...
  }
  ```
- No bounds checking on strcpy()
- If str is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior

### Changing Flags

- Suppose Web server contains this function
  ```c
  void func(char *str) {
    int authenticated = 0;
    char buf[126];
    strcpy(buf, str);
    ...
  }
  ```
- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in in.fingerd

### Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return

<table>
<thead>
<tr>
<th>Type</th>
<th>Text region</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>Text region</td>
<td>Heap</td>
<td>Stack</td>
</tr>
<tr>
<td>Addr</td>
<td>0x00...0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bottom</td>
<td>Heap</td>
<td>Stack</td>
<td>0xFF...F</td>
</tr>
</tbody>
</table>

### Stack Buffers

- Suppose Web server contains this function
  ```c
  void func(char *str) {
    char buf[126];
    ...
    strcpy(buf, str);
    ...
  }
  ```
- When this function is invoked, a new frame with local variables is pushed onto the stack
What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...
  ```c
  void func(char *str) {
      char buf[126];
      strcpy(buf, str);
  }
  ```
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations

Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon
  ```c
  exec("/bin/sh");
  ```
- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but...
- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Problem: No Range Checking

- `strcpy` does not check input size
  - `strcpy(buf, str)` simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - `strncpy`, `strcat`, `gets`, `scanf`, `printf`
Does Range Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
  - Programmer has to supply the right value of n

Potential overflow in htpasswd.c (Apache 1.3):

- Published "fix":
  ```c
  strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strcat(record, cpw, MAX_STRING_LEN-1);
  ```

Does Range Checking Help?

- Published "fix" for Apache htpasswd overflow:
  ```c
  strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strcat(record, cpw, MAX_STRING_LEN-1);
  ```

Misuse of strncpy in htpasswd "Fix"

- Home-brewed range-checking string copy
  ```c
  void notSoSafeCopy(char *input) {
    char buffer[512];
    int i;
    for (i=0; i<512; i++)
      buffer[i] = input[i];
  }
  void main(int argc, char *argv[]) {
    if (argc==2)
      notSoSafeCopy(argv[1]);
  }
  ```

Off-By-One Overflow

- 1-byte overflow: can't change RET, but can change pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - RET for previous function will be read from buffer!

Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return
Heap Overflow

- Overflowing buffers on heap can change pointers that point to important data
  - Sometimes can also transfer execution to attack code
  - Can cause program to crash by forcing it to read from an invalid address (segmentation violation)
- Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write to a normally inaccessible file
  - For example, replace a filename pointer with a pointer into buffer location containing name of a system file
    – Instead of temporary file, write into AUTOEXEC.BAT

Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(...)

Format Strings in C

- Proper use of printf format string:

```
int foo=1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

- This will print foo = 1234 in decimal, 4D2 in hex

- Sloppy use of printf format string:

```
char buf[14]="Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

- If buffer contains format symbols starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer.

Viewing Memory

- %x format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";
printf(buf);
```

- Stack location pointed to by printf's internal stack pointer will be interpreted as an int. (What if crypto key, password, ...?)

- Or what about:

```
char buf[16]="Here is a string:  %s";
printf(buf);
```

- Stack location pointed to by printf's internal stack pointer will be interpreted as a pointer to a string.
Writing Stack with Format Strings

- `%n` format symbol tells `printf` to write the number of characters that have been printed
- Argument of `printf` is interpreted as destination address
- This writes 14 into `myVar` (“Overflow this!” has 14 characters)
- What if `printf` does not have an argument?
  - Stack location pointed to by `printf` internal stack pointer will be interpreted as address into which the number of characters will be written.

More Buffer Overflow Targets

- Heap management structures used by `malloc()`
- URL validation and canonicalization
  - If Web server stores URL in a buffer with overflow, then attacker can gain control by supplying malformed URL
  - Nimda worm propagated itself by utilizing buffer overflow in Microsoft’s Internet Information Server

- Some attacks don’t even need overflow
  - Naïve security checks may miss URLs that give attacker access to forbidden files
    - For example, http://victim.com/user/../../autoexec.bat may pass naïve check, but give access to system file
    - Defeat checking for “/” in URL by using hex representation: `%5c%00` or `%5c%25%5c%00`. 