CSE 127 Notes

Taanish Reja

Fall 2025

1 Threat Modeling

1.1 Confidentiality, Integrity, Availability + Privacy

1.2 Confidentiality

Prevent unauthorized access to information

1.3 Integrity & Authenticity

Integrity Prevention of unauthorized modification of information, process, or function **Authenticity** Prevention of impersonation of another principal (entity that can be authenticated)

1.4 Availability

Prevention of unauthorized denial of service to others

1.5 Privacy

Person's right or expectation to control disclosure of their personal information (e.g. activity metadata)
Privacy vs Security

- Security is about hiding info from third-parties
- Privacy is about not being observed or monitored (including via public data)

1.6 Vulnerabilities

Weakness that can be exploited to damage assets (assets include CIA + Privacy)

- Implementation flaws
- Default password intact

Rely on assumptions in system

1.7 Threat Model

- Defines security goals
- Example def: "System that remains dependenable in face of malice"
  - What does it mean to remain dependenable? C.I.A preserved (the assets we need to protect)
  - Who are the attackers (capability + intent)?

What are we trying to protect from who? (Assets and Attackers)

1.7.1 Attackers

Defined by capability and intent
Capabilities: Time, money, training, access
Intent: Curiosity, money, fame, national interest

Attackers do not care about your threat model; it is your problem scope (what problem needs to be solved)

1.8 Risk Assessment

Security is not binary (everyything has some measure of risk)
Ideal risk assessment pipeline:

- Start by understanding system requirements
  What is the system supposed to do? What is the security boundary? What is the lifecycle. Where will it be deployed?
- Identify assets and attackers
  Who are the stakeholders? What needs to protected? Who are the attackers?
- Establish security requirements
  CIA + P (what cannot be compromised)
- Evaluate system design
  What are the components? What are the assumptions? Does implementation match design?
- Identify threats and classify risks
  What are the threats to security (what ways may attackers exploit vulnerabilities in system design or implementation to compromise assets?)

- Address identified risks
  Try to violate assumptions. Assume untrusted parts of system work in worst way possible.

Addressing risk:

- Avoid: remove component that creates risk
- Mitigate: add measures to decrease impact/likelihood
- Transfer: make it someone else's problem
- Accept: do nothing

1.9 Trusted Computing Base

- Set of systems your security depends on
- You need to trust something

Security boundary:

- Perimeter around components of same trust level
- Data coming in is untrusted and potentially malicious

Attack surface:

- Set of intersection points across a security boundary
- Parts of system handling input from or otherwise interacting with less trusted and potentially malicious entities
- API/IPC/Parser

2 Buffer Overflows

2.1 Overview

When is a program secure?
When it does exactly what it should do (not more, not less). Very hard to define
Better def: When it doesn't do bad things
But what if a program doesn't do bad things most of the time, only sometimes. Or could?
Exploit: mechanism by which an attacker triggers unintended functionality in system
Vulnerability:

A bug in software that allows an unprivileged user capabilities that should be denied to them
Most classic and important vulnerabilities violate control flow integrity (let attackers run code of their choosing on your computer; typically violate assumptions of programming language or runtime system)
Our threat model:

- Victim code is handling input that comes from across a security boundary (browser, word processor, etc...)
- Buffer overflow: provide input that "overflows" memory

2.2 Taking over machines

Program manipulates data, but data also manipulates your machine
Stack Review
Stack divided into frames
Each frame stores locals and args to called functions
Stack pointer points to top of stack, frame pointer (or base pointer) points to caller's frame on stack
Calling a function
Caller: Passes arguments, call and save return address
Callee: Save old frame pointer, setframe pointer = stack pointer, and allocate stack space for locals

Layout:

arg i+2
arg i+1
arg i
ret addr
saved ebp
local 5
local 6
local 7

Returning:
Callee: pop local storage (set stack pointer = frame pointer)
pop frame ptr
pop return address and return
Caller:
pop arguments AT&T arguments:
instruction src dest
movl %ebp, %esp
mem dereference: offset(%reg)
imm: $num

2.3 Shellcode

Redirect return address to point to existing code (i.e. shellcode we wrote in the buffer)
Shellcode tricks:

- Inline assembly to use gcc to trnaslate from asm to object code (compile and run in debugger)
- Using a call instruction to infer address of payload on stack (class example: jmp to call position; call pushes register and moves up 24 bytes (array size); then pop puts this adddress in a register, and we can do dangerous things)
- NOP sleds; relax constraints of guessing exact location of shellcode. Jump somehwere into NOP sled and slide down

2.4 Remote attacks

So far, shellcode works good locally. How do we open a remote shell to allow for remote attacks?

- Reverse: connect back to malicious server via network and present remote shell
- Bind: open network port and wait for connections; present shell
- Reuse: re-use existing network connection

2.5 Beyond strings

strcpy() and strcat() were common causes of buffer overflow vulnerabilities.
Idea 1: use strn* family of functions; specify safe amount to copy
strncpy: copies at most len chars from src into dst
strncat(): appends not more than count chars from append
Problems: can still be misused if you calculate max length wrong, strings aren't auto null terminated, doesn't stop programmer from returning local ptrs

2.6 Beyond overwriting return address

If we can only overwrite the frame pointer, the stack still moves to the attacker-supplied address, and we can still make up a fake frame for the caller function with a fake return address

3 Format strings, heap, integers

Function args and locals are stored on the stack (x86-32)
Function args are accessed by providing offsets relative to frame pointer

3.1 printf - reading

Doesn't actually check that percents and args match
Caller: pushes arguments onto stack, pushes pointer to format string on stack
Callee (printf):
parses format string; uses the format string to read corresponding arguments off of stack
What if there are too many or too few args? Problems
%08x -¿ print a word. Just put many of these in your printf and we can start reading args

3.2 printf - writing

%n, writes to a ptr number of chars written so far. Combine %n and reading trick to write to arbitrary addresses (you use reading tricks to move up stack and hopefully find a ptr that you can then apply %n on and exploit)

3.3 Heap vulnerabilities

Heap has data structures it manipulates to keep track of free and allocated memory
If you overwrite these data structures, the heap's memory management code will still use them as though they were valid
Heap manager basics:

- Heap manager maintains contiguous chunks of free memory
- Heap layout evolves when malloc() and free() functions are called
- Chunks are stored in doubly linked list (bins, roughly grouped by chunk size)
- When a new chunk becomes free, its inserted into one of these lists

- When a chunk gets allocated, it is removed from the list

Heap chunk:

- Variable size units of memory on heap
- Either free or in use
- User data + heap metadata

Heap metadata:

- Size: chunk size
- N Flag: non main arena
- M flag: is mmapped
- P: Previous chunk is in use

In use chunk:
malloc returns ptr to start of data block
Free can release chunk by looking at metadata in word before data block
Free chunks:

- Kept in doubly-linked list (bins), some of the unused data area of each free chunk is used to store forward and back ptrs (fd, bk)

- Consecutive free chunks are coalesced (no two free chunks can be adjacent to each other)
- Last word of unused data (first word of next chunk) contains a copy of size of free chunk (makes it easy to coalesce adjacent chunk; when freeing, if prev flag indicates the previous chunk is free, then we just look one word above to coalesce the chunk)

Chunks are inserted into free list when they are freed. Chunks are removed when they get allocated (or if they need to be combined with a newly-freed adjacent chunk)

3.3.1 heap corruption with unlink macro

If attacker can control whats in the fd and bk fields of a free chunk, we can control FD-¿bk with a value we control.
Write an attacker chosen value (P-¿bk) to an attacker-chosen address (P-¿fd_¿back)

3.3.2 Heap vulnerabilities via memory management errors

Use after free
Can access memory after its been freed and do weird things
Double free

- Can corrupt heap data structures by doubly freeing a ptr
- free() will happily add same chunk onto free list multiple times; so we can essentially allocate views of the same memory with malloc
- Frees are not obvious in multithreaded contexts

3.4 Integer overflow

Conversion is very complicated
Overflows can allow us to insert a big value but get a small value (allowing us to assign values we're not supposed to, in port example)

4 Control Flow Defenses

Stack canaries:
Detect overwriting of return adddress by placing special value between ocal variables and saved frame pointer. Check this value before popping saved frame pointer and return address from stack
Calling a func:
Caller:

- Pass arguments
- Call and save return address

Callee:

- Save old frame pointer
- Set frame pointer = stack pointer
- Allocate stack space for local storage + space for canary
- Pus hcanary

Returning from a func:
Callee:

- Check canary against global 'gold' copy (jump to exception handler if diff)

- Pop local storage (set stack ptr = frame ptr)
- pop framep tr
- pop return address and return

Caller:

- Pop arguments

What should canary be?
Terminator canary:
Hard to insert via string functions (easy to guess, though, and possible to overwrite when not using string functions)
Random canaries:
Secure as long as they remain secret

4.1 Specifying when we add canaries

- -fstack-protector (functions with character buffers ¿= ssp-buffer size (default is 8), or functions with variable sized allocas)
- -fstack-protector-strong (fucntions with local arrays of any size/type,a nd functions with references to local stack vars)
- -fstack-protector-all (all functions)

More protection is costly

4.2 Bypassing stack canaries

Overwrite with correct value

- Terminator canaries stop string functions, but not memcpy
- Brute force for random canaries. In servers, if any workers die, we fork a new one. But since forked process has same memory layouts and contents as parent, it also has same canary, and so fork on crash lets us try different canaries
- Information leak: printf, etc...

Additionally, local variables can still be overwritten (auth metadta, data pointers, function pointers, and so we might not even need to bypass the stack canary
Pointer subterfuge:

return ptr saved ebp canary ptr val buf[0-3]	Overflow buffer; change val to be return address we want. Change

ptr to point to return pointers address. If the code sets ptr's dereference val to val, then we have overwritten the return address without going past the canary
More mitiations

- Reorder local vars, place buffers closer to canaries.
- For args, copy them to the top of the stack too make overwriting them via local variables less likely

4.3 Non-executable stack/DEP

Pros:

- Little-no performance impact
- No changes to application software

Cons:

- Needs hardware support
- Doesn't automatically work with certain programming tricks (jit compilation which browers use heavily, self–modifying code)

Assumes that if attackers can't inject code, we deny them the ability to execute malicious code. But we can bypass this by repurposing useful executable code (libc, ROP), and many use cases make DEP impossible to use.

4.4 Randomizing shell code location with random stack base or ASLR

Random stack base
Add random offset. Can bypass with NOP sleds guesses, nop sleds or shellcode on heap
ASLR Randomizes other sections of memory
Tradeoffs: requires compiler, linker, and loader support. Increases code size with perf overhead, needs a random number generator, and has load time impacts for shared libraries
Bypasses: information leak, heap spraying

5 ROP and CFI

5.1 Code reuse attacks

Overwrite return address to point to start of system()
Place address of "/bin/sh" on stack so that system() uses it as an argument

(also want to push exit() on stack for graceful shutdown)
Attackers can move shellcode to unrpotected memory, change permission on stack pages (mprotect()), etc... by calling available functions with params of their choosing

5.2 ROP

Make complex shellcode out of existing application code
ROP Gadgets: set of code sequences ending in ret sequence (wahtever ends in 0xC3)
x86 has variable length instructions, so we can alter the pop/push instructions to be different here, creating new code sequences

5.3 Control flow integrity

Direct flow control transfer:

- Advancing to next sequential instruction
- Jumping to an address hard coded in instruction
- static in code

Indirect flow transfer:

- Jumping to an address in reg or memory
- Forward path: indirect calls and branches (functions you are calling)
- Reverse path: return addresses on stack

Create a graph of legitimate control flow transfer. Focus on protecting indirect transfer of control flow instructions
Restrict all control transfers to control flow graph. Assign labels to all indirect jumps and their targets. Before taking an indirect jump, validate that target label matches jump site

5.3.1 Fine grained CFI

We statically compute the CFG and dynamically ensure program never deviates from it

- Assign label to each target of indirect transfer
- Instrument indirect transfers to comprae label of destination with expected label to ensure it's valid
- Note, returns are aliased (can return to multiple places, even if not correct)
- Combined with idea of shadow stack to ensure control data is protected

5.4 Shadow stack

On function entry, sav a shadow copy of function call control flow data into another location
On function exit, compare version on stack to shadow copy
Requires hardware and compiler support

5.5 Coarse-grained CFI

Tradeoff precision for speed

- Identify if control transfer is clearly wrong, not that it is right (can only transfer control to a legitimate destination in CFG, but don't check which path)
- Only two labrels, no shadow stack

Label for destination of indirect calls (forward) (make sure every indirect call lands on a function entry)
Label for destination of rets and indirect jumps (reverse) - Make sure every indirect jumop lands at start of a basic block

5.6 CFI tradeoffs and bypass opportunities

- Overhead (overhead at runtime, longer binaries, hw support needed for viable performance)
- Scope (data isn't protected. Does not protect against interpreters. Needs reliable DEP)
- Precision (can still create gadgets if you don't validate all data-dependent control transfers i.e can still call system or exploit return paths)
- Performance/precision tradeoff creates holes
- V-tables complicate things

6 IPC, User Kernel and VM

6.1 Null referencing

The whole system used to crash on null ref (no memory protection, no protected kernel, sensitive interrupt vector table stored at address zero). Nowadays, this doesn't happen (process isolation and VM)

6.2 Secure design principles:

- Least privilege
- Prvilege seperation
- Complete mediation (check every access that crosses a trust boundary)
- Defense in depth (many security mechanisms)
- Simple designs preferred

6.3 Interfaces

Every interface in a system is a potential trust boundary

- Processor defined interfaces: memory reference, privileged instructions
- Software defined interfaces: system calls, file accesses, network emssages

We need to seperate functionality appropriately (least privilege and privilege separation), check access across trust boundaries (complete mediation), and have safe ways to increase and decrease privilege where needed
Example: Web browser architecture

- Browser process (handles privileged parts of browser)
- Renderer processes (handles untrusted attacker content)
- Communication restricted to RPC to browser/GPU process

6.4 OS Security

Process abstraction

- Each user can have one or more processes
- Processes have UIDs (user ids) that indicate what they're allowed to access

Process isolation

- Keep processes from touching each other's memory or state directly

User/kernel privilege separation

- Limit privileged operations to OS kernel
- check reqs from user against security policy
- Protect OS kernel from user processes

6.5 Unix Permissions

- Permissions in UNIX granted according to UID (process may access files, network sockets, etc...)
- Each process has a user id (UID). Special root user has UID 0 (can access any file)
- Each file has an ACL (grants permissions to users according to UIDs adnd roles; everything is a file)

But for things like passwd (program to change passwords) to work, pocesses have two UIDs. A real user ID (RUID) and effective user ID
Real user id: same as user id of parent process. Determines which user started the process
Effective user id: determines current permissions for process. Can be temporarily different from RUID
setuid programs: Program can have a setuid bit set in its permissions. If so, the caller's EUID is set to the UID of the file (temporary privilege elevation)

6.6 Virtual Memory

Each process gets its own virtual address space. Primary security mechanism for isolating processes from each other

6.6.1 Address translation

mechanism for mapping virtual to physical addresses
Provides:

- Isolation
- Memory access polymorphism (different access implementations for different memory regions/types, i.e. access controls (read, write, execute, etc...))

Translation review: (add offset specified in current part of vpn to translation table register, then get new address and move on to next level, repeat until we reach a page)
For multi-level page tables, some bits are used for the offset, then some bits are used for the page lookup, and then the rest aren't used
TLB: caches recently translated pages (before translating an address, the processor checks the TLB, which is done in parallel with memory cache lookup)
Not everything within a process' virtual address space is equally accessible. Page descriptors contain additional access control information (read/write/execute perms)

6.7 OS Security

Attack surfaces of OS: memory access, privileged instruction, sys call and faults, device accesses. Needs hardware (instruction granularity) and software (interfaces and abstractions) protection

6.7.1 Privilege levels

Multiple privilege levels (processor states)
Just two used by OS typically:

- privileged and non-privileged
- Kernel mode and user mode
- Supervisor and normal

Intel privilege levels (4 rings; 2 used by OS; ring 0 most privileged. Ring 3 is least (user programs), and overtime there have been new levels added that are more privileged than ring 0)
ARM privilege levels:
2 Worlds (secure and non-secure)
4 exception levels (2 used by OS, EL0 least privileged)
Boundary between privilege levels is a trust boundary (any cross privilege interface is part of attack surface)
Dedicated mechanisms for safely changing privilege is needed (anyone can drop; elevating is harder)
Elevating privilege levels is done with sys calls

6.7.2 Sys calls

User-mode processes may need frequent assitance from kernel (I/O, sys info, process control)
Kernel has its own page table (for its code and data, and also maintains page tables for all other processes)
Switching between two usermode processes requires switching between respective process adress spaces (which may require flushing TLBs, which is slow. Since these aren't so frequent its not that bad)
To make system calls fast, kernel's virtual memory space is mapped into every process but is made inaccessible when in user mode
When a process makes a sys call and transfers control to kernel, kernel mem space is already mapped so we don't need to change the tlb base register.
On a process switch, userland page table is swapped (TLB is updated, but all processes share same kernel mapping)
Kernel Threat Model
Confidentiality and integrity of kernel memory and control flow must be protected from compromise by usermode processes. Al user mode processes are

untrusted and potentially malicious
Kernel Operating Model
Usermode processes make frequent calls into kernel, with data passing between them
Kernel must be careful to keep track of whether it is operating on kernel data or user mode data (avoid becoming confused deputy i.e. being manipulated into abusing its privileges)
A usermode process may trick kernel into writing attacker-controlled data into kernel memory or leaking kernel memory to attacker

6.7.3 Return to null dereference

Assume attacker is a userland process trying to attack the kernel (elevation of privilege)
If process mapped page 0 and process manages to trigger a null pointer dereference in the kernel, then instead of crashing, the kernel will use attacker-controlled data on page 0 (return to user attack)
To guard against this, we prevent unprivileged allocation of page 0 up to some minimum (but can be bypassed by making a bigger buffer)
Virtualization: applies isolation between VMs (hypervisor acts as OS for oses)

7 Side Channels

Goal: guess information without direct access to it through side effects
Covert channels: side channels built on purpose
Mitigating side channels:
Eliminate dependency on secret data, make everything the same, hide (operate on encrypted data, not clear form), add noise (easily defeated)

7.1 Cache side channels

Cache is a shared system resource (not isolated). This doesn't impact contents, but does impact time that a read takes)
Cache thread model:

- Attacker and victim are two different execution domains on same physical system
- Attacker is able to invoke (directly or indirectly) functionality exposed by victim (i.e. with API, making sys call, etc...)
- Do not want attacker to infer anything about victim's memory

But some algorithms have memory access patterns that are dependent on sensitive memory contents (memory access patterns can leak memory contents)

7.1.1 What an attacker can do

- Prime (place known addr in cache)
- Evict (access memroy until given address is no longer cached)
- Flush (remove a given address from cache)
- Measure (measure how long it takes to do something)

7.1.2 Cache side channels

- Evict and time (kick stuff out of cache and see if victim slows down)
- Prime and probe (put stuff in cache, run victim, and see if accesses are still fast (no conflict) or slow down (have been displaced by victim)
- Flush and reload (flush particular line from cache, run victim, and see if accesses are still fast)

Lots of error in individual measurements, need to run many times to get useful values. Works on any levels of cache, but need to be wary of time domains

7.2 Rowhammer

SRAM: retains bit value as long as power is on
DRAM: requires periodic refresh to maintain stored value

7.2.1 DRAM

Cells are grouped into rows (1kb per row). All cells in a row are refreshed together. As DRAM gets smaller, there is more reliability issues
Repeatedly opening and closing a row enough times induces disturbance errors in adjacent rows
Exploit sketch: Identify protected target data to flip. Maneuver protected data over flipping location. Then hammer own memory locations to alter protected data
Random bit flips can be used to semi-deterministically take over computer
Soltuions:
ECC memory (compute error correcting code on write, check on read; can be bypassed, adds cost), process changes to mitigate inter-row effects, memory controller limitations on hammering or additional adjacent line refresh (reduces performance)

7.3 Meltdown and spectre

Instruction pipelining: Processors break up instructions into smaller parts (so they can be parallelized)
Out-of-order execution: Some instructions can be safely executed in a different order than they appear, avoiding pipeline stalls
Speculative execution:
Sometimes control flow depends on output of earlier instruction. Rather than wait to know which way to go, procssor can speculate about direction/target of a branch. This affects the cache
Meltdown:
Cache reads and privilege checks are done in parallel, allowing you to read memory before getting access (just a bug)
Spectre:
Mispredictions + cache side effects

8 Web Security

Web works via client-server architecture
Browser makes reqs and handle responses by:

- Rendering HTML+CSS
- Executing JS
- Invoking plugins

Server:

- Can send HTML, JS, static assets, etc...

8.1 HTTP Protocol:

Client and servers communicate by exchanging individual messages (not stream of messages; built upon TCP)
HTTP Versions (provide efficiency improvements):

- HTTP2 (47% of web): Message multiplexing (multiple reqs over one tcp conn), header compression, server push (preemptively sending resources)
- HTTP3 (36% of web): Uses QUIC instead of TCP

8.2 Cookies

HTTP is a stateless protocol. Session information managed by cookies. WEb server provides cookies in response to server (set cookie header). Browser attaches cookies to every subsequent request

- Session cookies: expiration prop not set, exist only during current browser session, deleted when browser exits (unless configured otherwise)
- Persistent cookies: saved until expiration time

8.3 Browser execution model

Each browser tab (separate process usually) loads content, parses html and runs js, fetches sub resources, and responds to events like onClick, onMouseOver, onLoad, and setTimeout.
Browser tabs can also have nested execution frames

8.3.1 Frames

- Delegate screen area to other source
- Browser isolates frames from page
- Parent may work even if frame is broken

Communication with frames done via postMessage API (frame must have event listeners set up for this to work)

8.4 Document Object Model

Contains web content (elements, but not executable js), browser object model (i.e. cookies, history, navigation, access window), etc...
JS changes DOM over time

8.5 Browser threat models

- Network attacker (intercept traffic)
- Web attacker (host suspicious website i.e. phising)
- Gadget attacker (inject limited content into honest page via frames, scripts, etc...)

8.5.1 Isolation

Browsers isolate tabs (different processes; so diff UIDs and address spaces) and uses the same origin policy to isolate cookies and local storage

8.6 Same Origin Policy

Origin: trust boundary of web ((scheme,domain,port) triple derived from URL).
If you come from same places, you must be authorized (fate sharing)
Goals: Isolate content of different origins, providing confidentiality (script from evil.com can't read bank.ch) and integrity (script from evil.com can't modify bank.ch)

8.6.1 DOM sop:

- Each frame in a window has its own origin; frames can only access state with the same origin
- SOP does not prevent requests across pages (i.e. you can still make a post request, just can't read the result, but this is still dangerous)

We can load cross origin HTML in frames, but can't inspect or modify the frame content

8.6.2 Scripts

We can load scripts across origins (libraries)
Scripts execute with privilege of page
Page can see source (js has reflection)

8.6.3 Images

Browser renders cross-origin images, but can't inspect individual pixels (can see other properties like width/height)

8.6.4 Cookies

Cookies use a separate definition of origin ([scheme], domain, path)
Server can set domain property for ANY cookie
What cookies can a website set?

- Any domain that is a domain-suffix of the cookie domain

- Path does not matter

How do we decide to send cookies? (distinct; this is based on what we set, not what's possible)

- Cookie domain must be a suffix of the URL domain
- Cookie path must be a prefix of path for it to send

Caveats:

- External scripts can still access cookies (on same origin; its just in the tab)
- SOP doesn't prevent data leakage (i.e. via url)

HTTPOnly cookies:
Don't expose cookie to JS via document.cookie
Problem: You don't need to know the cookie to use it; you can just make a request by URL (CSRF)
SameSite cookies:

- Srict: Send cookies only when request originates from same site (i.e. gmail link navigation cannot send cookies)
- Lax: Send cookie on safe top-level navigations (default)
- None: self-explanatory

8.7 CSRF

8.7.1 By requests

When a website issues an HTTP get request, it attaches all cookies associated with the target website

- If a user clicked a link (website, email)
- If another page embedded the target page in an iframe
- If a client-side script issues the request

What matters is the target of the request, not the originator
Only the target site sees the cookie, but has no idea if request was authorized by the user (side effects still pass!)

8.7.2 Drive by Phishing

CSFR isn't just about cookies/auth; can happen anywhere browser has privileged access **Example:** JS scans network looking for broadband network. Once network is found, try to login with default router passwords

8.7.3 Defenses

- Secret validation token (per session, in vulnerable forms, SOP prevents bad actors from making request to see token, but hard to implement everywhere correctly)
- Referer/origin validation (server looks at referer, which is page before current page, and origin, which is requesting page, to see if request should be made)
- SameSite=Strict cookies (prevents cookies from being sent across origins). SameSite=Lax prevents problems for most POST requests, but GET requests can have side effects, so can still be problematic

8.7.4 Fetch metadata

Most recent and comprehensive solution, contains context for current user session but is poorly supported

- Sec-Fetch-Site: cross-site,same-origin,same-site,none (who is making the request)
- Sec-Fetch-Mode: navigate, cors, no-cors, same-origin, websocket (what kind of request)
- Sec-Fetch-User: 1 (did user initiate request?)
- Sec-Fetch-Dest: audio,document,font,script (where does response end up?)

8.8 Command injection

When you take user input data and allow it to be passed on to a program/system that will interpret it as a code:

- Shell
- Database

Similar to our low level vulnerabilities, but this level of abstraction is higher (interpreted code vs. compiled code)

8.8.1 Via Shell

If a server takes in a user-string and appends that directly to a command, the user can escape out of the current command (with) and run arbitrary code. Most high level languages have safe ways of calling out to shell, but problems still persist
Prevention

- Blocklists (block symbols)
- Allowlists (only allow what you need)
- Don't use the shell for commands

8.8.2 Via SQL

If server uses raw user input while creating queries, very bad things can happen (they can use –, end quotes, use semicolons to run another queyr, etc...)
Solutions

- Input sanitation (very difficult)
- Escaping (very difficult; and if not using strings, user can still exploit lack of typing to do things inserting or 1=1)
- Parameterized SQL (safe)
- ORMS

8.9 Cross Site Scripting (XSS)

Malicious content is injected via URL encoding (query parameters, form submission) and reflected back by server in response. Browser then executes code that server provided (i.e. form that takes html, stores it in db, and reflects back to user).

8.9.1 Prevention

- Filtering: i.e. blocking angle brackets, blocking tags, etc.... Very difficult to do; filter evasion tricks can break this very easily
- Content Security Policy

8.9.2 Content Security Policy

Browser will only execute scripts loaded in src files received from whitelisted domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes)
Examples:

can only be loaded from same domain; no inline scripts: default-src 'self'

- allow images from any domain; restrict audio/video to trusted providers; no inline scripts: default-src 'self'; img-src *; media-src media1.com; script-src userscripts.example.com

Administrator serves Content Security Policy via http header or meta html object (¡meta http-equiv="Content-Security-Policy" content="..."¿ tag)
CSP also defends against click jacking (can't load these transparent click overlays/bad elements elements because of CSP)

8.9.3 IDOR attacks

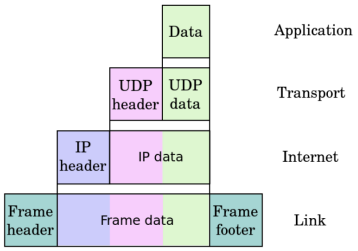
Sometimes, objects like parameters in a URL are encoded very simply (i.e. base64, guessable adjacent ids, etc...)
In this case (i.e. ?role=base64(user) vs. ?role=base64(admin)) we can very easily change our roles.

9 Network Security

Packets: internet constructed of discrete, self-addressed, chunks of data: packets

- Application Layer (HTTP, SMTP)
- Transport Layer: Port-addressed host-to-host communications (on LAN or WAN)
  - Fragmentation, reassembly, and end-to-end (across network boundaries)
  - Provides a uniform interface that hides underlying network topology
- Internet Layer (IP): Transmission of data frames within a local network (without intervening routers)
- Link Layer: Transmission of raw bits over a physical data link connecting two devices (1000BASE-T, 5GHZ ODFM WiFi)

9.1 The Network Stack



9.1.1 Application Layer

Turns click into GET Request

9.1.2 Name Resolution

Ask local DNS server for ip domain name

9.1.3 Transport Layer (TCP)

Break message into pieces (TCP Segments)
Deliver pieces reliably and in order

9.1.4 Network Layer: IP Addressing

Address each packet so it can traverse network and arrive at host
Addresses are generally globally unique

9.1.5 Datalink Layer (ethernet)

- Media Access Control (MAC): Can I send it now? What local hop do I take next? (next physical device)
- Send individual frames on a link

9.2 IP Layer:

Separate physical networks communicate to form a single logical network (IP addresses globally unique)

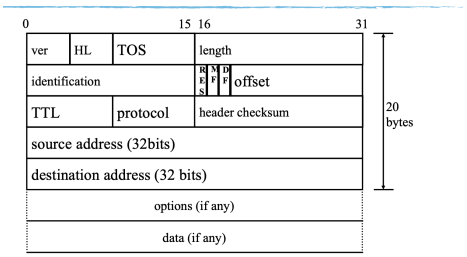


Figure 1: IPV4 Packet Header

IP Protocol functions:

- Routing: Your IP host knows the local router (gateway). IP gateway must know the routes to other networks (next hop). Packets usually take multiple hops to get to their destination.
- Addresses are globally meaningful
- Error reporting: send Internet Control Message Protocol (ICMP) back to source if there was a problem
- Fragmentation and reassembly (if max-packet-size on next hop link ¡ user-data-size in IP packet)
- TTL field: decremented after every hop (packet dropped if TTL=0). Prevents infinite loops

9.2.1 Fragmentation:

Sender writes unique value in identification field
If router fragments packet it copies this id into each fragment
Offset field indicates position of fragment in bytes (offset 0 is first)

- MoreFragments flag indicates that this isn't the last fragment
- DataFragment flag tells gateway not to fragment

All routers must support 576 byte IPv4 packets (MTU)

9.2.2 TTL:

How many more routers can packet pass through?
Decremented every hop. If TTL=0, router discards packet
Limits packet from looping forever

9.3 Transport Layer (TCP)

TCP provides reliable, ordered delivery of bytes
Establishes stateful, bi-directional session between two IP:Port endpoints
Each side maintains:

- Sequence number: sequence base (i.e. start from here) + num of bytes sent (sent first)
- Acknowledgment number: acknowledgment base + num of bytes received (sent back after receiving)

Special packet flags:

- SYN: I want to start a connection
- FIN: I want to shut down a connection
- RST: We are killing this connection now

Ports + IP identify a unique connection

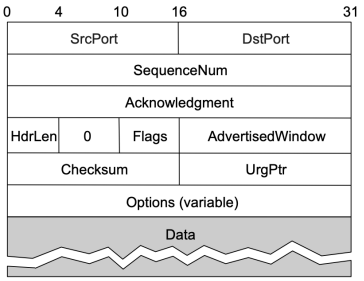


Figure 2: TCP Header

9.3.1 Three way handshake:

Client sends a SYN packet with sequence number = x
Server sends a SYN + ACK packet where SYN = x + 1 and ACK = y + 1
Client sends Ack = y + 1 back (after handshake, we use actual byte counts, not 1)

9.3.2 Security

Trusted network and hosts
End-to-end principle (intelligence at edges, network is simple)

Robustness Principle (an implementation is conservative in its sending behavior,liberal in its receiving behavior)
Built-in trust assumptions about networks being used only as intended and hosts being trusted admins, which is obviously not true. In the 80s, we didn't trust the hosts but trusted the network. Now, we don't trust the hosts or the network (all things can be compromised). Security provided at application layer
Attacker Models

- Person in the middle: can see, block, and modify traffic (attacker controls ap)
- Passive: attacker has passive tap or recordeded traces
- Off-path: attacker can inject traffic into network (anyone with access to network)

Who can see packets you send?

- Network (routers, switches, aps)
- Unprotected wifi network (everyone within range)

- Non-switched (i.e. old school) Ethernet: everyone on the same network
- Switched Ethernet: everyone gets their own link, but, sometimes someone can intercept your traffic

TCP/IP offer no authentication

- Attacker with direct access to network (including PitM) can spoof source address
- Connectionless protocols (i.e. UDP) especially vulnerable
- Can blast packets at a target by using spoofed source address, make it look like someone else is attacking them (Denial-Of-Service)
- Can try to interfere with existing communications between hosts (by injecting packets that seem to be part of the communication)
- Cant count on source address for authentication

9.4 Link Layer

Physical channel often shared by multiple hosts on local network (open wifi, non-switched ethernet)
Link layer controls physical medium:

- Also known as Media Access Control Layer
- Makes sure each host only gets frames addressed to it
- Each host is responsible for picking up only frames addressed to it and ignoring other ones (honor system)
- Filtering happens on network card (only frames addressed to this host are parsed/passed onto layer above)

Ways to spoof:

- Host configuration
- Address binding (IP to MAC)
- Routing

Example: Sending packet to 8.8.8.8
Local (check subnet masks of network): Send directly
Not local: Send to default gateway (router)

9.5 DHCP

Contains subnet mask, address of router, device IP, netmask, address of DNS, etc...
Host broadcasts DHCP Discover on local network (special broadcast address)
DHCP server responds with info for your host
DHCP Spoofing:
Someone listens for DHCP requests and sends responses (can tell someone to use router of their choosing)
One defense, DHCP Snooping: network switch configured to block DHCP messages from non-trusted hosts that aren't known to be DHCP servers

9.6 Network Routing

Ethernet frame contains MAC header, data, and checksum
Host needs to fill an ethernet destination address

- MAC adress of host on local network

- MAC address of gate for host not on local network

How to find Ethernet address from an IP address?

9.6.1 Address Resolution Protocol

- used to query hosts on local network to get link-layer (MAC) address for an IP address
- e.g. Alice broadcasts a message asking for MAC address of certain ip. Bob sees message and replies with MAC address. Alice sends IP packet for address in an ethernet frame

Note, anyone can send ARP replies
ARP Spoofing:
Attacker on network can impersonate any host
Mitigations:

- Static ARP tables (infeasible)
- Port-binding: Restrict MAC and IP-addresses to single port on switch at a time. First mover wins. Does not work well on WiFi.
- Depend on higher level to save you (i.e. SSH/TLS)

9.7 Pathway:

Want to send packet to 8.8.8.8

- Is host on local network? Send directly? Else, send via default gateway
- Create IP Packet
- Create and send link-layer frame
- Gateway picks next router and forwards the IP packet

9.8 Border Gateway Protocol:

- Manages IP routing between networks on the internet
- Each BGP node maintains a set of trusted neighbors
- Neighbors share routing information (who can reach who via where)
- No authorization: malicious BGP nodes may provide incorrect routing information that redirects IP traffic

Can be hijacked easily

9.9 IP Spoofing Attacks

No authentication in link/IP layers

- Eve can spoof Alice's IP; can claim to send packets claiming to be Alice. Eve may not be able to recieve packets addressed to Alice
- UDP: trivial, since stateless
- TCP: more complicated; still possible. Two endpoints share state (syn/ack). Attacker must be able to guess it)

9.10 Blind Port Scanning:

Attackers would like to know which TCP services are offered on a host
Port scan:

- Find a zombie host. Make request with zombie and check SYN number
- Send TCP SYN to port number on host with spoofed IP of zombie host
- Make another request with zombie and check SYN number (see if it incremented more than 1; tells you other service responded)

9.11 DNS

Instead of remembering 32-bit ips, we use domain names. The domain name system (previously big file; now distributed name servers) turns domain names into ips
Types of servers:
13 DNS root name servers:

- Hardcoded into all systems
- Chosen at random

Authoritative name servers for sub domains
Local name resolves (recursive resolvers):
Contact authoritative servers when they don't know a name

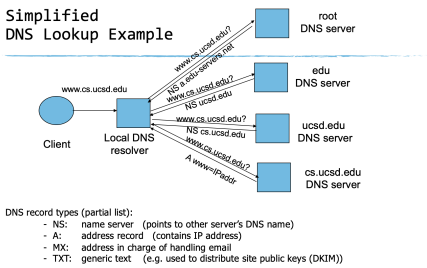


Figure 3: Simplified dns lookup, uncached

DNS queries (useful for finding name servers and IP addresses) and negative DNS queries are cached (saves time for non-existent sites)
Cached data periodically times out (with TTL controlled by owner of data)

9.11.1 DNS Cache Posioning

Convince DNS to cache a bad mapping

- Person in the middle: Listen to DNS resolver for requests. Send false response and block true response
- Passive: Listen to DNS resolver for requests. Send false response faster than true response (true response ignored)
- Off-path: Flood DNS resolver with responses blindly when you know user is likely to make request (problem: matching request and response)

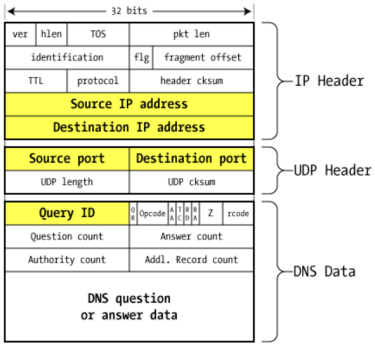


Figure 4: DNS packet

Query id: 16-bit random value that links response to query
Response echoes query id and gives some glue information (i.e. name of authoritative name servers and ip address of these servers)
The glue records are problematic. Can put some false mappings in the additional information section for unrelated websites (can find ip for amazon.com at this fake address)

9.11.2 Balliwick checking and Kaminsky Attack

Balliwick checking:
Response only cached if within same domain as query
Kaminsky attack:

- Attacker issues dns query for prefix + legit website (fake prefix that points nowhere)
- Attacker immediately sends fake responses to query, each with different query ID and false ip to paypal.com
- Attacker 1 repeats this with different prefixes until cache is poisoned

Defenses:

- Increase query id size: randomize src port and ensure it matches, adds 11 bits
- 0x20 encoding: randomly vary capitalization (DNS case insensitive) and ensure you get same capitalizaiton back
- Try to detect poisoning
- Rate limit queries
- Authenticated reqs/responses (provided by DNSSec)

DNS isn't robustly secure. Important for HTTPS to check signatures

9.12 Network Perimeter Defense

Network defenses on outside of org (between org and internet)

9.12.1 Firewalls

Types:

- Personal firewalls: run at end hosts, more app/user specific info
- Network firewalls: intercept and evaluate communications from many hosts
- Mediates all communication

Filtering strategies:

- Packet filters: operates by filtering on packet headers content
- Proxy-based: operates on level of application (e.g. HTTP web proxy)

9.12.2 Access control policies

What firewall enforces. Checks who is allowed to talk to whom and access what services
Distinguishes between inbound and outbound traffic

- Inbound: attempt from external user to connect to services on internal machines
- Outbound: attempt from internal user to connect to external services

Default allow: Permit all services and specify which to block
Default deny: Deny all services and specify which to allow. Safer, but assumes how many services there are (outbound traffic is usually default allow because of this)

9.12.3 Packet filtering firewalls:

Define list of access-control rules. Check every packet against rules and forward or drop
Packet filtering can take advantage of following information from network and transport layer headers:

- Source IP
- Destination IP
- Source Port
- Destination Port

- Flags (e.g. ACK)

Ports distinguish services on a machine
Note, these rules are stateless. Some firewalls maintain state about open TCP connections

9.12.4 Proxy-based firewalls:

- Proxy acts as both client and server. Note, must terminate connections, complicates HTTPS/TLS, so must take out certificates

- Can filer out using application-level info (block based on url, js, etc...)

- Proxies can provide other services (caching, load balancing, etc...)

9.12.5 Firewall pro/cons:

Reduces attack surface, filters out noise, and reduces liability, but provides false sense of security, makes a bottleneck and single point of failure, and costs money

9.12.6 Network content analysis

Lots of devices want to look at network traffic content for security (spam filters, NIPS/NIDS, data leakage, traffic differentiation for throttling services). Doing at endpoint is cheaper
Network Evasion Many network intrusion detection systems deployed at endpoints; assumption is that they see same traffic as host
However, we can send packets with different TTLs (so host sees bad message while NIDS doesn't)
Defenses:

- Protocol normalization: rewrite TTLs to be uniform, don't allow overlapping packets

9.13 Denial of Service

Ways:

- Exploiting logic vulnerabilities: buffer overflows, etc...

- Resource consumption: overwhelm with spurious requests (i.e. SYN flood, bandwidth overflow)

Defenses:

- Source address validation: filter packets with clearly bad source addresses. Network egress: filter outbound packets on link whose source addresses are not reached using link as next hop. Network ingress: filter outbound packets whose source addresses are invalid
- SYN cookies: delay allocation of state on server. Server initial sequence number encodes secret cookie that is a function of state. Only allocate state when client sends ACK to servers SYN/ACK (using cookie to validate)
- Packet filtering: filter packets with bad features
- Buy more resources/CDN

Reflection Attack: Spoof source address of victim (send DNS requests on behalf of victim; responses flood victim)

10 Cryptography

Alice and Bob aim to send messages to each other secretly (CIA maintained)
Attacker model:

- Passive attacker: can read only
- Active attacker: can read, create, and possibly modify, block
- Person-in-the-middle: can read, create, modify, or block

10.1 Encryption

Plaintext (m): raw message
Ciphertext (c): encrypted message
One-time pad:
XOR plaintext with a random stream of bits only known to Alice and Bob. Achieves perfect secrecy (for a given ciphertext, every plaintext is equally possible). No integrity or authenticity

10.2 Computational Cryptography

Sharing large secrets impossible; rely on sending smaller secrets. We sacrifice perfect secrecy (so not all plaintexts equally probable) and instead just make it computationally impractical to learn anything aobut the plaintext without the secret
Kerkchoff's principle: A cryptosystem should be secure even if we know everything about the system
Symmetric Cryptography Alice and Bob share a secret key they use to secure their communications.

Asymmetric Cryptography Alice and Bob each have their own private/public key. Private keys used for decryption, public keys used for encryption. **Encryption provides confidentiality, not integrity** Message Authentication Code (symmetric) and Digital Signatures (asymmetric) provide integrity without confidentiality
Cryptographically Secure Pseudo-Random Number Generator:
Given first n bits of a sequence, can't predict (n+1) bit with probability better than $\frac{1}{2}$. Requires system APIs for cryptographic random numbers (cannot just use rand)

10.3 Hashing

Cryptographic hash function maps arbitrary length input into fixed-size string with following properties:

- Pre-image resistance (given hash output, impractical to find input that generated it)
- Collision resistance (impractical to find any two inputs that hash to the same output)

Examples:

- SHA-2: fixed size
- SHA-3: arbitrary size

10.4 Message Authentication Code

Validates message integrity and authenticity based on a shared secret. Function of message and secret key. Impractical to forge without secret key. MACs can be constructed out of hash functions or ciphers

10.5 Symmetric encryption

Stream cipher: generate pseudo-random stream of bits as long as message and xor with message. Insecure if used more than once.
Block cipher: encrypt/decrypt fixed size blocks. Combination of permutation (each input mapped to one output) and substitution (some codewords mapped to other codewords) in multiple rounds (i.e. AES)
Encrypting message shorter than block: padding (must be indistinguishable from plaintext)
Encrypting message longer than block: chaining

10.5.1 Chaining (known as modes of operation)

Electronic Code Block: encrypt each block separetly (very bad; easy to find patterns, but fast)
Cipher Block Chaining: XOR ciphertext into next plaintext; use random initializaiton vector
Counter mode: Encrypt successive counter values and XOR result with plaintext (block cipher becomes stream cipher)
Authenticated encryption: simultaneously provides confidentiality, integrity, and authenticity. Designed to work with a single key. AES-GCM, ChaCha20+Poly1305AES
Limitations:
Requires sharing large keys

10.6 Asymmetric encryption

Public key known to everyone, private key kept secret
Confidentiality provided via encryption and decryption, integrity and authenticity by signing and verification
Signing and verification: Signing uses private key. Verification uses public key.
Examples:
RSA, ElGamal, DSA
Very expensive vs. symmetric ops. Usually combined with symmetric crypto for performance (use asymmetric to bootstrap one-time secret for symmetric crypto)

10.7 Public-Key Model

Assuming we encrypt and sign a message from Alice to Bob, we can only assume:
Bob knows:

- Alice knows plaintext (and everyone she shared it with)
- Alice (or someone with her private key) signed plaintext at some point

Alice knows:

- Only Bob (or someone with his private key) can extract the plaintext
- Bob (or anyone else) can prove Alice (or someone with her private key) signed the plaintext

10.8 Public-Key Infrastructure

We need a way to retrieve public keys and ensure they are correct
Solution: **Trusted Third parties**

- Alice and Bob exchanged keys with Charlie

- Charlie sends signed message with Alice's key to Bob
- Charlie sends signed message with Bob's key to Alice
- Alice and Bob trust Charlie to send real public keys

But, we can do better
Charlie creates a certificate, signs the certificate with his private key, and gives it to Alice. Alice sends Bob Charlie's certificate. Bob verifies the signature on the certificate. Bob trusts charlie, and accepts the public key from Alice
Two models:

- PGP: Charlie is another person you trust
- Everywhere else: Charlie is a Certificate Authority

10.9 PGP

User attests to trustworthy of another person's public key. People sign your PGP key (meaning Alice claims that this is really Bob's key). Only people who trust Alice can use her signature on Bob's key to be sure it's Bob's key.

10.10 Certificate Authority

Browser has list of public keys of trusted CAs. Includes expiration time, limitations on usage, issuing CA, and subject (name, domain)

10.10.1 TLS

- Client makes hello
- Server makes hello, sending certificate
- Client verifies certificate against CA public key (from browser list). Then, extract public key for encryption.
- Use public key to encrypt symmetric session key
- Use session key to encrypt session and HMAC for integrity

TLS ensures authenticity, confidentiality, and integrity

10.11 Certificate Revocation

CA's and PGP's support certificate revocation. In PGP, only Alice can revoke her own key (she signs revocation with her private key; others can verify this). In CA, Alice asks CA to revoke the certificate.
CA Revocation mechanisms:

- Certificate Revocation List: Certificate says where to get CRL. Client periodically gets and updates CRLs
- Online Certificate Status Protocol: Query CA about status of cert before using it. OCSP Stapling: Server includes recent OCSP status (signed by CA) along with certificate.

Browsers mostly don't check CRL or OCSP (instead Google harvests revoked certs and sends them to users)

10.12 CDNs:

Complicate things
Subject alternate name: authorizes CDN to speak on behalf of website

11 User Authentication

- Passwords: Something you know
- Tokens: Something you have
- Biometrics: Something you are

11.1 Passwords

Strong passwords Require symbols, length, expiration time etc.... Unintended consequence is people use the same strong passwords, inconvenience, predictable patterns, reuse, making small changes **Phishing** Very dangerous. Requires users to know domains associated with entity, and even then attackers can use homoglyphs. Trusted path is one way someone on windows always knows they are using the windows login path.
Password attacks

- Keyloggers
- Passwords in memory
- Stored passwords
- Monitoring unsecure transmission channels

- Password oracle (logging in with different passwords, looking for errors)
- Database leaks

Verifying password is correct Use a hash of password. Compare hashed attempt vs. hash in database. Never store raw passwords
Problems:

- Dictionary attacks: Try every string in dictionary until correct entry is found
- Can precompute hashes of all strings in dictionary, then do reverse lookups

Salts: Make precomputation impossible by hashing salt (random number) concatenated with password (still not impossible to crack)
Additionally, use very hard hash function
Passkeys are a replacement that remove user involvement (challenge response protocl using per-site public key pair). Weakly supported and hard to share

11.2 Smartcards:

Secret key embedded in computer. Interrogate with random challenge and verify signed response

11.3 One time password tokens:

Same idea (tiny computer with secret). Generates token periodically that user enters into website. Makes password not enough, is not predictable, and is single-use, but doesn't scale well to multiple accounts and vulnerable to phishing or man in the middle attacks. Server also needs to know secret key to validate tokens (single point of failure)

11.4 USB/NFC key

Same idea but removes the human involvement. No MITM risk. But not standardized

11.5 One time passwords without tokens:

Same but virtual. Factors lose some isolation, but scales better. Can be done via SMS, which is not very secure (sim-swapping)

11.6 Biometrics

Not transferrable, nothing to remember, very strong differentiator
Approach:

- Scan an analog sample
- Convert to set of digital features
- On enrollment save template of identifiable features
- On auth, do fuzzy match against saved features

Biometrics can require multiple samples. Usually try to avoid duplicate samples (could be man in the middle).
What does accuracy mean?

- False accept rate: lower FAR harder to attack
- False positive rate: lower FPR easier to use

12 Malware and Botnets

Any kind of malicious software **Virus Attachment** Write virus at end, set up jump at start of program to virus, jump back after virus replicates
Detecting Malware

- Scan signatures
- Validate programs on machine against list of known good hashes of executable
- Behavior detection

12.1 Encrypted Viruses

Viruses have small decryption loop that runs first, decrypts virus body, then launches body. Each time virus infects new file, it changes encryption key so virus looks different
Polymorphic Virus: Changes decryption algorithm with similar instructions (very difficult to detect)

12.2 Generic decryption:

Let malware do the hard decryption for you (emulate code execution until memory decrypts itself; check for signatures in memory)
Problems:

- How long to emulate for?
- What if virus only activates at certain time/on certain input?
- What if malware knows its running in vm?
- What if theres no signature?

Metamorphic virus: changes ALL code (not just decryptor)

12.3 Integrity Checking:

- change detection: assume programs safe when first installed, periodically check their hash.
- Allowlisting: import list of known good software. Validate all programs on disk hash to something on known good list

Issues: hash list must be well protected, hash list must be comprehensive, doesn't deal well with editable documents, malware doesn't have to be a file (could be in memory)

12.4 Behavioral Detection

- Identify suspicious behaviors (decrypting code, unusual sequences, unusual use of file system or network)
- Track software reputation
- Can run real time or use ML
- Problem: false positives. Forced to tune for low false positives

12.5 Worms

Spread over network (much faster and easier than file sharing viruses)
Ideas:

- How likely is given infection attempt successful?
- How frequently are infections attempted?

What can be done:

- Prevention (firewalls, turning off unused network services, patches)
- Treatment (white worms, reduce spread)
- Containment (rate limiting)

Network telescopes: Monitor unused IPs. If worm scans randomly, will hit telescope repeatedly.
Other ways to share malware:

- Drive-by downloads
- Social engineering
- File sharing networks

12.6 Botnets

Network of compromised computers with common command and control system (C2)
Bot controller sends commands via network to get botnet to do something en masse
Architectures:

- Centralized: old school or web server; multiple control servers for robustness, RR scheduled
- Peer to peer: self-organizing, each host can be a proxy or a worker; decided dynamically; multi-level hierarchy forwards traffic back to controller
- Push: attacker sends messages to workers
- Pull: workers ask for commands

Bots now have auto-update mechanisms, along with resilience.

- RR polling: if you can't talk to C2-a, try C2-b
- Domain generation algorithm: if cannot connect to C2-a, try domain based on time. Later, attacker can register that domain
- Digital signatures on updates