Lecture 11:
Security & Assorted Fun Stuff
CSE 120: Principles of Operating Systems
Announcements

• Discussion section cancelled this week.
  – Matus will hold lab hours on Thursday, 7pm - ?

• Next lecture is review.
  – Bring questions.

• **Final Exam:** 3p-6p on Saturday, August 1

• If you are lost, please make an appointment for some Office Hour time.
Announcements

• Project 2 bonus points
  – Fix your bugs from Project 2
  – Earn ½ credit back for all the things you fixed.
  – We will test your Project 2 code at the milestone, too.
  – Points for corrections will be awarded at the Project 3 deadline.
Announcements

• Project 3 deadline extended to Monday 8/3 at midnight.
  – This is a HARD deadline. No slip days / extensions.

• Project 3 milestone tonight
  – Includes Project 2 feedback

• 2nd Project 3 milestone, Friday @ midnight
  – Also includes Project 2 feedback
What is not a characteristic of 2-level paging?

- extra memory access on a tlb miss
- saving memory space because you don't need a table for unused entries
- have a master page table that contains entry to a secondary page table
- allow any table to be swapped out into disk
PeerWise

- Directory entries map file names to:
  - Page table entries
  - Inodes
  - Cylinders
  - Sectors
  - Tracks
Review

• We’ve explained how file systems can be structured
  – Many techniques are similar to those in memory management
  – Unix-style: Inodes, data blocks, files, directories, etc…..

• Performance of file systems highly dependent on disk
  – Seeks take a long time
  – Placement of data matters (swiss-cheese problem and seek avoidance)

• Berkeley Fast File System (FFS)
  – Originally separated inodes from data blocks.
  – Moved to cylinder groups (group files that are likely to be accessed together)
  – Larger block sizes to increase throughput
File Buffer Cache

• Applications exhibit significant locality for reading and writing files
• Idea: Cache file blocks in memory to capture locality
  – This is called the file buffer cache
  – Cache is system wide, used and shared by all processes
  – Reading from the cache makes a disk perform like memory
  – Even a 4 MB cache can be very effective
• Issues
  – The file buffer cache competes with VM (tradeoff here)
  – Like VM, it has limited size
  – Need replacement algorithms again (usually LRU used)
Caching Writes

• Applications assume writes make it to disk
  – As a result, writes are often slow even with caching

• Several ways to compensate for this
  – “write-behind”
    • Maintain a queue of uncommitted blocks
    • Periodically flush the queue to disk
    • Unreliable
  – Non-volatile RAM (NVRAM)
    • As with write-behind, but maintain queue in NVRAM
    • Expensive
Read Ahead (Prefetching)

• Many file systems implement “read ahead”
  – FS predicts that the process will request next block
  – FS goes ahead and requests it from the disk...
  – ..while the process is computing on previous block!
  – When the process requests block, it will be in cache
  – Complements the disk cache, which also is doing read ahead

• For sequentially accessed files can make big difference
  – Unless blocks for the file are scattered across the disk
  – File systems try to prevent that, though (during allocating)

• Unfortunately, this doesn’t do anything for writes
  – What if we could make write-behind sequential as well?
Log-structured File System

• The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
  • 1) Disk bandwidth scaling significantly (40% a year)
    – Latency is not
  • 2) Large main memories in machines
    – Large buffer caches
    – Absorb large fraction of read requests
    – Can use for writes as well
    – Coalesce small writes into large writes
• LFS takes advantage of both of these to increase FS performance
  – Rosenblum and Ousterhout (Berkeley, ‘91)
LFS: Approach

Optimize for disk writes

- Batch writes in disk cache
  - Utilize increase in disk throughput
- Treat the disk as one big log for writes
  - No need to worry about special seeks or placement
- All data in file system appended to log
  - Data blocks, metadata, inodes, etc.
LFS: Example

Disk

inode

File 1
Write a file
Modify file
Write to file

File 2
Write to file
Modify file
...
LFS Challenges

• How do you locate data?
  – FFS places files in a particular location
  – LFS appends data to the end of the log

• How do you free data?
  – At some point, you can’t “append” anymore
  – How do you track and recover stale blocks in the log?
LFS: Locating Data

- FFS uses inodes to locate data blocks
  - Inodes pre-allocated in each cylinder group
  - Directories contain locations of inodes
- LFS appends inodes and data (basically everything) to end of the log
  - Makes them hard to find
- Approach
  - Use another level of indirection: Inode maps
  - Inode maps map file #s to inode location
  - Location of inode map blocks kept in checkpoint region
  - Checkpoint region has a fixed location
  - Cache inode maps in memory for performance
LFS: Example (inode maps)

 Aren’t reads still slow?
Rely on buffer cache to store inode maps.
Large buffer cache means don’t need to worry about reads!
RAID

• Problem:
  – Disk drives fail frequently
  – Disks are SLOW (seek times & transfer rates)

• Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  – Files are striped across disks
  – Each stripe portion is read/written in parallel
  – Bandwidth increases with more disks

• Redundant Array of Inexpensive Disks (RAID)
  – A storage system, not a file system
  – Patterson, Katz, and Gibson (Berkeley, ’88)
• Striping
  – File blocks spread across disks.
RAID 1

- Mirroring
  - File blocks duplicated across disks.
• **Floating Parity**
  – XOR data blocks to make check blocks.
RAID Levels

• RAID systems advertised as supporting different “RAID Levels”
• Here are some common levels:
  – RAID 0: Striping
    • Good for random access (no reliability)
  – RAID 1: Mirroring
    • Two disks, write data to both (expensive, 1X storage overhead)
  – RAID 5: Floating Parity
    • Parity blocks for different stripes written to different disks
    • No single parity disk, hence no bottleneck at that disk
    • Lower storage overhead
• Levels can be combined
  – Raid “1+0”: Striping plus mirroring
RAID Challenges

• Small files (small writes less than a full stripe)
  – Need to read entire stripe, update with small write, then write entire segment out to disks

• Reliability
  – More disks increases the chance of media failure (MTBF)

• Turn reliability problem into a feature
  – Use one disk to store parity data
    • XOR of all data blocks in stripe
  – Can recover any data block from all others + parity block
  – Hence “redundant” in name
  – Introduces overhead, but assuming disks are “inexpensive”
Security

• What is “Security”?
• Understanding the problem
  – Who are the “bad” guys?
  – What do they want?
  – How do they do it?
• Where / Why are we vulnerable?
  – How do we fix this?

• Thanks to Stefan Savage and David Aucsmith for their slides.
What is Security?

• Webster’s Dictionary: “Security”

• Freedom from Danger
• Freedom from fear or anxiety

• Measures taken to guard espionage or sabotage, crime, attack, or escape
  (2): an organization or department whose task is security
Why do we discuss Security with Operation Systems?

• OS is responsible for managing system resources.
  – Bad guys want to steal resources / use them for nefarious purposes.

• OS provides abstractions for user-level programs
  – If abstractions are not well designed / understood, an attacker can gain unauthorized access to resources

• What resources?
  – Your protected information (identity, credit card #s)
  – You compute / communication resources.
Competing Philosophies

• **Binary Model**
  – Traditional Cryptography
  – Assume adversary limitations X and define security policy Y.
  – If Y cannot be violated without needing X, then system is secure.

• **Risk Management Model**
  – Most commercial software development (and real-world security – e.g. terrorism)
  – Try to minimize the biggest risks / threats
  – Improve security where most cost effective
Problems with Security Models

• **Binary Model**
  – Many assumptions are brittle in real life.
  – Hard to know what security policy should be.
  – VERY expensive
    • How do you guard against everything?

• **Risk Management Model**
  – The bad guys only need to break in once.
  – How do we evaluate risk or reward?
    • Example: Credit card fraud
Problems with Security Models

• Risk Management creates an arms race
  – forced co-evolution
Understanding the Landscape

- **National Interest**
  - Spy
  - Author

- **Personal Gain**
  - Thief
  - Expert
  - Specialist

- **Personal Fame**
  - Vandal
  - Hobbyist

- **Curiosity**
  - Script-Kiddy

Borrowed from David Aucsmith
Understanding the Landscape

- National Interest
- Personal Gain
- Personal Fame
- Curiosity

- Script-Kiddy
- Hobbyist Hacker
- Expert
- Specialist

Borrowed from David Aucsmith
Understanding the Landscape

- **National Interest**
- **Personal Gain**
- **Personal Fame**
- **Curiosity**

- **Script-Kiddy**
- **Hobbyist Hacker**
- **Expert**
- **Specialist**

- **Spy**

Tools created by experts now used by less-skilled attackers and criminals

Borrowed from David Aucsmith
When is a program secure?

• What does that even mean?
When is a program secure?

• When it does exactly what it should?
  – Not more.
  – Not less.

• But how do we know what a program is supposed to do?
  – Somebody tells us? (But do we trust them?)
  – We write the code ourselves? (But what fraction of the software you use have you written?)
When is a program secure?

• 2nd try: A program is secure when it doesn’t do **bad things**

• Easier to specify a list of “bad” things:
  – Delete or corrupt important files
  – Crash my system
  – Send my password over the Internet
  – Send threatening e-mail to the my professor posing as me

• But… what if most of the time the program doesn’t do bad things, but occasionally it does? Or could? Is it secure?
When is a program secure?

• Claim: Perfect security rarely exists
  – Security vulnerabilities are the result of violating an assumption about the software (or, more generally the entire system)
  – Corollary: As long as you make assumptions, you’re vulnerable.
  – And: You always need to make assumptions! (or else your software is useless and slow)

• However, some assumptions are more dangerous than others
Software Vulnerabilities

• Software vulnerabilities
  – A bug in a software program that allows an unprivileged user capabilities that should be denied to them

• Most problematic: control hijacking
  – Bug that allows an input to be executed as code

• Buffer overruns are the quintessential example
Buffer overflows

• The #1 source of vulnerabilities in software
  – 50% of all CERT incidents related to these
• Caused because C and C++ are not safe languages
  – They use a “null” terminated string representation: “HELLO! \n”
  – Standard library routines assume that strings will have the null character at the end (vs Pascal)
  – Bad defaults: the library routines don’t check inputs
• Easy to accidentally get wrong
• ...even easier to maliciously attack
Buffer overflow attacks

• Assumption (by programmer) is that the data will fit in a limited buffer

• This leads to a vulnerability: Supply data that is too big for the buffer (thereby violating the assumptions)

• This vulnerability can be exploited to subvert the entire programming model
  — i.e. execute arbitrary code
How to hijack control of a system

• Get your code into victim’s address space (placement)
• Get their program to jump there (diversion)
• Lets look at stack overflows first...
Recap: Stack activations for C

Stack Grown Down

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs

Frame N-1

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs

Frame N

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs
Example

f() {
    g(parameter);
}

g(char *string) {
    char buf[16];
    strcpy(buf, string);
}
What this looks like
(Windows x86 cdecl call)

Prolog

push ebp  // save old frame pointer
mov ebp,esp // Set current frame pointer
sub esp,10h // reserve 16 bytes for buf
push ebx   //callee saves
push esi
push edi

... do stuff

Epilog

pop edi   // restore callee saves
pop esi
pop ebx
mov esp,ebp // unroll stack
pop ebp   //restore old frame pointer
ret 3     // pop eip and jmp to it

Caveat: no opt, no /GZ, no /GS
Problem: Strcpy is unsafe

• Basic problem is that the library routines look like this:

```c
void strcopy(char *src, char *dst) {
    int i = 0;
    while (src[i] != "\0") {
        dst[i] = src[i];
        i = i + 1;
    }
}
```

• If the memory allocated to dst is smaller than the memory needed to store the contents of src, a buffer overflow occurs

• Particularly problematic with c’s idiom of using local temporary buffers – allows “stack smashing” attack
Stack smashing in action

```c
f() {
    g(badstring);
}

g(char *string) {
    char buf[16];
    strcpy(buf,string);
}
```

- **Parameters**
- **Return Address**
- **Frame Pointer**
- **Locals**
- **Callee-save regs**

- Evil shellcode address
- 16 bytes

July 29, 2009
Aside: why is it called shellcode?

```assembly
xor eax, eax
push eax
push 0x68732f6e
push 0x69622f2f
mov ebx, esp
push eax
push eax
push ebx
mov al, 59
push eax
int 80h
```

```csharp
char shellcode[] = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
"\x69\x89\xe3\x50\x50\x53\x50\xb0\x3b\xcd\x80";
```

System Call (execve)

Shellcode courtesy Foster, Osipov, Bhalla and Heinen
Some issues

• What textual restrictions on attack string?
  — Can’t have a NULL

• How do you know where the return address is on the stack?
  — Deterministic relative to local buffer on stack frame

• How do you know what the precise address of your shellcode is?
  • Deterministic location: Trial and error (educated)
  • Semi-deterministic (or lazy): NOP sled’s to the rescue (0x90)
Lots of string-based buffer overflows

- Nov 2, 1988, finderd (morris worm)
  - Interestingly, not a control hijack
  - /usr/ucb/finger to /bin/sh

- July 17, 2000 Microsoft Outlook
- July 26, 2000 Adobe Acrobat
- May 18, 2000, Kerberos v4
- ...

July 29, 2009
Early thinking

• Problem is that the string functions don’t have range checking

• Use versions that do have range checking and all will be well
  – e.g., strncpy(char *dst, const char *src, size_t n)
  – No more than n characters copied from *src to *dst
  – Simple no?

• Not so simple in reality…
Problem: You have to use it right

- Vulnerability in htpasswd.c in Apache 1.3
  
  ```c
  strcpy(record, user);
  strcat(record, ":");
  strcat(record, cpw);
  ```

- “Solution”
  
  ```c
  strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strcat(record, cpw), MAX_STRING_LEN-1);
  ```

- Can write up to 2*(MAX_STRING_LEN-1) + 1 bytes!
Another problem... it's not just strings

- memcpy, bcopy
- Arrays
- Pointer arithmetic on local buffers
- etc...
What to do?

• Compile time approaches
  – Memory-safe languages
  – Testing tools

• Run-time approaches
  – Stack validation
  – Memory protection
  – Randomization
TOCTOU vulnerabilities
Time of check/Time of use

• Key issue: program makes assumptions about *atomicity* of actions

```c
f() {
    check_something();
    then_do_something();
}
```

Is the thing you checked still true?
TOCTOU example

- Scenario: root process wants to create a unique /tmp file
  - **Step 1:** choose a name
  - **Step 2:** check to see if it exists
  - **Step 3:** if it doesn’t exist, create it
**TOCTOU example**

- **Scenario:** root process wants to create a unique / tmp file
  - **Step 1:** choose a name
  - **Step 2:** check to see if it exists
  - **Step 3:** if it doesn’t exist, create it

- **Here’s the problem:**
  - Attacker interrupts between steps 2 and 3
TOCTOU example

• Scenario: root process wants to create a unique /tmp file
  
  **Step 1:** choose a name
  **Step 2:** check to see if it exists
  **Step 3:** if it doesn’t exist, create it

Here’s the problem:
  – Attacker interrupts between steps 2 and 3
  – Creates a link from expected /tmp file name to a major file, i.e. /etc/passwd
TOCTOU example

- Scenario: root process wants to create a unique /tmp file
  - **Step 1:** choose a name
  - **Step 2:** check to see if it exists
  - **Step 3:** if it doesn’t exist, create

Here’s the problem:
- Attacker interrupts between steps 2 and 3
- Creates a link from expected /tmp file name to a major file, i.e. /etc/passwd
- When program does the create, it stomps /etc/passwd with program’s authority
Quick background on suid

- Unix programs have access rights one of which can be “setuid” +s

- This means that the program can operate with the privileges of the *program*, not the *caller*

- Use by programs like passwd

- These programs need to be careful to check that the caller is allowed to do certain things
TOCTOU Example #2

• Code running with root/admin permissions
  
  /* access returns 0 on success */
  if(!access(file, W_OK)) {
      f = fopen(file, "wb+");
      write_to_file(f);
  } else {
      fprintf(stderr, "Permission denied trying to open %s\n", file);
  }

• Attack
  touch dummy; ln –s dummy pointer
  [call program with file=pointer; run again after access is called]
  rm pointer; ln –s /etc/passwd pointer
Is this realistic?

- You can’t control exactly when you run can you?

- Not usually, but you can try again and again and again... you only have to get it right once
What to do?

• Prevent or delay certain operations to the same filename in a given time (i.e. stat followed by link)

• Limit interleaving of operations to files from different processes (i.e. until one hasn’t touched file for x milliseconds)
Prevention tips

• Try to only operate on file descriptors, not names (immutable binding)
  – Especially avoid unlink()

• Don’t use access() to check privilege
  – Instead run program *unprivileged* and let file system do the checks for you
Vulnerabilities, threats and hindsight

• Bug or exploitable vulnerability?
  – No threat
  – Denial-of-service threat
  – Arbitrary remote code execution threat

• Lots of hot air expended on this topic
  – “Yes, you found a bug, but it’s not exploitable”
  – “This class of bugs is very hard to exploit”
  – “While the DoS threat is significant, this vulnerability can’t be used for code injection”

• Historically these distinctions have changed with experience
  – Case in point: the off-by-one stack overflow
  – Historically, not considered a major control hijacking threat
  – Today, considered easy
I’m in! ... Now what?

• Launch your payload code
  – Fork & exec a shell: exec( /bin/sh );
  – Send spam
  – Distributed Denial of Service (DDoS)
    • Blackmail
    • Political
    • Impress your friends / enemies
  – Spread to other machines
  – Steal passwords, key strokes, credit card numbers
I’m in! ... Now what?

• Root Kits - Cover your tracks
  – Manipulate system logs
  – Hide malicious activity (modify top, ps, tcpdump)
  – Hide malicious files (modify ls, stat)
  – Deactivate anti-virus
This is just the surface...

• If you’re into this stuff,
  – Take CSE 127… this is only the tip of the iceberg.
  – Read Kotler’s “Advanced Buffer Overflow Methods” for more shellcode hacks
    • E.g. using program literals as serendipitous instructions; jumping into middle of instructions, etc
  – Read Dowd et al’s “Art of Software Security Assessment” for more nasty C/C++ issues (they also update a blog with new ones)
  – Vulnerability research literature on-line is very good (see next slide)... very dedicated people… I no longer believe in unexploitable bugs
If you’re interested in vulnerabilities...

• The important vulnerability research literature is generally not from academia

• Two classics  
  – Smashing the Stack for Fun and Profit, AlephOne  
  – The Tao of Windows Buffer Overflows, DilDog

• To keep up to date  
  – Dave Aitel (Daily Dave mailing list)  
  – H.D. Moore (browserfun.blogspot.com & metasploit)  
  – Halvar Flake (ADD/XOR/ROL)  
  – Matasano blog (general)
Next Time

• Review for Final

• Check Web site for course announcements
  – http://www.cs.ucsd.edu/classes/su09/cse120