Goals for Today

- Software security
- Buffer overflow attacks
- Other software security issues
- Practice thinking about the security issues affecting real systems

Software problems are ubiquitous

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
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, …)

Morris Worm and Buffer Overflow

- We’ll consider the Morris worm in more detail when talking about worms and viruses
- 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)

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

Sequence

- 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 (!)

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

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

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

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>Text region</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0x00_0</td>
<td>Addr 0xFF_F</td>
<td></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

```
Address 0xFF_F
Command /usr/bin/ld
```

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

```
Address 0xFF_F
Command /usr/bin/ld
```

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

```
Address 0xFF_F
Command /usr/bin/ld
```
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

<table>
<thead>
<tr>
<th>Saved SP</th>
<th>ret/IP</th>
<th>str</th>
<th>Caller's frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execute code at this address after func() finishes

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

<table>
<thead>
<tr>
<th>buf</th>
<th>Saved SP</th>
<th>ret/IP</th>
<th>str</th>
<th>Caller's frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>buf</th>
<th>Saved SP</th>
<th>ret/IP</th>
<th>str</th>
<th>Caller's frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>buf</th>
<th>Saved SP</th>
<th>ret/IP</th>
<th>str</th>
<th>Caller's frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When function exits, code in the buffer will be executed, giving attacker a shell

- Root shell if the victim program is setuid root
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
    - `strcpy(char *dest, const char *src)`
    - `strcat(char *dest, const char *src)`
    - `gets(char *s)`
    - `scanf(const char *format, ...)`
    - `printf(const char *format, ...)`

- **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):**
    - `strcpy(record, user);`
    - `strcat(record, ":");`
    - `strcat(record, cpw); ...`
  - **Published “fix” (do you see the problem?):**
    - `strcpy(record, MAX_STRING_LEN-1);`
    - `strcat(record, ":");`
    - `strcat(record, cpw, MAX_STRING_LEN-1); ...`

---

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

<table>
<thead>
<tr>
<th>buf</th>
<th>Saved SP</th>
<th>ret</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0xFF...F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- When function exits, code in the buffer will be executed, giving attacker a shell
  - **Root shell if the victim program is setuid root**

---

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

```
exec("/bin/sh")
```

<table>
<thead>
<tr>
<th>exec(&quot;/bin/sh&quot;)</th>
<th>Saved SP</th>
<th>ret</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0xFF...F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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
Misuse of strncpy in htpasswd “Fix”

- Published “fix” for Apache htpasswd overflow:
  ```c
  ... strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,:);
  strncpy(record,cpw,MAX_STRING_LEN-1);
  ```
  MAX_STRING_LEN bytes allocated for record buffer

Off-By-One Overflow

- 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]);
      }
  ```
  This will copy 513 characters into buffer. Oops!
Off-By-One Overflow

- 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];
  }
  ```

- 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!

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 into 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

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

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)(...)

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[13]="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
  ```
  printf("Overflow this\n",myVar);
  ```
  - 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?
  ```
  char buf[16]="Overflow this\n";
  printf(buf);
  ```
  - Stack location pointed to by printf's 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
  - Naive security checks may miss URLs that give attacker access to forbidden files
  - For example, http://victim.com/user/../../autoexec.bat may pass naive check, but give access to system file
  - Defeat checking for `/` in URL by using hex representation: `%5c` or `%255c`.

Non-Executable Stack

- NX bit on every Page Table Entry
  - AMD Athlon 64, Intel P4 "Prescott"
  - Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD

- Some applications need executable stack
  - For example, LISP interpreters

- Does not defend against `return-to-libc` exploits
  - Overwrite return address with the address of an existing library function (can still be harmful)

- ...nor against heap and function pointer overflows
- ...nor changing stack internal variables (auth flag, ...)

Preventing Buffer Overflow

- Use safe programming languages, e.g., Java
  - What about legacy C code?
- Mark stack as non-executable
- Randomize stack location or encrypt return address on stack by XORing with random string
  - Attacker won't know what address to use in his or her string
- Static analysis of source code to find overflows
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
- Black-box testing with long strings
Run-Time Checking: StackGuard

- Embed "canaries" in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

![Diagram showing buffer, saved SP, ret/IP, and caller's stack frame]

![Diagram showing buffer, 0000canary, saved SP, ret/IP, and caller's stack frame]

- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

Run-Time Checking: StackGuard

- Embed "canaries" in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

![Diagram showing buffer, saved SP, ret/IP, and caller's stack frame]

![Diagram showing buffer, 0000canary, saved SP, ret/IP, and caller's stack frame]

- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- PointGuard also places canaries next to function pointers and setjmp buffers
  - Worse performance penalty
- StackGuard can be defeated!
  - Phrack article by Bulba and Kil3r

Defeating StackGuard (Sketch)

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
  - strcpy will write into RET without touching canary!

![Diagram showing buffer, dst, canary, sfp, and RET]

Suppose program contains strcpy(dst,buf)
Defeating StackGuard (Sketch)

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
  - strcpy will write into RET without touching canary!

Suppose program contains strcpy(dst,buf)

```
buf  dst  canary  sfp  RET
```

Return execution to this address

```
canary  sfp  RET
```

Suppose program contains strcpy(dst,buf)

```
badpointer, attack code  canary  sfp  RET
```

Defeating StackGuard (Sketch)

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
  - strcpy will write into RET without touching canary!

Suppose program contains strcpy(dst,buf)

```
buf  dst  canary  sfp  RET
```

Return execution to this address

```
badpointer, attack code  canary  sfp  RET
```

Run-Time Checking: Libsafe

- Dynamically loaded library
- Intercepts calls to strcpy(dest,src)
  - Checks if there is sufficient space in current stack frame
    \[ \text{frame-pointer} - \text{dest} > \text{strlen(src)} \]
  - If yes, does strcpy; else terminates application

PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
  - Attacker cannot predict the target program's key
    - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address
**Normal Pointer Dereference** [Cowan]

1. Fetch pointer value
2. Access data referenced by pointer

**PointGuard Dereference** [Cowan]

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

---

**PointGuard Issues**

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt only pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwitten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG’d code doesn’t mix well with normal code
  - What if PG’d code needs to pass a pointer to OS kernel?

---

**Integer Overflow and Implicit Cast**

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

- If `len` is negative, may copy huge amounts of input into `buf`
  
  (from www-inst.eecs.berkeley.edu—impflaws.pdf)

---

**Integer Overflow and Implicit Cast**

```c
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

- What if `len` is large (e.g., `len` = 0xFFFFFFFF)?
- Then `len + 5` = 4 (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from www-inst.eecs.berkeley.edu—impflaws.pdf)
TOCTOU

- **TOCTOU == Time of Check to Time of Use**

  ```c
  int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
      return -1;
    if (!S_ISRREG(s.st_mode)) {
      error("only allowed to regular files!");
      return -1;
    }
    return open(path, O_RDONLY);
  }
  ```

- **Goal:** Open only regular files (not symlink, etc)
- **Attacker can change meaning of path between stat and open** (and access files he or she shouldn’t)

Randomness issues

- **Many applications (especially security ones) require randomness**
  - **“Obvious” uses:**
    - Generate secret cryptographic keys
    - Generate random initialization vectors for encryption
  - **Other “non-obvious” uses:**
    - Generate passwords for new users
    - Shuffle the order of votes (in an electronic voting machine)
    - Shuffle cards (for an online gambling site)

C’s rand() Function

- **C has a built-in random function: rand()**
  ```c
  unsigned long int next = 1;
  /* rand: return pseudo-random integer on 0..32767 */
  int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
  }
  /* srand: set seed for rand() */
  void srand(unsigned int seed) {
    next = seed;
  }
  ```

- **Problem:** don’t use rand() for security-critical applications!
  - Given a few sample outputs, you can predict subsequent ones

Problems in Practice

- **One institution used (something like) rand() to generate passwords for new users**
  - Given your password, you could predict the passwords of other users
- **Kerberos (1988 - 1996)**
  - Random number generator improperly seeded
  - Possible to trivially break into machines that rely upon Kerberos for authentication
- **Online gambling websites**
  - Random numbers to shuffle cards
  - Real money at stake
  - But what if poor choice of random numbers?

Big news... CNN, etc..

Obtaining Pseudorandom Numbers

- For security applications, want “cryptographically secure pseudorandom numbers”
- Libraries include:
  - OpenSSL
  - CryptoAPI (Microsoft)
- Linux:
  - /dev/random
  - /dev/urandom
- Internally:
  - Pool from multiple sources (interrupt timers, keyboard, ...)
  - Physical sources (radioactive decay, ...)

Security Analyses

- Recall
  - Assets: What you are protecting
  - Security Goals
    - Confidentiality
    - Integrity
    - Availability
  - Adversaries: Who might try to attack the system
  - Threats: What they might try to do
  - Potential Vulnerabilities: Possible weaknesses in system
  - Protection mechanisms: How to protect/deter attacks
- Last time: Voting machines

Your Turn

- Talk amongst your neighbors (2 to 3 people per group)
- Consider one (or two) of the following products:
  - [(Removed before posting online)]
  - Product of your choice
- Write-down (around 1 - 3 sentences for each)
  - Summary of product
  - 2 - 3 assets + security goals
  - 2 - 3 adversaries + threats
  - 2 - 3 potential weaknesses + protection mechanisms
- We’ll discuss in N minutes. Turn in papers at end of class for extra credit. (1 per group; names/student IDs at top.) I have paper.
Reading Assignment

♦ Chapter 11 of Stamp

♦ Read *Smashing the Stack for Fun and Profit* to understand details of overflow exploits
  ▪ Will *really* help with the project without it!
♦ Read *Exploiting Format String Vulnerabilities*
♦ Read *Blended Attacks* by Chien and Szor to better understand buffer overflows