ECE 531 – Advanced Operating Systems
Lecture 36

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 December 2025

Announcements

- Project presentations Wednesday and Friday
  Final writeup due by December 19th (Friday)
- Remember to do course evaluations
- Remember: no more homeworks/tests. There is no test during finals week

Side note on Division

- Compiler issues with __aeabi_uldivmod
- ARM1176 has no hardware divide instruction
- If you divide by a constant the compiler can do this at compile time and it's not a problem (easiest is if it's a power of 2)
- Sometimes the compiler can't for some reason, or else you divide by a variable
- The compile will emit a call to a helper routine in libgcc.a
- Since we are bare metal we would have to implement

this ourselves, which we can do but the code I gave you doesn't

IPC – Inter-Process Communication

- Processes want to communicate with each other
- The OS prevents you from writing to another process' memory, so to do IPC the OS has to be involved
- Two major issues:
  - getting the message across
  - synchronizing

IPC Examples

- Window manager wants to tell all running programs it's about to logout
- OS wants to tell all running programs it is starting a shutdown
- You want to signal a process to tell it something has changed (window resized, want it to draw a progress bar, etc)
- You want to have processes work together on a problem (w/o using threads)

- You want to send the output of one program to the input of another

Linux IPC Methods

- There are nearly 20. Other OSes like windows also have a lot.
- Many are historical
- I took an OS class once where many weeks were spent talking about IPC for some reason

Linux IPC – Files

- Just write to a file, all can see it.
- What happens when multiple readers/writers?
- `fcntl()` syscall, with one of its many features being file locking
- For example: mail daemon writing mail to mailspool while your client is also trying to delete mail from the middle, what happens
- Things get exciting over network filesystems (NSF)

Linux IPC – Signals

- set up signal handler, sort of like a userspace interrupt.
- Can catch many things, such as segfault, control-C, control-Z (sleep), hangup
- A few user-allowed ones like USR1 you can send.
- Has many of the problems of interrupts (locking).
- Many functions are not signal safe.
- Crazy ways (setjmp) to exit signal handler without returning to where signal happened.
- Send with a kill() syscall (or kill/killall command line).

- What happens if system call interrupted?

Linux IPC – Pipes

- Anonymous – parent opens fd, forks child, child reads in from fd. One way communication (half-duplex).
- `ls -la | sort | uniq`
- Linux actually has a `pipe()` system call
  - uses a magic invisible pipe filesystem
  - Creates two fds, one in, one out.
  - Write to in, appears on out.
  - There is a maximum size before it blocks waiting for other side to consume (10k or so?).

- Shell command line pipes
    - Shell forks children
    - Over-writes stdout of one to be a fd
    - Over-writes stdin of other to be same fd
    - Can use `close()` and `dup()` syscalls to set this up
    - Can use the `popen()` c library call to do something similar from C.
    - TODO: lookup how vmwOS does this

Linux IPC – Named Pipes (FIFOs)

- mkfifo creates a file on disk that's a pipe.
- Multiple writers can write to this file and it is queued up and then a process can read it.
- Processes that aren't a child can access these
- Can open nonblocking
- Also get SIGPIPE if write to a closed pipe.

Linux IPC – Message Queues

- Sort of like a mailbox.
- Unlike pipes don't have to wait until other side connect (think phone-call vs text)

Linux IPC – SysV IPC

- Supports channeling (extra data that can be used to filter)
- Can read things out of order.
- Use `ipcs` to see.
- Use `ipcrm` to remove
- Use `msgset()` system call
- POSIX – supports priorities

Linux IPC – Shared Memory

- Make a region of memory common between two processes
  - SysV – part of SysV IPC. Historical. Use `shmget()` `syscall`
  - POSIX –
  - Anonymous `mmap()` memory – how Linux implements POSIX? Can do this with a file, or anonymous if want to be visible only in process.

Linux IPC – Sockets

- Regular network sockets – can open a regular network connection to localhost (see ECE435)
- UNIX Domain Sockets – fast. Like network sockets, but local so without going over the network. Live on the filesystem (usually under /tmp or /var/run) so can have file permissions. Can send file descriptors across.
- Netlink Sockets – designed for fast communication between userspace and kernel. Also allow for broadcast in user to user

Linux IPC – Sockets

- Inotify – can monitor filesystem and notice when someone else changes a file

Linux IPC – FUSE

- FUSE – used for implementing a filesystem in userspace, but can also be used to communicate

Linux IPC – Locks

- Locks can be thought of as a way of communicating between processes/threads
- SysV semaphores – Use `semget() syscall`
- POSIX semaphores
- FUTEX – kernel accelerated locking, used by pthreads() and such. You're not really supposed to use these manually.

Speeding up IPC

- Splice – move data from fd to pipe w/o a copy? VM magic?
- Sendfile. zero copy?

IPC in the news / DBUS

- kdbus – dbus into kernel to make it faster
  - Desktop Bus, D-Bus
  - allows communication on a desktop bus between apps and kernel
  - example – incoming skype call can notify system, and apps like the audio adjust and mp3 player can pause music and set up microphone
  - multicast
  - example – battery low notification, apps can listen,

save state, prepare to shut down.

- Kernel developers resist it
  - Most because who has to maintain it?
  - Also how well designed is it? can it be used by other tools?
  - We already have a lot of IPC in the kernel, can it be made generic?
  - It is faster, but what if user programs just bloat to negate this?

OS Security

- At the end of the notes there's some background if you haven't learned about computer security before
- Here we will talk about aspects of it tied to operating systems

Users/Passwords

- Multi-user systems — are they needed? How do they work?
- Traditionally each user had a user-id (uid) and group-id (gid)
- Access to things like files was based on if the file permissions (rwx, we talked about this with filesystems) and file ownership matched your id
- How big is UID? Was 16 bits (64k) eventually updated to 32-bits (which broke some things) as some large

universities ran into the limit

User Authentication

- /etc/passwd – hashed password, one-way hash
  can you crack a password? dictionary attack?
- /etc/shadow
- NSS, LDAP, PAM, NSS
- How enforced? Syscalls such as kill and such, check uid.
  Virtual memory usually cannot see any other processes.
  Our OS trickier, on ARM with large pages can have up to 16 different owner/protections but with us it has 1MB granularity.

Privilege Escalation

- If you can get kernel or super-user (root) code to jump to your code, then you can raise privileges and have a "root exploit"

- If a kernel has a buffer-overrun or other type of error and branches to code you control, all bets are off. You can have what is called "shell code" generate a root shell.

Setuid Programs

- Some binaries are setuid.
- They run with root privilege but drop them.
- If you can make them run your code before dropping privilege you can also have a root exploit.
- Tools such as ping (requires root to open raw socket), X11 (needs root to access graphics cards), web-server (needs root to open port 80).
- Ways to solve?
  - Careful coding

- Capabilities (fine grained permissions)
- Can you give an exec permission *only* to open port80?

  Problem is clever hackers are good at taking one capability and escalating to full.

File Creation Races

- Many programs open temporary files in /tmp and write to them
- If you can guess the name of a /tmp file a program will open, you can create a link or otherwise redirect it.
- Manage to create a link to /etc/shadow or similar

Security Through Obscurity

- KASLR randomization – knowing where parts of a program are can make exploits easier, so try to randomize where kernel and executables run
- Doesn't work well on 32-bit, only so many random places
- Also binaries released with distros are a bit predictable
- Makes debugging harder

Hardware Against You

- Side Channel Attacks
  - Accurate timing (perf, high-res timers, etc)
  - Meltdown
  - Spectre
- DRAM Rowhammer – bang on memory cells, can have nearby bits flip to 0. How can that be an issue

Hardware Help You

- VM/Pagetables – provide permissions
- Read/Write permissions to pages
- No-execute, seem obvious, came later (why?) No room in pages. Added with move to PAE or 64-bit PTEs often
- Newer support (ARM, etc) for kernel-noexec pages, i.e. to keep kernel from accidentally/on purpose jumping to and executing user code
- Can kernel just write to user provided addresses? Bad idea. Linux has copy_to_user and copy_from_user

that tries to sanitize addresses and inputs before using.

- Newer support for fast hardware bounds checking

ROP programming

- Just mark all code as execute only (no read/write)
- What if can chain together series of small code snippets ending in return, and push onto the stack, then return from it?
- How to fix that? Somehow enforce code entry points

Pi Secure Mode

- Hardware can provide security features
- Pi has separate secure mode. OS can run in unsecure mode with limitations, separate vectors/page tables
- Only secure code can update important registers
- A pain if you're trying to do fancy stuff like turn off caches from userspace

DRM

- Can you make truly "secure" machine only running code you want?
- First you have to trust the hardware and firmware
- Also need to trust the compiler (see: Reflections on Trusting Trust, Thompson 1984)
- Firmware only runs code signed with key
- Bootloader signed with key, as is operating system and any drivers
- In theory could also enforce signing of apps

- Only code officially blessed by central authority can run on your machine.
- Might be more secure, but it would spell end to general purpose computing
- Media companies like it (no stealing movies)
  Video game companies (no pirating)
- Where does it go wrong? What if key leaks?
  What if you have a buffer-overrun and hackers can get it to run unsigned code? [this is how jailbreaks for phones and consoles often work]

Sandboxing

- Try to run "safe" subset of instructions
- Google NaCL
- Containers
- Virtualization

Finding Bugs

- Source code inspection
- Watching mailing lists
- Static checkers (coverity, sparse)
- Dynamic checkers (Valgrind). Can be slow.
- Fuzzing

Reporting Linux Bugs

- All bugs are potentially security bugs
- Figuring out if any bug (and hundreds are fixed per week) is a security bug takes time
- False positives/negatives
- Having to manually mark them as such can help black hats find them easier

Fuzzing

- 1988. Barton Miller. Noticed unix programs crash due to line noise. General case OK, but die if you feed them out-of-range data.
- Idea is to check for potential crashes in programs by feeding them random inputs and see how they handle it.
- Trinity for Linux (by Dave Jones) is one current project
- perf_fuzzer (by me) is another

- Often the best way to get crashes isn't truly random inputs, but almost-correct inputs

perf_fuzzer

- Wrote my fuzzer in response to a perf_event bug found using trinity (I had contributed perf_event support to trinity).
- That bug was a 64-bit config value was only checked for validity with a 32-bit cast (so only bottom 32-bits). A user could then use the full 64-bit value to point to memory and increment values. The clever hacker managed to increment the IDT instruction vector table, point the undefined instruction interrupt to root shell

code, and then executed an undefined instruction.

- I wanted to see if perf_event (a research interest of mine) had any other vulnerabilities.

Fuzzing – Bugs I found

- ARM config too big – the config field was used as an offset into a (small) array without checking the size. Kernel Panic
- ARM dangling pointer – a small struct with function pointers was cast to a larger struct with function pointers, and one of the function pointers off the end was called. Kernel Panic

  Also, by luck on very specific 3.11-rc kernels the dangling pointer pointed to 0x80000000, which is user mappable.

Put code there to set uid to 0 and exec a shell and you have root. (Got a CVE, but getting this fixed took forever. Luckily it's very unlikely anyone was ever affected by this).

- Have found other, more boring, bugs. Usually just computer lockups.

Academic Fuzzing

- Just “regular” fuzzing is considered old-news, so despite being able to find new bugs all the time it’s hard to get academic funding or research for it
- Linux and hacker conferences are interested, although those are somehow not counted as exciting by other academics.

Worse is Better

- Worse is better / the right thing
- Richard Gabriel, 1989, made famous by JWZ sending it around
- New Jersey vs MIT
- UNIX/C vs LISP
- Is it better to spend lots of time coming up with a perfect specification on paper, then implement it?
- Is quicker/easier to understand better than complex and perfect?

- Should you come up with something “good enough” and let it grow naturally?
- Is it possible to ever design a perfect interface?
- Original example
  - PC loser-ing problem
  - Signal comes in during a system call. Right thing would be to somehow back everything out, return to userspace, and restart, all transparently to the user
  - UNIX example was to just cancel the syscall and return an error
  - This means that if you have bad luck on UNIX any

system call can fail. You should check each for EINTR

- Pretty much no-one does this
- Is it right to make the kernel simpler/easier to understand at the expense of userspace?
- Eventually "the right thing" was done, and you can add SA_RESTART to your syscall to automatically restart

General Computer Security

Computer Security

- Most effective security is being unconnected from the world and locked away in a box
- Modern systems are increasingly connected to networks, etc.
- Internet of Things

The Problem

- Untrusted inputs from user can be hostile.
- Users with physical access can bypass most software security.

What can an attacker gain?

- Fun / Mischief
- Profit
- A network of servers that can be used for illicit purposes (SPAM, Warez, DDOS)
- Spying on others (companies, governments, etc)

Sources of Attack

- Untrusted user input
  Web page forms
  Keyboard Input
- USB Keys (CD-ROMs)
  Autorun/Autostart on Windows
  Scatter usb keys around parking lot, helpful people plug into machine.
- Network

cellphone modems
Ethernet/internet
wireless/bluetooth

- Backdoors
  Debugging or Malicious, left in place

- Brute Force – trying all possible usernames/passwords

Types of Compromise

- DoS (Denial of Service)
  Fork bombs, etc.
- Crash (extreme form of DoS)
  “ping of death”
- User account compromise
- Root account compromise
- Privilege Escalation

- Rootkit
- Re-write firmware? VM? Above OS?

Trojan Horse and others

- Create program in home directory called “ls” that gives you root permission, then runs actual ls. Convince someone with root to cd to your dir and run ls. This is why “.” is not in the search path by default.
- Fake login screen, get password (sort of like phishing)
- Backdoors
- Virus that can infect executables. How can you detect this? Heuristics? Read-only-fs? offline list of checksums?

- Worm – famously the Morris worm in the 1980s

Unsanitized Inputs

- Using values from users directly can be a problem if passed directly to another process
- SQL injection attacks; escape characters can turn a command into two, letting user execute arbitrary SQL commands; xkcd `Robert '); DROP TABLE Students;--`
- If data (say from a web-form) directly passed to a UNIX shell script, then by including characters like ; can issue arbitrary commands

Buffer Overflows – Big Drawback of C

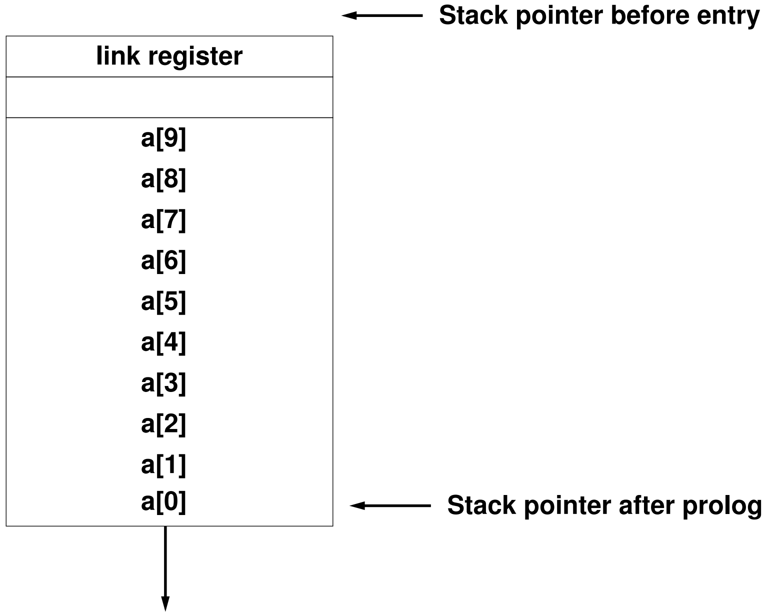
- User (accidentally or on purpose) copies too much data into a fixed sized buffer.
- Data outside expected area gets over-written. This can cause a crash (best case) or if user carefully constructs code, can lead to user taking over program.

Buffer Overflow Example

```
void function(int *values, int size) {
    int a[10];

    memcpy(a,values,size);

    return;
}
```

Maps to

```
    push    {lr}
    sub sp,#44

    memcpy

    add sp,#44
    pop {pc}
```



A value written to a[11] overwrites the saved link register. If you can put a pointer to a function of your choice there you can hijack the code execution, as it will be jumped to at function exit.

Mitigating Buffer Overflows

- Extra Bounds Checking / High-level Language (not C)
- Address Space Layout Randomization
- Putting lots of 0s in code (if strcpy is causing the problem)
- Running in a “sandbox”

Dangling Pointer / Null Pointer Dereference

- Typically a NULL pointer access generates a segfault
- If an un-initialized function pointer points there, and gets called, it will crash. But until recently Linux allowed users to `mmap()` code there, allowing exploits.
- Other dangling pointers (pointers to invalid addresses) can also cause problems. Both writes and executions can cause problems if the address pointed to can be mapped.

The Future of Operating Systems

- What is the future of Operating Systems?