LECTURE 1: intro

Random Stack Stuff

- Stack Pointer - points to top of stack
- Frame Pointer - points to the current active frame
- Link register holds the address to return to
- see written diagram for stack contents and how a buffer overflow happens

file descriptor - literally an integer that describes how to access a file on an OS, so you pass them around
unix permissions - read/write/exec file.
for dir, read = user can look at filenames inside dir
    write = user can add/del files
    exec = user can run ls or cd inside dir. can open files in ls
for sockets, the **user must have r/w permissions to connect to it**

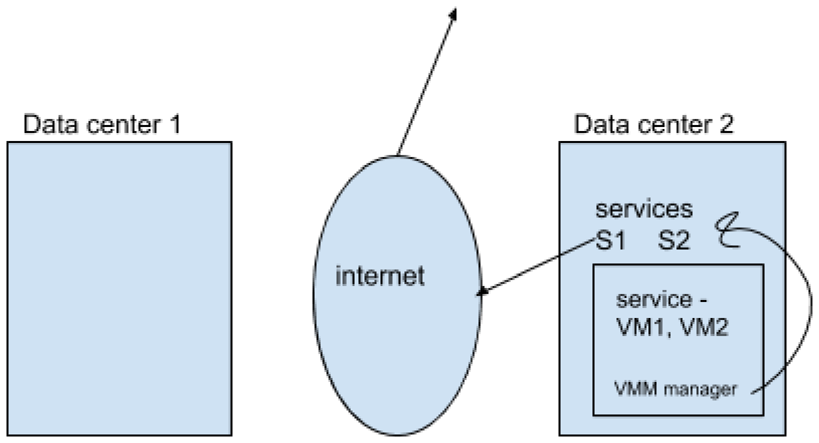
Security Plan

- **Goal** - what does your system want to do?
  - Example: Do you want it to only be readable to certain people or be available at some times?
- **Policy** - a plan/action to achieve the goal.
  - some permissions for readability or require a password
- **Threat model** - assumptions about what the attacker can do
  - guess passwords, but not steal a server
- **Mechanism** - software/hardware that system uses to support policy
  - password/encryption
  - maybe a human aspect (like don’t be stupid human mechanism)

Lecture 2- Security Architecture - defend against classes

- **Ask:**
  - **What are we defending? Credit card numbers? or Everything?**
  - **Who is/would be attacking us? Defines what they might do**
  - **Threat model? What are the assumptions of our attacker?**

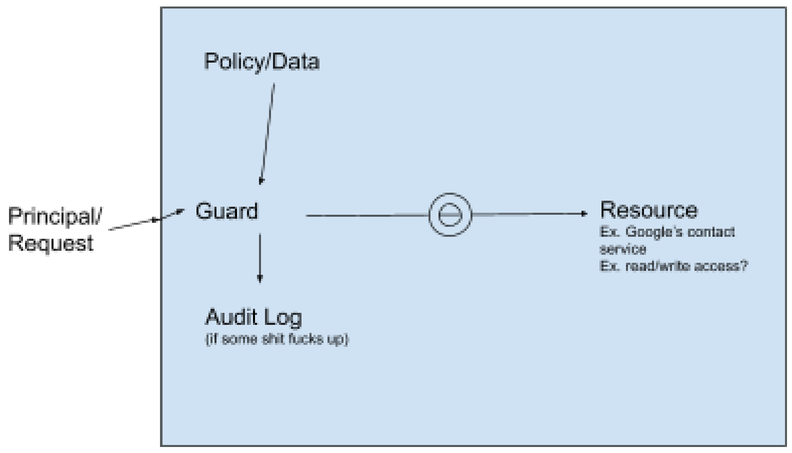
        - **EX: Can we assume they can break cryptographic schemes?**
- What kinds of attacks would Google be afraid of?
    - Bugs in Google's software
    - compromised network - attackers can listen in on network communication or traffic patterns or inject packets into network
    - stolen employee passwords
    - malware on phones/computers
    - insider attacks
    - malicious server hardware (since servers are from third parties)
    - old data discs



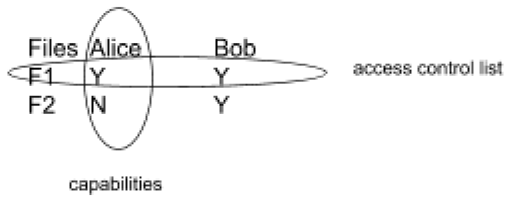
VM monitor looks at where incoming RPCs are coming from and seeing if its allowed (the guard w/ policy access control lists)

- <u>**Isolation**</u> - two things can run at the same time without affecting each other
    - like sometimes they run on the same Linux machine, but different user IDs
    - language sandbox
    - kernel
    - run VMs like above
    - dedicated services/servers
    - shortcomings
        - some VMs can talk to each other
            - side channels (changes hardware/cache)
            - maybe you can guess what's happening on other VMs o.o
    - **VMM -** keeps VM's away from each other

- **Reference Monitor - controlled sharing**

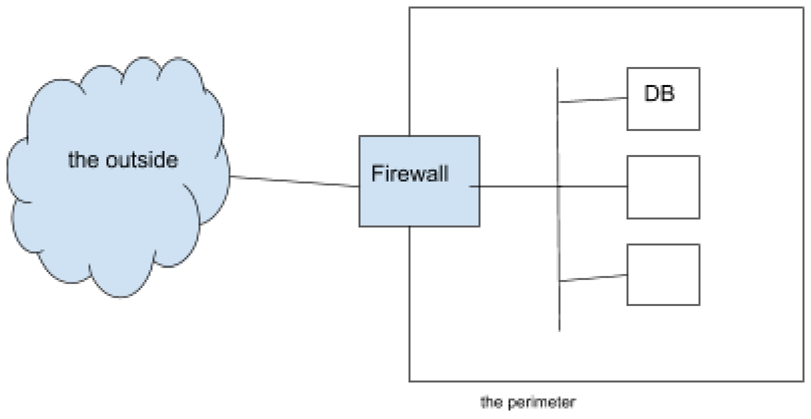
- 

- What the guard does:
    - Authenticate
        - “supply a password”
    - Authorize - policy function
        - What are certain users allowed to do?

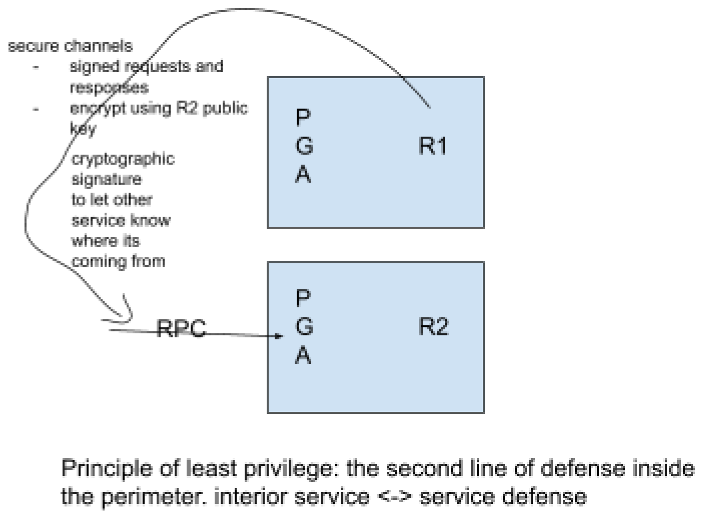


        - ACL (store by row)
        - Capabilities are like tokens or passing down data to children
        - Google example - administrator white list (ACL)
        - Google example - end-user permission ticket to access Contacts service (capability)
    - Audit
- Key: the resource is SEPARATED from the policy and the guard. doesn’t embed security checks in code!
- Cons: Well what if the system depends on the state of the resource? then might have to be a different design than the reference monitor design. also doesn’t do DoS prevention

- Perimeter Defense was what was used before.



-
    - weakness - once you get into the perimeter, everything is BAD
- instead of having a perimeter, google has many reference monitors


secure channels
- signed requests and responses
- encrypt using R2 public key
cryptographic signature to let other service know where its coming from
RPC
PGA
R1
PGA
R2
Principle of least privilege: the second line of defense inside the perimeter. interior service <-> service defense

    -
- CPU - instead of going directly b/w BIOS, it goes thru a security chip and then thru BIOS

- Dealing with DoS attacks
    - massive server-side load spreading
    - authenticate traffic as soon as possible, and only let legit users get priority

Lecture 3 - user auth

Reading -

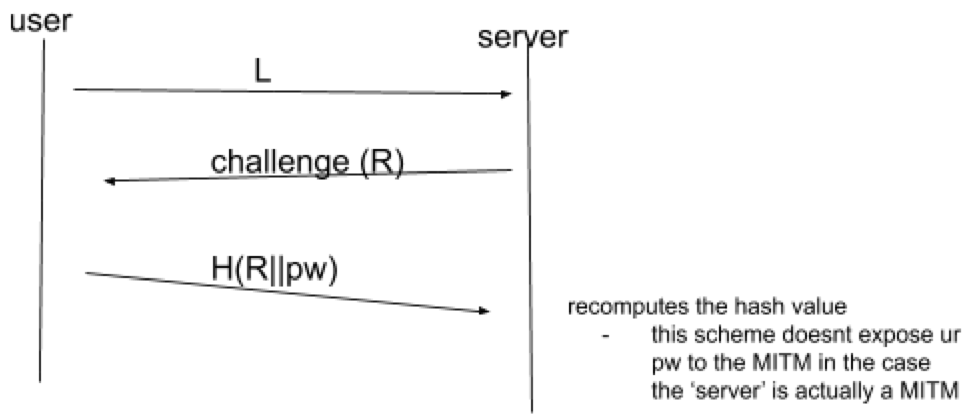
- requiring password changes may do more harm than good :O
    - many people just implemented transformations to their passwords

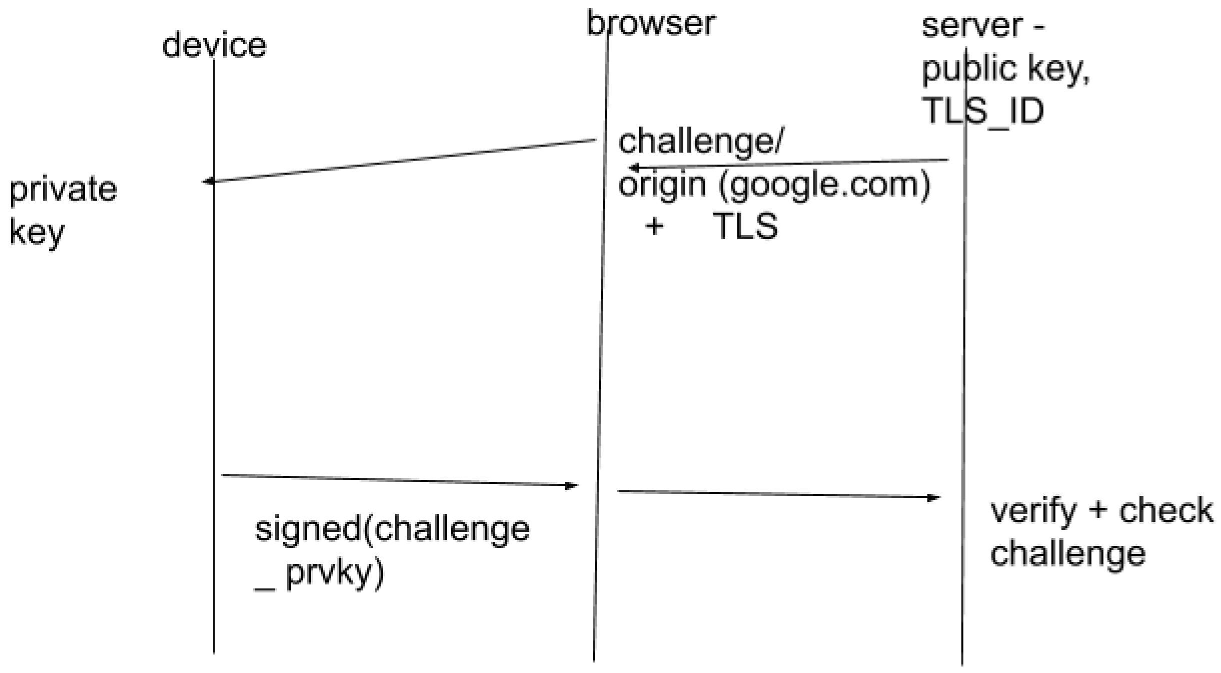
    - for 17% accounts, knowing a user’s previous password allowed them to guess their next password in fewer than 5 guesses
    - people choose weaker pws
- better methods
    - stronger length/complexity
    - slow hash functions
- FIDO U2F Protocol
    - UAF - Universal Authentication Framework Protocol
        - mechanisms like swiping finger, looking at camera, etc
    - U2F device - physical device like bluetooth LE devices or near field communication or USB
        - provides public key and key handle to website during registration
        - then, when user authenticates, website sends key handle back to U2F -> user private key and signature back to origin
    - ideally in future, any device will auto work with U2F device

Lecture 3 notes - User Authentication

- approach
    - user U registers with some S secret password
    - U uses secret to authenticate
- setting - after you register, what do the servers know about you?
    - MIT: physical person
    - Amazon: not so much...
- passwords:
    - pros
        - user friendly
    - but these are very valuable
- defenses
    - users use password managers
        - PW manager picks pw for u (high entropy)
        - unique per site
    - servers use password rules (how many char, $/%/& char), trying password rate limit
    - use pw as little as possible (authenticate once, and then youre good)
    - captchas? well the threat model of the assumption that computers aren’t good at image recognition isn’t correct, and hackers can just hire other users to crack them
- storing passwords
    - crypto hash function ST H(input) => short output ST it's hard to invert
    -

Users	pw	Salt
U1	H(salt || pw)	random #

  - many people might have the same pw, but different random salts
  - **rainbow tables - a dictionary of hashes of all common passwords**
  - **use expensive hash functions that take a long time**
- how to transmit the password?
  - Do we send it as cleartext over the network? no that's bad
  - encrypted connection? MITM
  - Challenge/response
  - 
user
server
L
challenge (R)
H(R||pw)
recomputes the hash value
- this scheme doesnt expose ur pw to the MITM in the case the 'server' is actually a MITM
- 2FA
  - SMS code
  - Duo/Google authentication
- U2F


device
browser
server - public key, TLS_ID
private key
challenge/ origin (google.com) + TLS
signed(challenge _ prvky)
verify + check challenge

if verified TLS, then no MITM

-
- origin = hash(protocol || hostname || port)
- prevents MITM from pretending to be the server (origin)
- prevents MITM from listening and intercepting info (TLS)
- 2 operations
    - sign(m,private_key)
    - verify(S, public_key)
- replay attack - you record the traffic and replay it later
    - but this won't work with above because the challenge basically acts as a timestamp

Lecture 4: Buffer Overflows

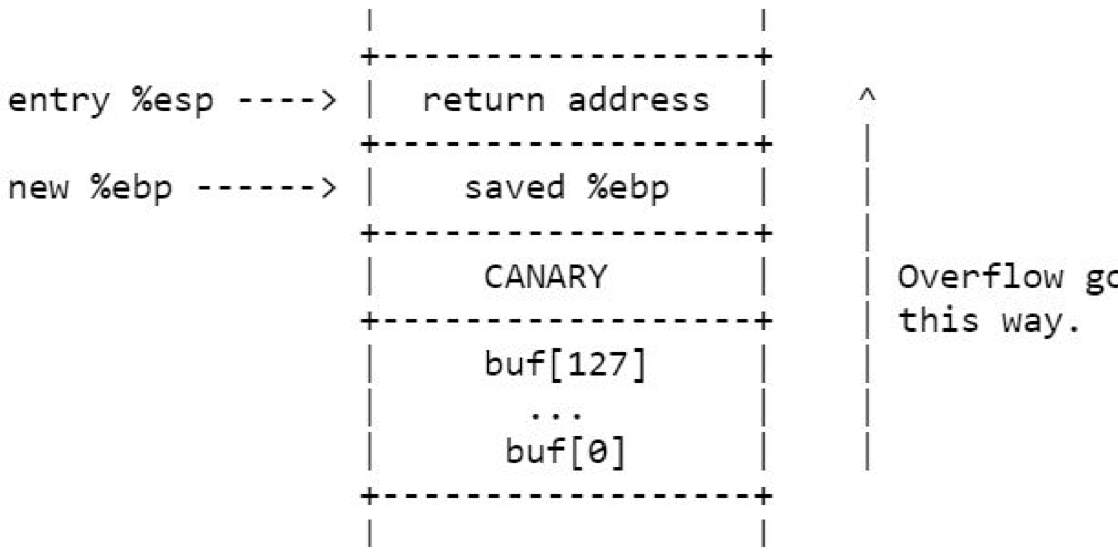
Lecture 4 Reading - Baggy Bounds Checking

- attacks that exploit out of bounds errors
- currently we just have a data struct w bounds for each alloc object (table lookup)
- now we restrain sizes of allocated memory regions for more efficient bounds lookups
- fat pointers - pointers w/ bounds info

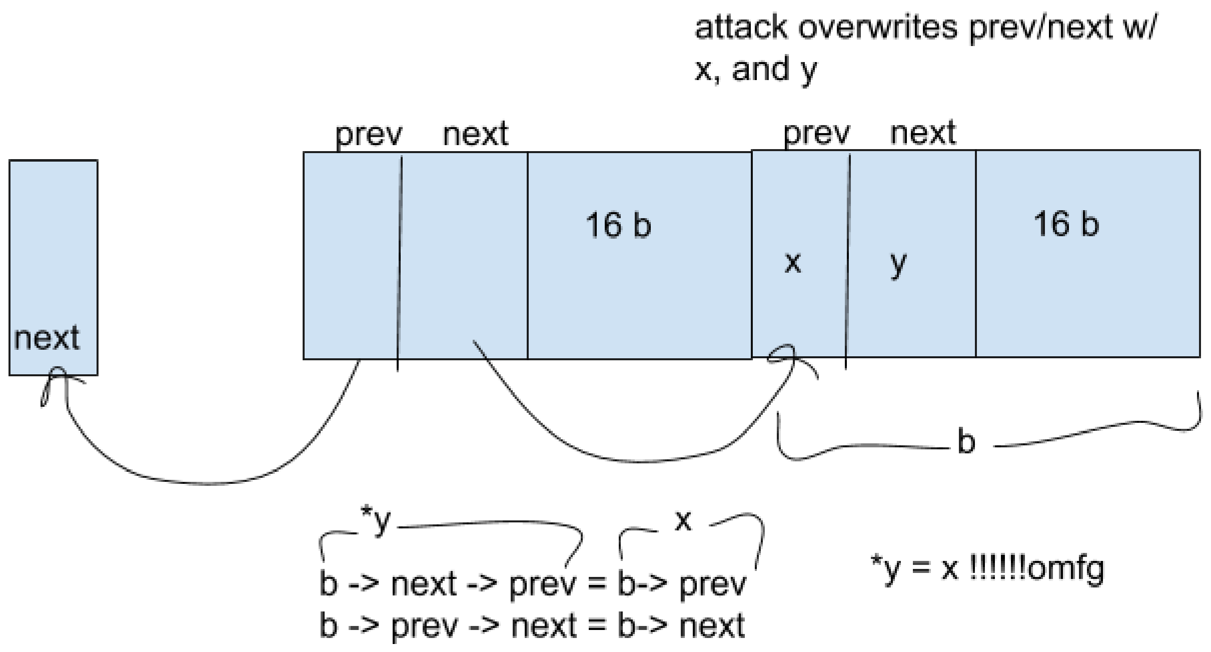
- baggy bounds checking - instead of precise object bounds, enforce allocation bounds (padding) - introduces benign out of bounds errors
    - - powers of two to store the binary log of the allocation size
- slot_size = 1 entry in the bounds table per slot.
- if p in allocation bounds
- - return element
- - if marked out of bounds
    - - if more than half a slot: error
    - - if less than a half a slot, mark out of bounds and return pointer
        - - any attempt to deref here will trigger exception

Lecture 4 Notes - Buffer Overflow Defenses

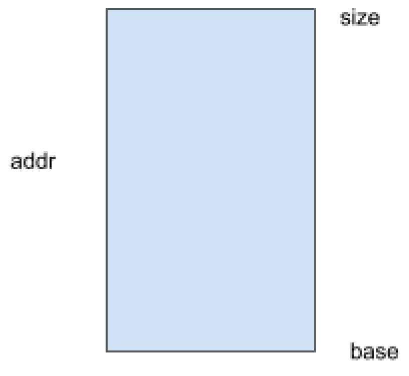
- Summary of attack situation
    - Buggy C code
    - write code into buffer
    - overwrite Rip (return instruction pointer)
- Defense idea!
    - NX bit (VM) - tells hardware not to execute on the stack
    - stack canaries



        -
        - random canary value - after something runs, check to see if canary is the same value as it was before to see if it was overwritten.
            - *but what if you can tell what the canary value is?*
    - other idea: ASLR (address space layout randomization) - put memory at random addresses
        - *But what if your random seed is v bad??*
- Heap overflow attacks - malloc
    - free list - ex. malloc(16)


attack overwrites prev/next w/ x, and y
prev
next
16 b
prev
next
x
y
16 b
next
b
*y
x
b -> next -> prev = b-> prev
b -> prev -> next = b-> next
*y = x !!!!!!omfg

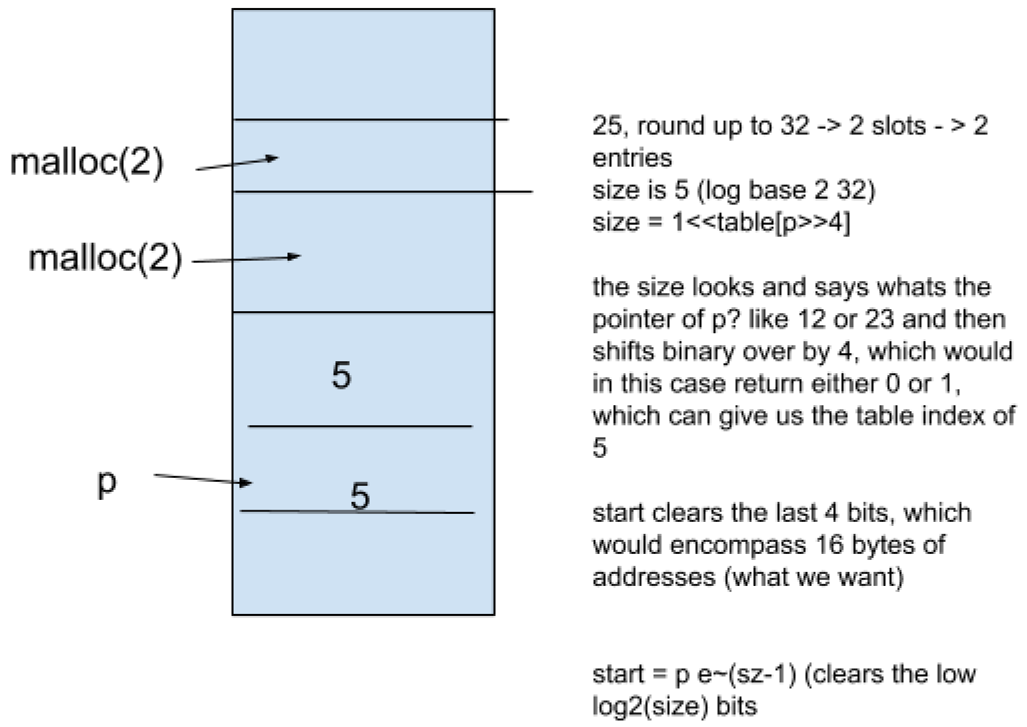
bottom stuff is just adding pointers for malloc
the thing that y points to is now equal to x (oh NO)
edit: ok so what's actually happening here is that when you call malloc you gotta first clear out the middle block (b), which shifts the pointers as so. but note that you can literally just *y = x something if an attacker puts x and y there beforehand

- defense approaches: bound checking
    - arithmetic ops q = p + 10
    - pointer deref *q = 1
    - **Approach #1 : Fat pointers**
        - pointer w/ addr, base, size
        - 

good ex in lec 4 online notes - addr could be something really wild but it'll fail the check
        - *but what happens if you try to combine this with a section of code that doesn't use fat pointers?*
    - **Approach #2: bounds table p->{base, size}**
        - we'll look at dereferencing and pointer arithmetic.

- **Approach #3: baggy bounds**
    - one entry covers 16 bytes
    - a slot = one table entry, and the table entry has just the size as log base 2
    - alloc only power of two sizes (you can figure out the start of an obj)!
    - use VM system to prevent OOB derefs - most significant bit marked to indicate OOB so it doesn't even try the deref
    - <1/2 slot size in case pointer moves back to slot_size

Baggy bounds example
p = malloc(25)



25, round up to 32 -> 2 slots - > 2 entries
size is 5 (log base 2 32)
size = 1<<table[p>>4]

the size looks and says whats the pointer of p? like 12 or 23 and then shifts binary over by 4, which would in this case return either 0 or 1, which can give us the table index of 5

start clears the last 4 bits, which would encompass 16 bytes of addresses (what we want)

start = p e~(sz-1) (clears the low log2(size) bits

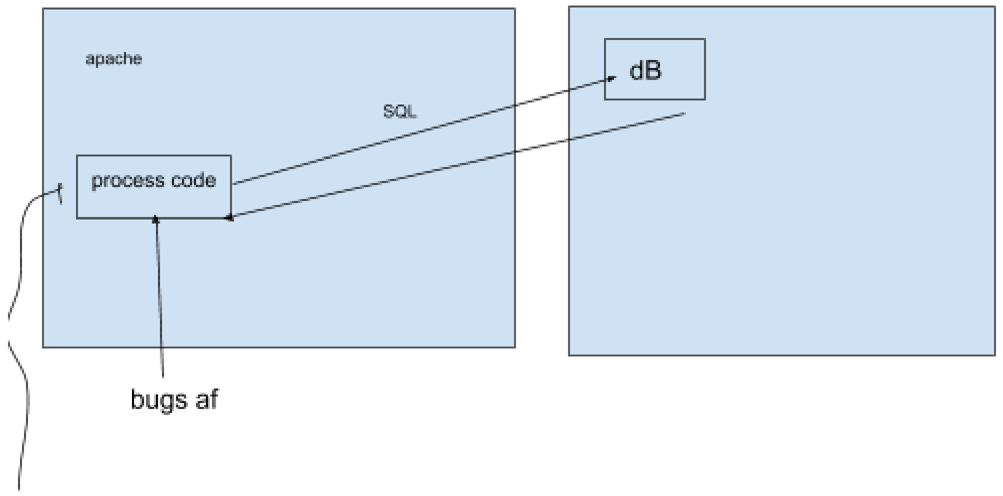
q = p+ 10   ---we're ok, we look up p in the bounds table, we see 5 (size 32, 10 < 32) we gucci
r = p + 35 -----principle OOB (35 !<32), but OOB, 3 < slot_size/2. so we set the OOB bit.
*r = 1 ------crash, but ok
s = p + 35 ------sets OOB bit
s = s-20 ------bc s was within slot_size/2, we can subtract a slot_size to identify the size (5) and then extract the start of the object to determine that 15 <32 is OK
t = p + 100 ------CRASH and its ok bc over slot_Size/2, so baggy bounds cant recover the pointer

LECTURE 5: privilege separation

Lecture 5 - Reading OKWS

- problem - one flaw can lead to several more throughout a system in a web server
- OK Web server - specialized for dynamic content, separation of privileges
- some bugs
    - unintended data disclosure
    - buffer overflows
    - DoS attacks
    - w/ PHP, it’s even worse bc dynamically generated content
- some bugs are inevitable - limit the effectiveness tho
    - server processes should be chrooted
    - server processes should run as unprivileged users
    - minimal set of database access privileges

Lecture 5 - Privilege Separation, Isolation



- traditional bug classes
    - code injection
    - missing access checks
    - open(“/p/” + user)
    - SQL injection
- plan B - how do we be secure despite bugs
    - IDEA - **privilege separation**: split code and data to limit damage
        - limit damage from successful attack - “least privilege”
        - limit direct access to buggy code - you can only attack the surface

- OKWS (refer to Fig 1)
    - OKLD - starts all processes
        - *compromise? all the privileges O.O but attack surface is small (no user input other than svc exit)*
    - OKLOGD - logger
        - *compromise? change/delete log entries, cover attacker's tracks. you can use log messages from okd, okld, svcs*
    - pubd - templates (can read files)
        - *compromise? some file system access, corrupting templates. attack surface includes requests to fetch templates from OKD*
    - OKD - accepts HTTP requests from browsers
        - *compromise? you can intercept responses/requests, and the attack surface would be parsing the first line of an HTTP request*
    - DBPROXY -> dB
        - restricts what queries each service can use
        - *compromise? access/change all data in the DB. attack surface is requests from authorized services (bug in service + bug in dbproxy)*
    - SVC (service processes) - each service runs as a separate process
        - ex. profile editors, login, messaging
        - this is where most bugs will be
        - ideally, the security person writes the other stuff, which will be safer
        - can't read/write files
        - *compromised? privileges include service's data, and its own requests to dbproxy, but attack surface is huge (HTTP requests)*
- dbproxy knows which user it is by a token. What if the token is disclosed?
    - what if an exploited SVC tries to read tokens from okld' config?
    - help from the OS!!!!!!
- Isolation and UNIX - what can a process manipulate?
    - processes - only if same UID can processes kill/debug each other
    - Files rwxr-xrr-x - only some processes can access files
    - processes cannot see or interfere with other FDs
    - IPC - inter server communication
    - TCP - servers can ignore, firewalls can block. but servers can't directly tell who a client is

Lecture - 2/25 - UNIX Security

UNIX principals - something that can have access privileges -- usually names of users/groups, or IDs (system identification)

- root - highly privileged (user ID 0)
    - sudo lets u temporarily be root

- nobody - highly unprivileged user
    - u could run like browsers or something as nobody
- sudo adduser name -u id#
- su name (switch user to)

permissions

- r, w, x (read write execute)
- user, group, or other
- make sure to give directories rwx
- chown - change owner
- chmod - change permission bits
- chgroup - change group assoc w file

real/effective/saved UID/GID

- effective UID - used for permission checks
- real UID - user who ran the program
- saved UID -
- seteuid, or setruid
    - if you run printuid in sudo, then u get all 0's

chroot

- useful for jailing
- what it looks like
    - chdir(/jail)
    - chroot(/jail)
    - setuid(uid), uid !=0, bc if it's root, then like they can do anything
    - can't escape from this jail, can't access anything outside of it

LECTURE 6: Software Fault Isolation

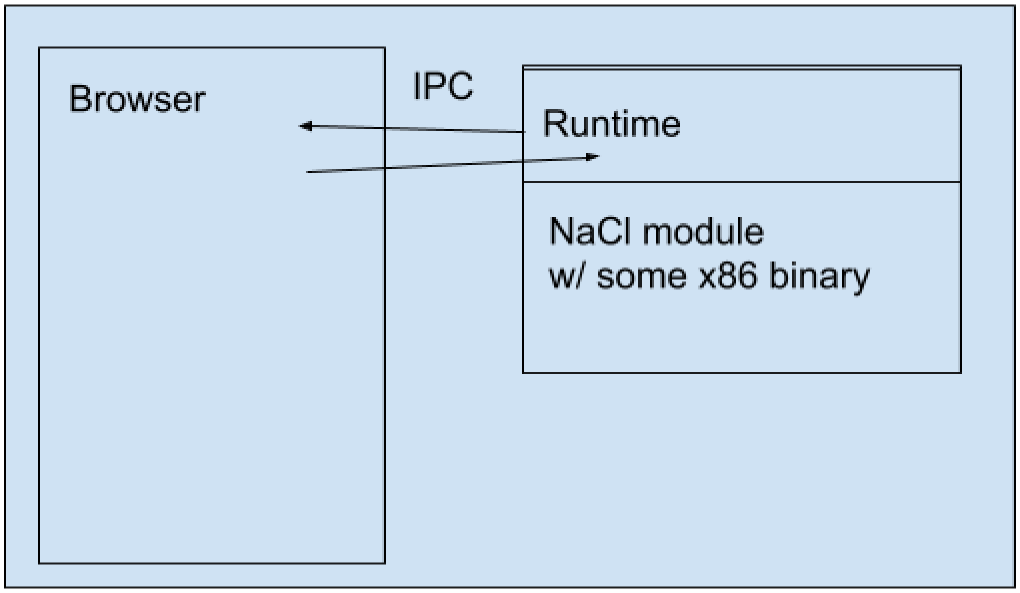
Native Client - Reading 2/27

- Native Client - sandbox for untrusted x86 native code
    - software fault isolation and a secure runtime to direct system interaction
- motivation - high computational load requires other languages
- separated into a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting native code
- threat model
    - untrusted modules + arbitrary code
    - NaCl module may rack up resources
    - may send data via communication interface etc

- inner sandbox - uses static analysis to ensure all reachable instructions are identified and are safe
    - ex. cannot invoke os directly
- IMC - handles communication from/to/between NaCl modules
    - JS uses sockets to communicate with modules

Lecture 6 - Sandboxing

- keep the attacker in rather than out!
- Concepts it depends on
    - VMs, language, processes, chroot(), SFI
- Tradeoffs: light or heavy? slow or fast? compatible or new env? integrated or distant? Are we dealing with bugs or malice?
- **Existing solutions - 1) trust code developer?** ok but users are useless **2) O/S mechanisms, like OKWS, where you restrict what system calls the untrusted code can invoke?** NaCl uses this as its outer sandbox. but unfortunately, some sandbox mechanisms require root. **3) SFI (NaCl) - check instructions**
- all native clients can do is modify their own memory and exchange messages w the javascript via IPC
- what it does
    - it can catch buffer overflows and other blatantly illegal things before bad stuff happens



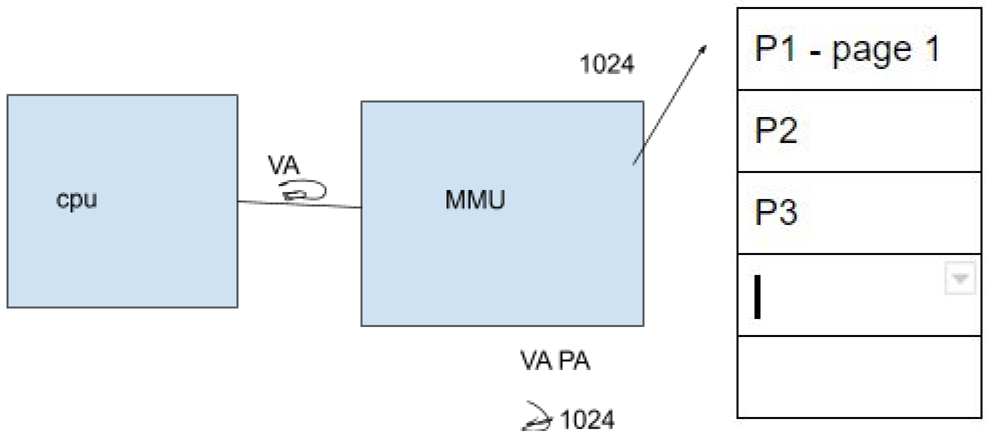
How do we safely run x86 code?

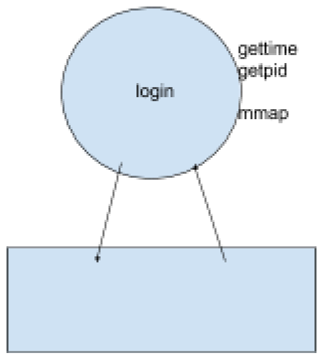
- control what system calls the untrusted code can invoke
- but this is quite different over O/S's. and some mechanisms may require run as root.

SFI (software fault isolation)

- validator - ensures checks are present. checks each instruction in the binary
  - if instructions are sometimes safe
    - compiler inserts check instructions
    - like sometimes JMP, LD, ST
  - never INT
- Why are variable length instructions a problem?
  - x86 instructions can be anywhere from 1 to 15 bytes
  - 25 CD 80 00 00
  - at 25, it's a five byte instruction. from CD, its a 2 byte interrupt instruction
  - so if u jump to the wrong place, its bad
- How do we prevent this?
  - Fall through disassembly - goes through to bottom of instructions with some offset. (like 0 . 5. 10. 15. etc)
    - yes this is just a subset of instructions
  - remember valid instruction addresses
  - look at every jump and require it targets a remembered location
- But what about indirect jumps???? like virtual method calls and program computing target address at run-time
  - AND $0xffffffe0 , %eax
  - JMP *%eax -> clears the lowest 5 bits and requires it goes to multiples of 32 bytes
- 32 is optimal because some people decided it was. (16 is probs too small and 128 would just require a lot)
- segmentation
  - MMU hardware provides segments
  - CPU -> MMU -> memory
  - each memory access is wrt some “segment”
  - each instruction can specify what segment to use for mem access
    - ex. code fetch uses %cs segment
- So what does this solve?
  - *buffer overflows* - well this would involve an indirect jump after an overwritten return address, so this is disallowed. We can only jump to validator module code, which isn’t useful. Also the code pages are write-protected, so it's not like we can even inject code.
  - and we can’t escape!
- *How can we still exploit this?*
  - exploiting bugs in the inner sandbox ie the validator
  - exploiting the outer sandbox via system calls
  - exploiting bugs in the main browser via the IPC

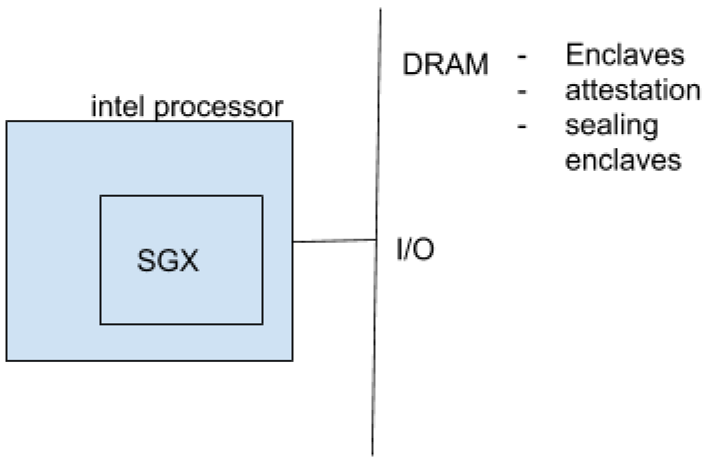
SGX + Haven - when the OS is malicious

- Process isolation, and all are isolated from the OS (even though they communicate with it. Processes use system calls to talk to OS
- Implement Virtual memory - process page tables
    - 

        - DRAM
- threat model for process isolation
    - OS is trusted
    - process is malicious
- threat model for SGX
    - The OS is malicious.
    - hardware is trusted
- Goal for SGX - OS cannot modify the process.
- What can the malicious OS do? (The Iaggo attacks)
    - 

gettime and getpid turn out to be important bc replay attacks, if OS modifies gettime and getpid. what if your random seed is :(
    - mmap - if OS modifies mmap to map a physical page that OS controls
- Defend - use trusted hardware
    - TPM/late boots
    - trusted hypervisor
    - special processors (enclave processors)
    - SGX

- SGX



-
- EPC - Enclave page cache
    - real similar to DRAM, but it's inside the processor!
- EPCM - a map that has something for each EPC page
    - page type
    - enclave ID
    - virtual address for the page
    - permissions
    - map is consulted on each enclave page access
- Instructions
    - ECREATE(va, size)
    - EADD(va, src, perms)
        - OS allocates page in EPC
        - SGX fills it with content from outside PRM (But still inside process)
        - *but how do we know if it's good code?*
            - EXTEND(va) - updates measurement/hash. adds the hash of the page added.
    - EINIT(sigstruct) - have the process enter enclave mode
        - contains signed hash of the content we expect to be in the enclave
        - signed w/ private key of developer (and dev's public key)
- Attestation - is the enclave running the correct code?
    - EREPORT() - generates signed report=[hash, signed hash, public key]

Now, what problems are solved?

- seed - RDRANDOM
- mmap - well now you can't allocate memory in your enclave. so you're good here.
    - mmap modifies page tables - EPCM prevents this attack
- you can't read or write data directly into the enclave (EPC prevents this)
- buffer overflow into the enclave (when something is copied into the enclave)?
    - EPC copies code into enclave - if it doesn't check, then it will overflow.

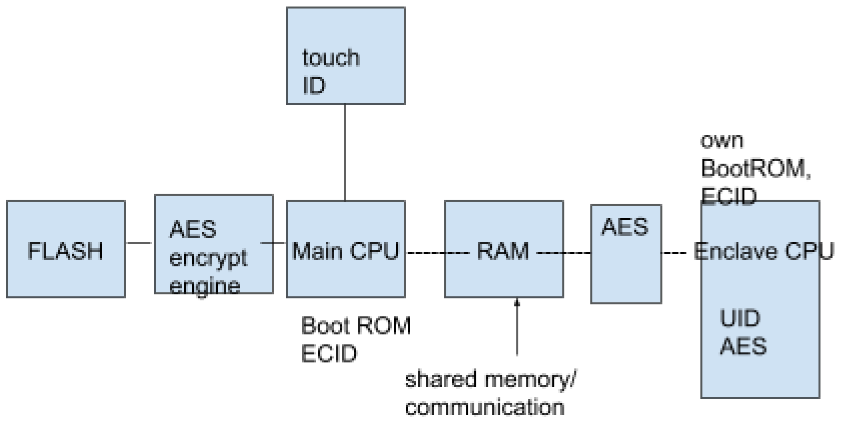
Security Discussion

- TCB
  - Hardware is trusted - if you have the processor, then keys aren't exposed
  - Intel is trusted
- I/O - what if your communication is compromised and adversary can see all the keys you type in
  - paper for Wednesday actually does secure I/O
- Side channels - side leaks - maybe the OS sees page faults and can reconstruct contents of enclave? thats bad

LECTURE 8: CLIENT DEVICE SECURITY

Apple Security

- **Main Attack** - theft of iPhone? extraction of data?
- **Assumptions** - device is locked at time of theft and is passcode protected (otherwise everything is useless)
- **Potential attack types**
  - trying all passwords
  - impersonating fingerprint or face
  - extract data from storage chips or RAM
  - install hacked version of OS?
  - exploit a bug in the OS
  - downgrade the software to a version w/ known bug
- **Hardware Architecture**



- **Secure Boot**
  - Boot ROM - contains a public Apple key to check signatures and ECID of software its gonna load. For example, if the kernel from FLASH doesn't have the signature, then boot rom is not gonna load the OS kernel.
    - prevents hacky OS's from being loaded

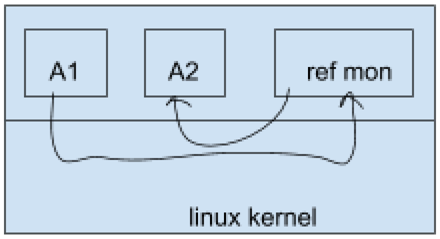
        - if you remove the Boot ROM and replace it with your own thing w ur own public key then you win, but thats **hella** hard
        - UID is also hardcoded (and really impossible to read) on the central chip
        - ECID is also burned into central chip - when you upgrade your phone, your phone sends up the ECID and Apple sends back a signed upgraded OS with your ECID included
            - prevents downgrade attacks as it prevents OS's without your ECID from being loaded
    - now that we got our signed kernel, we can see our secure enclave
        - idea is **main CPU** doesn't see any keys
        - idea is it defends against passcode guessing
        - Secure Enclave is pretty similar to SGX
            - shares the goal of hiding stuff from a possible compromised kernel
            - Here though, the enclave is a separate CPU rather than a separate operating mode.
        - The enclave handles all the password/key stuff and deciding whether a fingerprint is a good fingerprint or not
            - decryption keys, fingerprint/face data on enclave
        - Primary authentication method: Data crypto keys are derived from passcode (encryption keys not stored)
            - Passcode -> Enclave_UID(Enclave_UID(passcode)) = data encryption key (encrypt w/ UID many times)
                - prevents attacker from guessing a lot of times on their own computer because the Enclave CPU can limit password guessing, and you have to check password on Enclave because UID is only known by the Enclave.
        - key wrapping - E_{k1}(k2)
            - idea - each file is encrypted w/ a key (Kf_n)
            - file system has a master key (Kfs)

Content	Metadata
f_1 E_Kf$_1$(f1)	key for file encrypted w/ or wrapped w/ data encryption key $E_{\_KD}(K_{f1})$
f_2 E_Kf$_2$(f2)	$E_{\_KD}(K_{f2})$
	Enclave returns K_{fn} encrypted with $K_{AES\ e,}$ because only enclave knows KD. but the OS never sees the raw keys.

        - What this implies is that if you have Kf1, then you can decrypt f1.
        - The metadata is the file's key wrapped with data encryption key

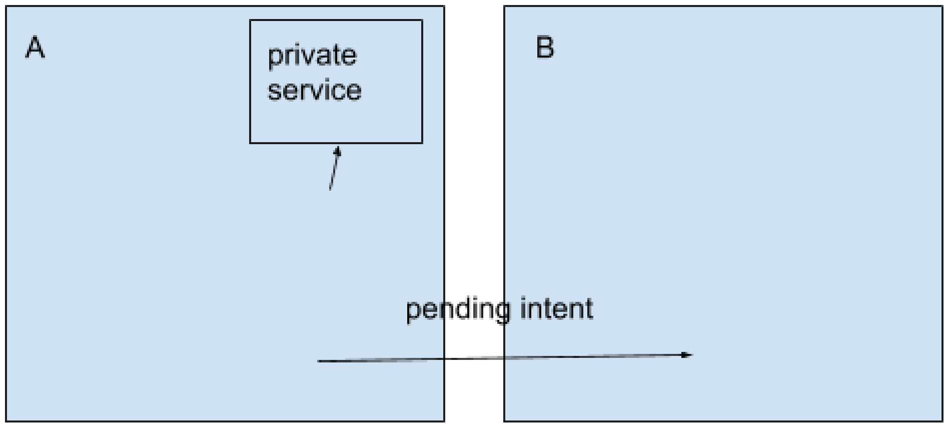
            - Multiple levels of data protection
                - Complete protection
                    - can only decrypt if phone is unlocked - derives KD from passcode! and then its thrown out when phone is locked
                - Until first auth
                    - derived from passcode on first unlock, discard on power-off

LECTURE 9: Android Security - app vs app access control

- Assumptions - isolation of apps, bug free software, no physical theft
- What if we use our desktop application security scheme?
    - every app has the full user permissions though
    - no app vs app control for the same user :/
- <u>App architecture</u>
    - **Components**
        - Activities - UI things, accepts input
        - services - background services, can see RPC things
        - content provider (special service) - SQL database
        - broadcast receiver - hears things from other components
    - **each app has....**
        - private file storage
        - a single Linux process - its own UNIX UIDs
            - now Android uses SELinux and seccomp to limit system calls and which ones are on an allowed list
        - a manifest file declaring permissions
    - **Intents**
        - component target name
        - action (the opcode)
        - data - ref to data (like a URI)
        - 

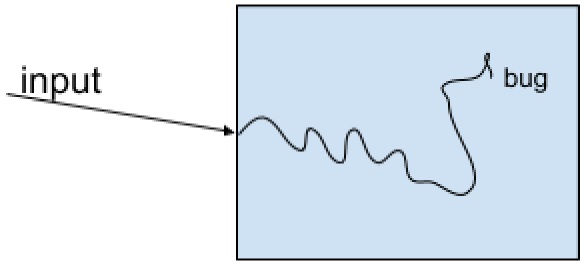
track of an intent A1->A2
        - *implicit intents* - good for components that send things w/o knowing exactly which thing to send it to

                - For example, an app who wanted to look something up in contacts wouldn't know which contacts app to choose - the reference monitor would handle that.
            - *reference monitor -*
                - decides where to send something
                - decides whether to send something at all
                    - "permission labels" like com.android.phone.DIALPERM
                    - each app has a set of labels
                    - Each component has a single label.
                    - things will only be sent from app to component if that label is in the app's set of labels
    - Permission Types
        - Normal - unlikely to be a security problem
        - Dangerous - like SMS, contact info (will notify user tryna install)
        - Signature - can only be used by apps signed by same developer
    - Note that the manifest helps users understand security implications and helps Google Play analyze apps. This is an example of **Mandatory Access Control.**
        - permissions are separate from code, as opposed to **discretionary access control** where permissions may be changed over time
- **Sounds good for accessing components, but what about broadcasts?**
    - what if we have private data that's being sent over?
    - After all, open components receive all data. [yike]
    - so we send(intent, label)
    - *Temporary Permissions*
        - URI delegation - if app1 send a URI to App2, then app2 can read URI content too
        - reference monitor keeps track of delegations
    - *Pending intents*
        - use cases- callbacks into your app like for event notification



        - pending intent later returns to component A, which then forwards it to private service. We know then that the notification must have come from B because we sent it the pending intent.

LEC10: Symbolic Execution

- Question: How do we approach testing?
  - We could use checkers, we could have tests, or we could have fuzzers (generates random inputs, but this is hard to cover all cases)
- **Symbolic Execution: execute with symbolic values**
  - **build path constraints - avoid impossible paths.**
  - **Generate an input to create the error**
  - 

  - Question: What kinds of bugs can we find?
    - division by zero
    - null dereferences
    - out of bounds
    - triggering asserts - finding bugs that trigger when your own app doesn't fit your desired behavior
  - Example
  - Path example

read x,y	let x = alpha, let y = beta	no constraints
if x > y:	fork into 2 programs (don't take this path) -	alpha <= beta
x = y	not run	
if x < y:	for into 2 programs - take path	alpha < beta
x = x + 1	alpha = alpha + 1	
if x + y == 7:	alpha + 1 + beta == 7	with alpha < beta
error()	yes, this will trigger an error bc there exists alpha, beta	

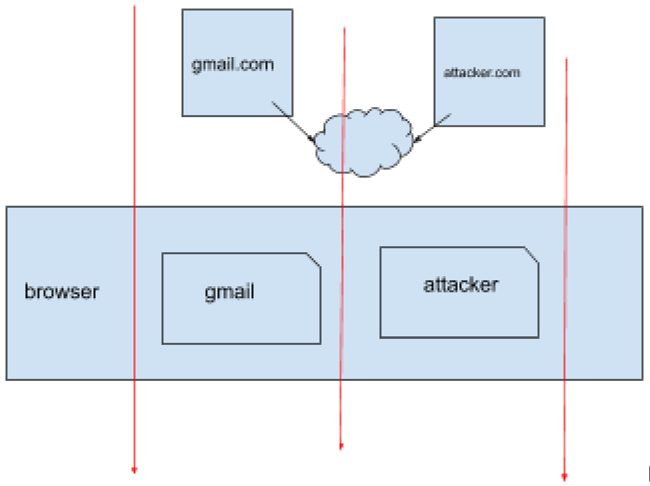
  -
- EXE : C to C translator
  - State of program -
    - table[address range] = symbolic value (as a bitarray)
      - ex - table[x] = \alpha + 1

        - current path constraints
    - EXE adds code to every assignment, expression and branch
    - for IFs, we fork and extend the path condition. We ask solver if there exists a solution for the set of path constraints
    - SMT solvers - finds some problems easy, some hard, and some impossible to determine satisfiability. so we have a timeout on the solver
- Arrays
    - S[c] - concrete value into a symbolic array: easy to figure out, a specific symbolic value
    - C[s] - symbolic value into a concrete array: slightly harder as it could be any element
    - *p - if p is symbolic, which array? Can we guess which array we're tryna go into?
- A real life example, **BPF** - a filter that permits link-layer packets to be sent/received
    - given a packet in, the interpreter(filter + validator) -> spits out something
    - filter = array of instructions with opcode, memory offset
    - the BPF interpreter will iterate over the filter and executing the opcode action
- EXE + BPF
    - was able to identify bugs in BPF
    - what happened
        - generated filters by symbolic execution
        - generated packets by symbolic execution
        - checked if a packet generated error
    - an unchecked buffer overflow error when BPF_LDX opcode
- Discussion
    - size of execution trees?
    - How long will EXE run?
    - What kind of search heuristic are we going to use to decide which branch to go down first? DFS (nah that's not the greatest for this). we use BFS (best? first search)
- environment - how often are we communicating with the environment?
    - with above example, maybe not much
    - but with zoobar, we're having to deal w packets and databases, so symbolic execution will be difficult
        - we will use concolic execution instead.

LEC11: Web Security - Same Origin Policy

- before, there was just static html, but now there's scripting and downloading JavaScript code to your browser
- problems - old browsers, how do websites communicate with each other? different browsers? no good standards?

- threat model- an attacker that can host a malicious website
    - assumption - you're looking at the attacker's website
    - assumption - you also do sensitive things like bank transfers or email
    - assumption - the browser is safe and is our friend
    - assumption - browser is bug-safe
- **Same Origin Policy:** imposed by browser onto websites. sets restrictions on what they can do
    - browser labels scripts and pages w/ host/origin it came from
        - https://gmail.com -> https + gmail.com + 443
    - Resources - servers, DOM
    - Rule: if you go to attacker.com, it's not allowed to see the DOM of gmail.com bc origins aren't the same



- red - SOP
- JavaScript XMLHttpRequests - by default javascript can only send these to its origin server. --see CORS
- authentication between browser code and server - SSL (http**s**)
- question: how gmail.com know a request for your email came from you and not someone else?
    - **COOKIES** - keeping state of the browser
        - anytime we make a request to gmail.com, we include all the cookies we got from gmail.com
        - like literally set_cookie: key=value, domain=google.com
            - javascript can only return cookies for a specific domain
        - session ID
    - **problem -** omfg we literally send back all the cookies with every request, which is problematic. **idea** is that you only send cookies back when you're tryna get sensitive information
    - the browser also disallows attacker.com's domain name to not be a suffix of attacker.com

- **cross-origin**: we can call other people's images in the html, but browser won't let javascript access it. In calling the image, we send all the cookies too when tryna get the image, which is bad
  - **cross-site request forgery**
    - example: attacker.com, <img src="https://bank.com/xfer/500/attacker">
    - bank server sees the request and so the request will include session ID/cookies for bank and it'll complete the transfer
  - **CSRF Defenses** - tokens sent in every bank URL, which is sent by bank website to bank server, so server knows it's really the bank
- **iframes -** small windows into another website within another site.
  - any website can generate iframes that refer to any other website
  - **cross origin resource sharing:** the site will ask the server from another origin and ask "will you do this? yes or no"
    - protects servers from malicious requests. these are all in the headers
- **another attack - Cross site scripting (XSS)**
  - <Script> js js js sj js </script>
  - browser executes above in a comment box? oof YIKES
    - they put stuff in it like grabbing your cookies
  - solution - you can mark your cookies as "only acceptable by http, and not javascript, even same-origin js"
  - solution - strip out tags, parsing

lab3 NOTES (symbolic execution)

unsigned avg - take the a/2 + b/2 and + 1 if a & b & 1
concrete values vs symbolic values - concrete is like 42, when symbolic is like a+b/2

watch branches - make a symbolic constraint every time this happens

decide if there's an input that will cause the program to execute a certain way - requires a sat solver to see if some branch is satisfiable

What are we looking for? well we're tryna kill some invariant, or perhaps some crashes in C, or maybe some Python-level code injection attacks like via an input into eval()

scheduler decides which path to try, like DFS or BFS

fuzzing - randomized approaches, inputs to program python - all ints and strings, which is a ton of inputs. so luckily, we have CONCOLIC execution, which is b/w fuzzing and symbolic execution. store both a concrete and symbolic value. branches take both into account.

What we did

1. we used Z3 expressions to represent conholic values of variables look at symex/fuzzy.py
2. sym_str and sym_int both inherit from sym_ast
3. const_str and const_int inherit from sym_ast
    a. both test for equality for types too
4. we implemented a bunch of symbolic operations like multiply and divide and concat/endswith
    a. for example, dividing two expressions is just z3epxr(a)/z3expr(b)
5. ASTs are constructed by using these variables and other operators
6. concolic_int and str both have a value and a sym. operations return new concolic_int/str objects that utilize sym operations and actual value operations
7. concolic_Exec
    a. FIRST concolic_find_input: given a `constraint`, ask z3 to compute concrete values that make the `constraint` true. put these into `ConcreteValues`.
    b. THEN concolically exec the `testfunction` with the given `concreteValues`
        i. figure out the branches it encountered to compute some given result
    c. EXPLORE DIFF BRANCHES using `concolic_force_branch`. here we negate the branch condition of the bth branch in `branch_conds`
    d. now we execute every branch!!!!
        i. concolic_execs()
            1. first get the concrete values and then concolic_Exec_input to get branches it encountered.
            2. get different branches using concolic_force_branch
            3. find new inputs for these new branches
8. Then we went and checked a whole bunch of inputs for zoobar using concolic inputs on the key names and we found bugs!!

LEC12: TCP Security

- TCP Sequence # Prediction is a problem - see paper notes
    - *Defense: change the ISN variable #, random increment, crypt ISN gen*
- Why is this ^^ a security problem? Check Source Routing abuse
    - Forging IP source addresses is a problem with IP address authentication schemes.
        - *Defense: don't use this! lol*
    - DoS connection resets - send a bunch of stuff

    - Hijack existing connections
        - Guess a seq # - inject data into existing connection
- *Defenses: use cryptographic authentication like SSL. Filter out packets w/ forged IP source address.*
- DNS
    - messes up because if adversary knows the UDP port of a client and the DNS seq # of the query (same prob as above), then they can send false addresses for name translations oh no
    - *Defense: randomness, DNSSEC. MIT DNS servers*
- SYN flooding - DoS
    - so the server is waiting for the client's ACK(SN_s) to confirm
    - but what if the attacker starts up like 50 fake connections and then the server is just waiting for 50 fake acks
    - now the server just ignores real connection requests oof
    - *Defense:* SYN cookies - *make the server stateless until it receives the ACK.*
- bandwidth amplification - DoS
    - send ping packets to broadcast address. each machine on the network will send an ICMP echo reply - fake a packet through from victim's address, so they get all the replies lol
    - *Defense: you can't send packets to a broadcast address now.*
- Routing protocol abuse
    - BGP - what if attackers control ISPs and BGP routers?
        - you can announce you have a path to MIT and then everyone routes through you and then you can see all the traffic
    - *Defense: sign original announcements, and then there is a trusted dB of who is allowed to announce what. still not great though*

LEC13: Secure Channels

paper notes - cut and paste attacks - cut sensitive data into the header, so that when they send the decoded header to translate to DNS, the attacker can intercept it and see sensitive data.
[link to helpful SSL stuff]
- Public key operations
    - **KeyGen() -> Public key and a secret key**
    - Encrypt(PK, message) -> ciphertext c
    - Decrypt(SK, c) -> message

    - Sign(SK, message) -> signature
    - Verify(PK, message, signature) -> ok?
- private/symmetric key operations
    - Keygen -> K
    - Encrypt(K, message) -> ciphertext c
    - Decrypt(K, c) -> message
    - MAC(K, m) -> tag
    - whats the diff b/w digital signatures and MAC auth
        - basically MAC maintains integrity of message, and digital signatures verifies integrity of message and sender
- Proposing a (broken) secure channel

C --> S [connect]
C <-- S [send the public key]
C --> S [Encrypt(PK, freshkey)]. S can decrypt the private freshkey
C <--> S [Encrypt(freshkey, messages)]

Problems: What if C doesn't connect to S, and it's some malicious S'?
Solution - Certificates. Trusted authority server that maintains a table of name/publicKey pairs

Second Problem: We never authenticated messages. Some middleman [C -> M -> S] could change some bits in the encrypted message, and S will just believe the message is fine.

Solution - authenticated encryption.

Last step: C <--> S [c=Encrypt(freshKey1, message), t=MAC(freshKey2, c)]
Message will be rejected because the tag will be incorrect.

third problem: compromised server that can use SK to decrypt packets (and later figure out freshKey)
Solution - forward secrecy! don't use long term keys for encryption. use them for signing, and then use short-term keys for encryption.

step 2 C <-- S [I got a PK so let's check via cert, and then sign(SK, PK_conn)]

final handshake (in english)

preventing SSL rollback attacks - client adds marker in authenticated padding bytes, and then the server knows client can communicate in 3.0, not just 2.0

POODLE - problems padding bytes are unspecified, and MAC doesn't look at padding.

LEC14: CERTS

nowadays - let's encrypt! CA says "if you are the true owner of a server, then you must be able to upload or update a file to your server" so then it retrieves the file and checks the nonce to see if its correct

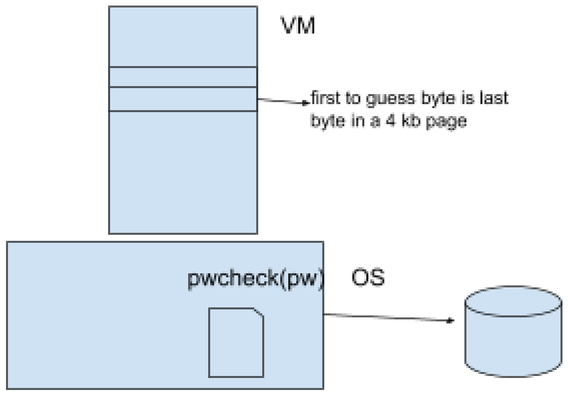
What if the CAs are corrupt? Well that's why we look at multiple CAs. so people haven't really thought about that, but then it happened. so if you pay someone off at CA and get them to make you a google.com certificate and you stole the google.com private key, and you manage to reroute peoples traffic to attacker.com, then you can ask for their login and then you just forward that along to google.com and send them back their inbox but SHOCKER YOU GOT THEIR PLAINTEXT and you've just executed a MITM attack.

LEC18: Spectre

- speculative execution - executing code during delays and predicting outcomes and seeing if they were right
- Spectre - tricking the processor into executing instructions that should not have executed [transient instructions]
- steps
  - locate a seq of instructions within the process address space which when executed acts as a transmitter of memory or register contents
  - trick CPU
  - retrieve info over the covert channel
- example - conditional branch example on page 2
- features of modern high-speed processors
  - out of order execution
  - speculative exec
  - branch prediction - BTB keeps a mapping from addresses of recently executed branch instructions to destination addresses to predict future code addresses
  - cache hierarchy

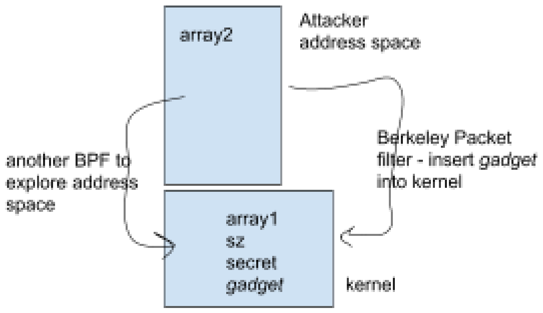
- attack overview
    - setup - mistrain processor so that it will later make an erroneous prediction
        - prepare side channel
    - execute instructions that transfer confidential info, like via syscall or socket
    - sensitive data is recovered, flush+reload or evict+reload

LEC18...? : Side Channels

- Modern Side channel attacks
    - web page cache based on load speed
    - image size directly correlated with some metric on a chart
    - private key interception - steal the private key on the other side of the SSL channel based on slightly wrong public key encryptions [bit by bit] and measure speed of server response
    - POODLE - cookie was stolen by trying different padding lengths
    - TenexOS - virtual mem on top, OS kernel has pwcheck(pw), and passwords stored in plaintext
        - 

        - kernel checks password one byte at a time, and then returns error when mismatch
        - adversary aligns password ST the first to guess byte is the last byte in a 4 kb page, and then unmap the second page to disk
        - so if the first character is correct in the guessed password, the OS will take a long time to bring back the second page to check the rest of the password
- Spectre
    - processor side channels
    - break isolation

    - architectural side channels
- Goal of processor design - performance
    - for performance, speculative execution [make a guess of the code you’re intending to run next when waiting for a branch to execute]
    - for performance, implement cache hierarchy
- example of speculation:
    - `if(offset < size) {`
        - `[code]`
    - `}`
    - if size isn’t cached, then it has to bring a value into L1 cache, but yikes that takes a while, so we’ll just guess how this branch is gonna go.
    - so maybe the last 100 times, this branch was taken. so we’re gonna guess its gonna go ahead in that way
    - So once we get to the branch, if we do take it, then yay we commit all the stuff that we had run. if not, then there's a **speculation failure,** so we need to squash CPU state and reload it with the state prior to branch (bc we checkpointed it)
        - but unfortunately, it leaves traces if it was a failure. like the state of the cache isn’t rolled back. and this is enough to do side channel attacks.
- how to exploit this
    - outline:
        - `char array1[sz]`
        - `char secret[8]  # ‘abcdefgh’`
        - `char array2[]`
        - `if (offset < sz) {`
            - `v = array1[offset]`
            - `v2 = array2[v]    # find array2[“a”]`
        - `}`
    - arrange for `sz` not to be cached, and make sure array1 to be cached so that speed is all good and we aren't squashed before line 6. make sure `array2` is *not* cached. such that line 6 will now put `array2` into the cache.
    - but what if we get a value that goes outside the bounds of array1
    - but we can't just directly see v in line 5. but we coulllllld get lucky with line 6.
    - **<u>the measure step</u>**
        - we’re now indexing into array2 (which is size 256 because letters are a 2^8 kind of deal)
        - we can measure `array2[0], array2[1], array2[2]`......until `array2[“a”]`, which is really fast actually because we queried it in line 6
- challenges of this

    - finding the gadget to exploit
    - attacker must be able to access `array2`
    - being able to evict `sz` and `array2` from the victim's address space
    - training the branch predictor
- What is appendix A doing?
    - well lines 11 to 30 is the victim code (secret + gadget)
    - and lines 31 - end is the attack code
- other setup
    - 
array2
Attacker address space
Berkeley Packet filter - insert *gadget* into kernel
another BPF to explore address space
array1
sz
secret
gadget
kernel

LEC15: airbnb lecture

- XSS bugs have been around for like 19 years
- problem: security isn't scaling with the rate of everything else
    - well why don't we just test it? well there's not that many people who are trained to
- new bugs also are a thing - containerization and kubernetes, blockchain. even really basic stuff- re. spectre
- security is dependent on use, threat model, budgets, company cultures
- how to prevent risk
    - be threat-agnostic: assume something will fail
    - build specific defenses around every point of failure
    - always self-assess (bug bounties, human review)
- why is this hard
    - systems are built on growth. if security opposes growth, things will be sad.
    - so security tends to focus on the most immediate threats (external threats)
    - and then it becomes HARD SHELL - soft center, so once you get inside, you're really inside

- lots of attacks on companies fall through a path through weak links in specific servers or interfaces and then u can get into a really privileged part of the system
    - so tldr segmentation is hella hard.
    - internal controls is hard.
- airbnb case
    - 2500 services, 20000 network nodes (like IP addr), and 1100 engineers and hundreds of deploys a day
    - question: can we implement security without people/other engineers noticing?
        - philosophy: don’t build security around the development process. unify them.
        - security should be out of the way of engineers, but it should be **hard** to do something insecurely.
        - security should be flexible for the network/protocol
    - approach
        - **TLS in service discovery proxies**
        - **identity bound to nodes**
        - **generated authorization map**
        - so nodes each have some ID and they are connected by TLS pipes, and the map feeds them info on auth
    - Mutual TLS
        - **client verifies identity of server** or server verifies client identity
        - tldr: two way auth based on signed keys
    - service discovery - system for one node to discover another node, based on identity or function
        - smartstack - a map of the network
        - old way - service A - >outbound proxy -> service B over HTTP
        - new way - service A -> outbound proxy ->inbound proxy -> service B over HTTPS
            - we can just check auth checks here!
            - good for tracking metrics
            - sending/receiving looks the same
        - But what if you have some attacker trying to connect to the service B using HTTP request not HTTPS? well they bind underlying services to localhost so nothing outside can get in
    - identity binding
        - segmentation - service level, not subnet level
        - allow payment config page to call payment service, but don't let the Slack bot!

            - well we have proxies that understand TLS on both sides of service comms, and TLS is good at verifying identities
            - So how do we identify nodes?
                - identity should be sufficiently varied, unchangeable, easily detectable, represented in a cert (so make sure its p short)
                - build a map of what identities should be able to access which services
                - distribute map to service discovery proxies and enforce it
        - building a trustworthy map
            - make a configurable list? well that's a lot of effort
            - infer it from existing code!!!!!!!!!!!
            - maybe if a service has some other services in dependency, and you push another one, then you update the map!
                - arachne - generating a web of services. figures out what services should talk to what other services.
            - What kind of barriers are you going to put in place to changing the map? well that depends on you.
        - fine-grained authorization - inject headers into HTTP streams, signals auth info to app code
- cons
    - well you need to constantly sync your allow-list to your nodes
        - caching can help this
    - if TLS messes up, you’re really screwed
    - added complexity in reverse proxies
    - installing those listener softwares on the services is easy when u own everything, but what if u have some third party vendor software
    - cert revocation is sometimes tricky
- PLAN - to roll this security plan out!!!!
    - compute auth map and verify correctness
    - ship everything a cert
    - install receiving proxy everywhere and LISTEN
    - configure some services to start using the system
    - cut everything over overnight lol
    - bind services to localhost so secure proxy must be used
- things that went well
    - 14.8% internal TLS -> 70.1%
    - ensured non-security benefits
    - you can disable the system selectively
- not so well

    - thing going thru inbound proxy have unexpected behavior
    - binding to localhost took long time
- performance - better TLs can be use same session instead of redoing the handshake
    - lots of things had hand-implemented mutual-TLS on the app layer, so a lot of these things started up fresh each loadup time. so when they started using this protocol, there's more TLS session caching, so they don't have to do the handshake all the time.

LEC16: Messaging Security

Priorities: confidentiality, authenticity

- alice --TLS--> gmail --TLS--> list server -> IBM -> MIT -> CSAIL --TLS--> bob
- **confidential:** only Bob and Alice can read the message
- **authentic:** bob can be sure alice sent it

Hop by hop security vs end to end security - end to end is much more believable in that it depends on src/dst users that don’t have to depend on an underlying network security

Three things

1. **establish trust**
2. **conversation security**
3. **transport privacy**

each prioritizes security/privacy properties, usability, and ease of adoption

threat model

1. local adversary - controls local networks
2. global adversary - controls nation-state level networks
3. service providers - service operators of messaging systems

Primitives

- m = plain text
- h = Hash(m)
- tag = MAC(shared key, m)
- c = E(symmetric key, m)

- m = D(symmetric key, c)   <- symmetric encryption (both parties have to have same key)
- c = E(publicKey_alice, m)   < - Bob to alice
- m = D(secretKey_alice, c)   <- Alice decrypts the message. <- public key encryption
- signature = Sign(secretKey_alice, m) <-- Alice signs a message with her secret key
- ok = verify(publicKey_alice, m, signature) <-- bob checks the signature with Alice's public key

Secure single message (PGP)

- Alice --> Bob
- c = E(publicKey_bob, m)
- s = Sign(secretKey_alice, m)
- Alice sends c,s to Bob
- Bob can check verify(publicKey_alice, c, s) and then D(secretKey_bob, c)
- Problems: what if a replay attack happens [no replay detection]? How do we get the public keys for Alice and Bob? Is this interactive? like for games or instant messaging vs one-time emails?

For games, and apps like that, you can use a **secure channel [TLS]**

- Alice<>Bob
- A sends to B -> E(publicKey_bob, "A_identifierB_identifier" + random #), sign(secretKey_alice, the message)
- B sends to A -> E(publicKey_alice, "A_identifierB_identifier" + diff random #), sign(secretKey_bob, the message)
- now they both have "randomArandomB", so they can both compute a hash of keys
    - Kx = H(randomArandomBx)
    - so take K1, K2 and
    - A -> B: E(K1, messages+seq#), MAC(K2, M)
- Can we replay bits? no, because random numbers are involved. so an attacker would use the old random B #, where B actually generates a new random number. also like think about if adversary pretends to be A and sends the first line, but then B seconds back it's randomB # to actual A, and then adversary can't successfully pull off the replay because it never got the new randomB
- seq# prevents replay attacks during an active conversation
- but again, **how do we get Alice's and Bob's public keys?**

Opportunistic encryption

- magic
- A->B: publicKey_Alice
- A <- B: publicKey_bob

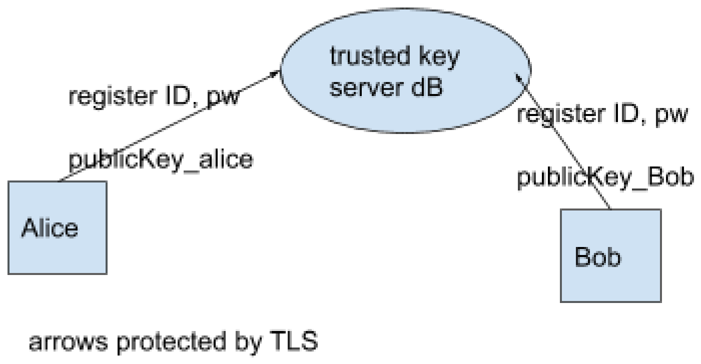
Trust on First use TOFU

- assume first connection was OK

- PIN public key
- like SSH - do you want to connect to this unknown or not-yet-trusted endpoint?
- human validation of connections
- pros: convenient, strong defense against a new attack
- cons: not so useful against a long-term attacker, what happens with new devices?

Key Server

- Alice < ------ > Bob



-
- Alice can get bob's public key from server
- remember only Alice and Bob know their secret keys. so the key server can't decrypt messages.
- What if the key server lied about the public key and returned some adversary's public key? we just absolutely have to trust them

how to make sure key transparency is there and key servers aren't bad:

- public log of key changes -
    - like Bob added PK_x
    - Alice added PK_y
    - Bob deleted PK_z
- key owners periodically check the log to make sure it's right for their keys

LEC20: IS&T

- MIT network
    - DNS - external view on Akamai, internal view of DNS addresses internally for MIT
- DDoS happens a lot
    - obfuscation, IP spoofing, not enough providers provide filtering for spoofed packets
    - botnets - using a bunch of legit bot systems to flood. some botnets for hire

    - mitigating attacks - DNS record of target to service provider (Akamai)
    - mitigating brute force attacks - BGP mitigation
        - BGP uses AS's to route packets through the Internet.
        - Inject another AS that acts as an MITM. If a target comes under attack, then the additional AS advertises itself as the target, and all traffic goes towards the added AS. This AS is owned by Akamai, and Akamai can filter out and redirect all legit traffic to us.
- big attack on 12/3/18 where email filtering came under DDoS attack.
- emails went under a migration from Symantec email filtering to Microsoft Exchange Online Protection Email filtering. They had to implement support for SRS for email forwarding.
- terms
    - MTA - mail transfer agent
    - SRS - sender rewriting scheme
    - SMTP - simple mail transfer protocol - send mail
    - DKIM - DomainKeys Identified Mail (prevents mail spoofing) via digital signature
    - SPF - sender policy framework - identify which mail servers are permitted to send email on your domain.
    - DMARC - domain-based message authentication, reporting and conformance (a standard for a domain to publish a policy regarding email validation via SPF/DKIM)
- mailing lists for a while broke Yahoo's DMARC.
    - from: header doesn't match domain name of list, which broke their SPF
    - mailing list software modified body or headers - broke DKIM

LEC19: Tor stuff

LECTURE NOTES

- anonymity
- tor threat model: system needs to be useful for web browsing. but you're hiding users from other users.
- A -> [R (relay)] -> B to prevent B from knowing it was A that sent the message, and maybe encrypted TCP stream for dual messaging
- multiple (n) people connecting to R
- What if R is compromised?

        - **say someone is watching the relay**
        - **have multiple relays**, each of which have their own encryption level, so now your message is like
            - E1(E2(E3(message))) and each relay1 relay2, relay3 can all decrypt one layer
            - this solves the problem of someone watching one relay bc they still can't read ur message
- question: where do relays know where to redirect traffic after they decrypt it?
    - link local identifier - entries - routing tables - symmetric keys, established w/ public keys
    - pre-tor: Onion structure -
- question: well if the attacker sees both ends, they win!
    - assumption: users choose paths at random, with what probability will a given path be compromised?
    - (c/n) of a node being compromised. (c/n)^2 compromised relays/total# = probability of a compromised path
    - over time, if users continue choosing random paths, then their probability of choosing a compromised path goes to 1
    - ***solution:*** helper nodes. A0 use 1 first hop consistently
    - **what if the attackers roam and get random compromised paths?**
    - **path capture attack -** compromised relays only tell you about other compromised relays
        - if you want to be a relay, then you have to tell all the **authorities** about yourself, and the authorities vote on whether the relay is in or not
        - So how do you inform everyone about relays? well hopefully like...a cache? but how often do they update? every few hours? that might be a lot of network usage
        - well do we need the client to know the whole list of relays? ever !!!???
        - *node discovery:* send snips (one line, authenticated by authority) of the routing table, and let the clients know the snips of the relays where they're going. verifies that the routing table is honoring the client's request for random and routing.

Paper notes

    - terms and main ideas
        - **circuit -** a path of nodes to route packets
        - **perfect forward secrecy -** the incremental path-building design where Alice negotiates session keys which each hop in the circuit
        - **directory servers -** some nodes provide signed directories describing known routers (ORs)
        - **exit policies -** each node has a policy describing hosts and ports that can connect

        - **end to end integrity -** verifies data integrity before leaving network
        - **rendezvous points -** in case Bob doesn't want anyone to know his IP addr, he has several introduction points, and then after Alice contacts an introduction point via a rendezvous point, Bob will connect to the RP
    - design goals - **deployability, usability, flexibility, simplicity (so users don't have to alter their system)**
    - threat model - adversaries can observe some level of traffic, modify or delay traffic, compromise some OR's.
    - **tor design**
        - each OR has a long-term *identity key* (used for signing TLS certs, OR's router descriptor, directories) and a short-term *onion key* (decrypts requests from users to set up a circuit and negotiate keys
        - Cells
            - fixed-size (512 bytes)
            - headings for circuit iD, command. then data
        - circuits - build preemptively. users rotate circuits a lot.
            - to build: 1) CREATE - send g^x, get back K = g^(xy) 2) RELAY EXTEND to the last node in the circuit to extend one hop further.
            - to send: 3) RELAY CELLS - nodes decrypt relay header and processes it
        - streams - OP finds new circuit 1) RELAY BEGIN wait for remote host connect 2) RELAY CONNECTED
            - integrity checking streams - end to end, SHA-1 digests
    - rendezvous points [great diagrams here]
        - Bob chooses introduction points, advertises them on lookup service, signs ad, builds circuit to each point
        - Alice chooses an OR as RP and builds circuit to it.
        - Alice opens anon stream to one of Bob's intro points and sends a message w/ RP and rendezvous cookie, and start of DH handshake
        - Bob builds the circuit to Alice's RP and sends the rendezvous cookie, second half of handshake, and hash of session key.
        - RP cannot recognize Alice or Bob or data.
    - possible attacks
        - DoS
        - exit policy abuse
        - directory servers compromised

LEC 21: Keybase

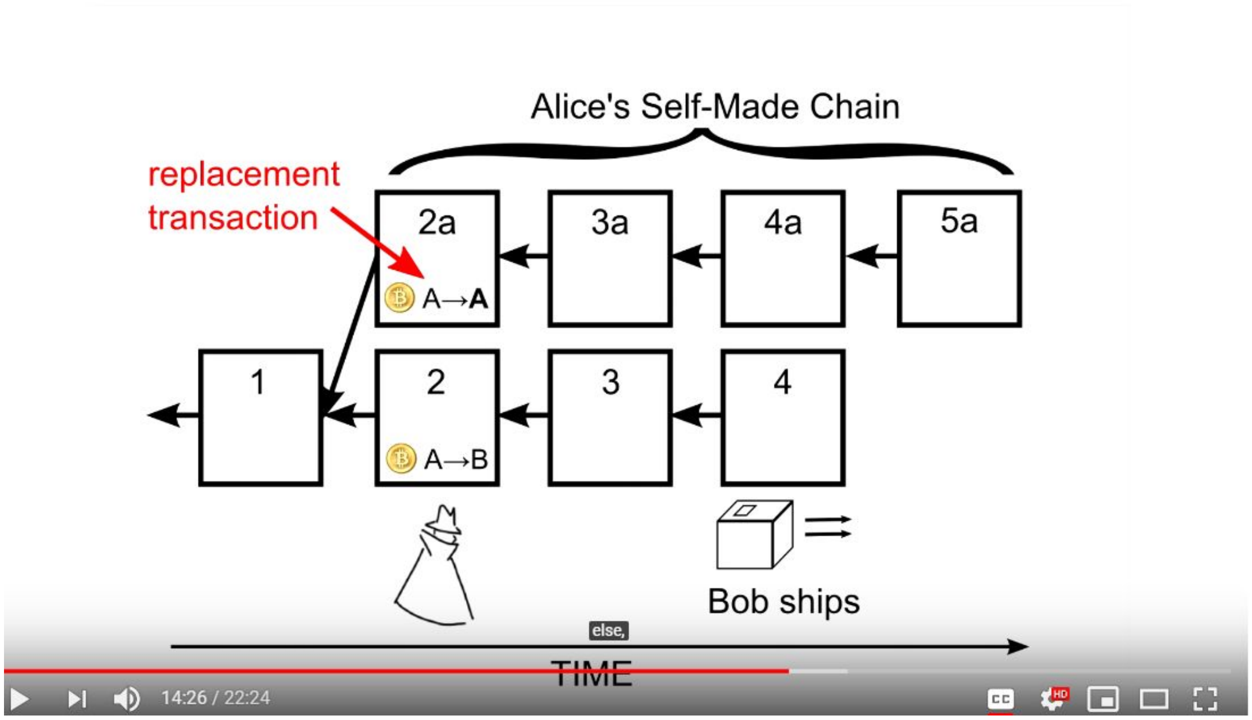
- needs
    - *authenticate* user invitations

    - security across *multi-device support*
    - *customized naming* for groups
- goals
    - future messages are not available to a revoked device
    - forward secrecy (archive or nah)
- **threat model**
    - malicious people own server infrastructure, can recover locked device
- Slack is going public soon - they warned investors like yo we're a target now for nation-state hacking
- ideas
    - keybase v0 - one private key, encrypted with password. new machine? just decrypt w/ ur password and now u can retrieve your private key
        - BUT passwords might not be strong enough
    - keybase vNow - user's think about devices, not keys.
        - idea - all devices are equally powerful. the more devices, the less likely you are to lose your data. you can revoke access to stolen devices from others.
        - idea - lots of systems are like the oldest one is the master, but you're most likely to discard your oldest device. as long as you maintain some set of devices.
        - Devices -> Users as Users -> Teams
- how apps work
    - Every team has a random shared symmetric key that rotates when users are removed from the team or when a device is revoked.
    - updates to chat channel/repo/fs are encrypted for current shared symmetric key.
    - authenticated encryption vs regular encryption - well what if some random malicious user goes and sends random encrypted data via the server? gotta authenticate
        - authentication also u gotta sign by the user that made the update so Alice isn't faking some message for Bob
- chain of signatures - your account on keybase
    - two key pairs (signing key s and key d) never leave your device. sign d with s. [see: using different keys for signing and encryption]
    - a per-user key pair (u) is encrypted for D, and is shared across your devices
    - adding a device?
        - new s, d keys
        - same sign d w/ s, encrypt u for d

    - current way of doing it - say you're logged in on ur comp and want to add ur phone. gotta use the same session key
    - revoking access?
        - revoke S and D from retired device
        - rotate per-user-key, re-encrypts u for all other devices
            - so here we encrypt u' for u. be able to decrypt anything on old devices. but if u dont have new u', then you won't be able to go down the chain
            - this is adding more links to the chain!
    - claimed external identities also live here
- How does Bob look up alice?
    - he asks the server for her chain of signatures
    - what if the server is malicious
        - then it cuts off the chain, and then now bob is encrypted using an old chain, and then old device can be used to decrypt messages bob sends her
    - How do we hold the server accountable?
        - download merkle root from server and verify explicit signature? we don't implicitly trust TLS
        - assert consistency of new links with current Merkle tree
        - fork consistency? using a global Merkle tree prevents local forks, but what if fork the global tree?
            - using the Bitcoin blockchain to confirm the global Merkle tree's correct version (if someone tried to fork it)
- Teams
    - named teams or Alice and Bob teams
        - admin can make membership changes
        - non-admins can just see team secrets
        - add a sigchain link
            - encrypts T for U_c
    - revoking team members/revoking devices/resetting team accounts
        - just rotate the T key to T'
    - loading a team
        - play the team chain forward and ensure the tail matches the Merkle Tree
        - all modifications are signed by authorized admins [new]
- other notes - PGP (pretty good privacy), is used for providing encryption and authentication for data communication in texts/emails. OpenPGP is a standard.
- what is a merkle tree??? general: [x] context of bitcoin: [x]

LEC22: Bitcoin/distributed ledgers - append-only

- we had talked about certificate transparency by having a largescale deployment of a log for certificate transactions earlier
    - In this scheme, a server operator would ask CA for a cert, and the CA would sign it, but also send a copy to the log operator. SO would then send cert + log receipt to the browser. browser requires valid log receipt.
    - SO can use log for mis-issued certs
    - **question: how do we know the log operator is trustworthy?**
        - Merkle tree of the log
        - monitor: store latest Merkle root hash, request a Merkle audit path. Cert w/ correct timestamp must hash up to the Merkle root
        - assert certs are in increasing timestamp over
        - assert Merkle root hash == old log + new entries
- bitcoin - decentralized ledger, log of cryptocurrency transactions signed by users' private keys
- head of ledger = longest chain
- confirming a transaction - find block header in list, then Merkle proof
- consensus scheme : designated servers
    - voting protocol b/w well known servers
    -
- interesting discussion the double-spending problem

- 

- if alice makes the chain much faster after 4 gets confirmed into the blockchain.

lab4 - BROWSER SECURITY

1. part 1: cross-site scripting attack
    a. ex 1 we put a script into users.html that alerts the cookie
    b. ex 2 then we emailed the cookie
        i. this involved invoking a cross-site script via an image that emails pw
    c. ex 3 then we did reflected cross-site scripting by putting code into the URL
        i. we put the </input>ex2<input> into user field in GET request
    d. ex4 same thing we emailed the cookie thru a url
        i. same thing tbh
    e. ex5 then we emailed the cookie and hid our tracks using CSS
        i. set some display css things
2. part 2: cross-site request forgery attack
    a. ex 6 we made a local transfer zoobar form and linked it to our server localhost:8080
        i. copied the form and action=http://localhost basically. hardcoded values in input fields
    b. ex 7 we submitted this form on load
        i. script submits form

   c. ex 8 we hid our tracks by redirecting to http://csail.mit.edu afterwards
      i. after form submits to f, we listen to hear that f is loaded, and then we replace url
3. part 3: fake login page
   a. ex 9: we made a local login zoobar form and linked it to serve localhost:8080 like ex6
   b. ex 10: we look at form values as user submits and alert their password
      i. onsubmit of form, we run steal_password
   c. ex 11: we steal the password via email
      i. add listener to form submit, and we track whether the form was submitted. if first submit, then we email via new Image, and submit the form again to actually trigger the form submit.
   d. ex 12: after stealing pw, user sees the official site
      i. track which button was clicked and reclick it
   e. ex 13: we steal pw if not logged in or we steal zoobars if already logged in
      i. figure out if logged in by checking myZoobars field and submit the transfer form if so. otherwise, do same thing in ex12 w/ button tracking and form double submit
4. part 4: worm
   a. ex 14: we make a worm ST if a user visits infected profile, they get the worm
      i. XML Http request (which you can do bc you're asking for another localhost:8080 thing) and ask for a transfer and set_profile.
      ii. This propagates because the legit script is set as the profile in the db when a user views an infected profile.