CS136: Computer Security

Professor Reiher

Thilan Tran

Winter 2022

Contents

CS136: Computer Security 3

Principles, Policies, and Tools 6
- Design Principles 6
- Policies 7
- Tools 9

Access Control 10

Cryptography 13
- Cryptanalysis 14
- Symmetric Cryptosystems 14
- Asymmetric Cryptosystems 17
- Checksums and Signatures 19
- Ciphers In Practice 20

Key Management 22
- Key Exchange 23
  - Symmetric Key Exchange 23
  - Public Key Exchange 25
- Key Generation 26
- Key Infrastructures 26

Authentication 29

Operating Systems 33

Network Security 37

Traffic Control Mechanisms 38

CS136: Computer Security

- why is computer security necessary?
  - people may have malicious intents
  - computers handle a lot of money and a lot of important information
  - society is increasingly dependent on correct operation of computers
- there is now a big problem with computer security:
  - only a matter of time before a real disaster
  - companies go out of business due to DDoS attacks
  - identity theft and phishing
  - cyberattacks e.g. Stuxnet
- examples of large scale security problems:
  - malicious code attacks:
    * new viruses, worms, Trojan horses, etc. to create ransomware attacks
    * increasing attacks on infrastructure systems
  - **distributed denial of service (DDoS)** attacks:
    * use large number of compromised machines to attack, one target
    * exploiting vulnerabilities or generating lots of traffic
    * in general form, an extremely hard problem to tackle
  - vulnerabilities in commonly used systems:
    * systems e.g. Android, Windows, iOS, macOS, etc.
    * middleware e.g. Windows Installer, Apache, Node.js
    * even security systems themselves
    * critical hardware flaws in hardware e.g. Intel and AMD processors
  - electronic commerce attacks
    * e.g. identity theft, ransomware, extortion, mining on compromised machines
  - cyberwarfare
    * e.g. Stuxnet, attacks on Ukrainian power grid, cyberspying, Russian election hacking
  - privacy concerns:
    * data mining by the government
    * Facebook, Google, Amazon, etc.
  - **passive threats** are forms of eavesdropping, mostly threats to secrecy
  - **active threats** are more active e.g. destruction or interruption / misuse of services
  - social engineering attacks are also a common effective threat
    * especially phishing
- recent Log4j vulnerability:
  - a programming flaw in a popular package for Java program logging

  - allows attacker to force a server to execute arbitrary remote code
    - essentially end goal of any attacker i.e. "game-over" type of vulnerability
  - not a new or sophisticated type of flaw
- why aren't all computer systems secure?
  - difficult due to hard technical problems
  - as well as cost / benefit issues:
    - security only pays off when there is trouble
    - buyers want 100% effectiveness with 0% overhead, learning, inconvenience
  - ignorance also plays a role
  - also constrained by legacy and retrofitting issues:
    - e.g. core Internet design, popular programming languages, commercial OSs
    - retrofitting security works poorly, considering the history of patching:
      - when to patch, patches are small and near-sighted, not all software will be patched, patches themselves can have vulnerabilities, etc.
      - malware spreads faster than the patching
- why isn't security easy?
  - different than most other problems in CS
    - universe is much more hostile and adversarial, since humans seek to outwit us
  - fundamentally, we want to share secrets in a controlled way
    - classically hard problem in human relations
  - you have to get everything right:
    - any mistake is an opportunity for attackers
    - do we really have to wait for completely bug-free software?
  - computer itself isn't the only point of vulnerability
    - users, programmers, system administrators, supply chain
- how common are software security flaws?
  - about 1500 found per year
    - only considering popular software, real security implications, and publicized flaws
- important definitions:
  - **security** is a policy e.g. no unauthorized user may access this file
  - **protection** is a mechanism e.g. the system checks user identity against access permissions
    - protection mechanisms implement security policies
  - a **vulnerability** is a weakness that can allow an attacker to cause problems
    - most vulnerabilities are never exploited

  - an **exploit** is an actual incident of taking advantage of a vulnerability
  - **trust** or doing certain things for those you trust and not doing them if we do not trust them:
    * how to express trust, how do we know who someone is, what if trust is situational, what if trust changes
    * trust relationships such as transitive trust
      · e.g. peer applications, chained certificates, database used by a web server, code that calls code that calls code
    * most vulnerabilities are based on trust problems
      · taking advantage of misplaced trust
- what are our security goals?
  - confidentiality, integrity, availability
  - involves prevention, detection, or recovery
- what are the categories of threats that security faces?
  - disclosure, deception, disruption, usurpation
- the principle of easiest penetration:
  - an intruder must be expected to use any available means of penetration
    * not necessarily the most obvious or the one against which there is the most solid defense
  - opponents attack where we are weak
- the principle of adequate protection:
  - worthless things need little protection
  - things with timely value need only be protected for a value

Principles, Policies, and Tools

Design Principles

- each principle has its own tradeoffs
- **economy**:
  - security tool must be economical to develop, use, and verify
  - should add little or no overhead
  - keep it small and simple
- **complete mediation**:
  - apply security on every access to a protected object
    - e.g. each read of a file, not just the open
  - check access on everything that could be attacked
- **open design**:
  - don't rely on security through obscurity
    - specifically, secret of how it works vs. secret keys
  - assume all potential attackers know everything about the design
  - obscurity can provide some security, but it's brittle
- **separation of privileges**:
  - provide mechanisms that separate the privileges used for one purpose from those used for another
    - e.g. separate access control on each file, different passwords for every website
  - allows flexibility in security systems
- **least privilege**:
  - give bare minimum access rights required to complete a task
    - e.g. don't give write permissions if program asked for read
  - require another request to perform another type of access
  - extremely important when building complex systems
- **least common mechanism**:
  - avoid sharing parts of the system's mechanism
    - among different users or different parts of the system
  - coupling leads to possible security breaches

- **acceptability**:
  - mechanism must be simple to use
    - people use it without thinking about it
  - must rarely or never prevent permissible accesses
  - e.g. principle of least astonishment
- **fail-safe designs**:
  - default to lack of access
  - so if something goes wrong or is forgotten or isn't done, no security is lost
    - if false negatives, we can change the default on an individual basis

Policies

- policies describe how a secure system should behave:
  - describes what should happen, not how you achieve that
  - if you don't have a clear policy, you don't have a secure system
    - you don't know what you're trying to do
  - should address all relevant aspects of confidentially, integrity, and availability
  - *difficulties*:
    - hard to define policies properly
    - hard to go from policy to the mechanisms
    - hard to understand implications of policy
- informal policies:
  - e.g. "system executable should only be altered by system administrators"
  - e.g. "users should only be able to access their own files"
- formal policies:
  - typically expressed in a mathematical security policy language
    - e.g. Bell-La Padula model
  - tending towards precision
  - hard to express in formal ways and reason about them
  - mathematically, a policy partitions the system states into a set of authorized and unauthorized states:
    - secure system starts in an authorized state and cannot enter an unauthorized state
    - can reason about the system as an FSM moving between the states
- the **Bell-La Padula model** is the best-known formal computer security model:
  - two parts of clearances and classifications
    - real systems use classes of information with different classifications

    - corresponds to military classifications
    - combines mandatory and discretionary access control
    - each object has a **classification**:
        * describes how sensitive the object is
        * using same categories as clearances
        * informally, only people with the same or higher **clearance** should be able to access objects of a particular classification
            · a subject S can read object O iff. $l_O \leq l_S$
    - also concerned with object contents, not just objects themselves
        * what if someone with top secret clearance writes the information to a lower classification object?
        * additional Bell-La Padula *-property:
            · S can write O iff. $l_S \leq l_O$
            · prevents **write-down**
    - how do you use the system?
        * due to write-down, cannot communicate with someone lower privilege
        * needs mechanisms for reclassification, requiring explicit operation
    - can prove a system meeting these properties is secure in terms of confidentiality:
        * doesn't address integrity at all
        * **confidentiality policies** place no trust in object, just whether an object can be disclosed
- on the other hand, **integrity security** policies are designed to ensure that information is not improperly changed:
    - key issue for commercial systems
    - secrecy is nice, but not losing track of inventory is crucial
    - integrity policies operate based on how much some object can be trusted
        * policies then dictate what a subject can do with that object
- the **Biba** integrity policy addresses integrity:
    - subject set S, object set O, set of ordered integrity levels I:
        * subjects at high integrity levels are less likely to screw up date
        * data at a high integrity level is less likely to be screwed up
    - s can write to o iff. $i(o) \leq i(s)$
    - s_1 can execute s_2 iff. $i(s_2) \leq i(s_1)$
    - s can read o iff. $i(s) \leq i(o)$
- in hybrid models, sometimes the issue is keeping things carefully separated:
    - issues of *both* confidentiality and integrity
    - e.g. in the Chinese Wall model, all the resources, computers, people are separated and do not touch each other

Tools

- **physical security**:
  - lock up your computer
  - but what about networking, and mobility?
  - in any case, lack of physical security often makes other measures pointless
- **access controls**:
  - only let authorized parties access the system
  - difficult, particularly in a network environment
- **encryption**:
  - algorithms to hide the content of data or communications
  - only those knowing a secret can decrypt the protection
  - one of the most important tools
- **authentication**:
  - methods of ensuring that someone is who they say they are
  - vital for access control
  - often based on cryptography
- **encapsulation**:
  - methods of allowing outsiders limited access to resources
    - preferably making inaccessible things invisible
  - challenging in practice
- **intrusion detection**:
  - need to notice failures and take steps
  - reactive, not preventative
  - should be automatic to be really useful
- **common sense** is also a tool
  - social engineering attacks

Access Control

- how do we give access to only the right people?
  - at the right time and circumstances
  - similarly, how do we ensure a given resource can only be accessed when it should be
  - *goals*:
    - complete mediation
    - least privilege
    - useful in a network environment
    - scalability
    - acceptable cost and usability
- main types:
  1. access control lists
  2. capabilities
  3. access control matrix (both of the first two)
  4. role based access control
- definitions:
  - **subjects** are active entities that want to gain access to something e.g. users or programs
  - **objects** represent things that can be accessed e.g. files, devices, records
  - **access** is any form of interaction with an object
  - **mandatory** access control is dictated by the underlying system
    - can't be overrided by individual users
  - **discretionary** access control is under the command of the user:
    - system enforces what they choose
    - most users never change the defaults
    - not wise to rely on it to protect important information, for system designers
- **access control lists (ACL)** is the first mechanisms for implementing access control:
  - for each protected resource, maintain a single list
  - each list entry specifies a user who can access the resource, and allowable modes of access
  - when user requests access, check the ACL
    - can also have lower granularity than per-user, e.g. dictate per-process access
  - *issues*:
    - how do we know subject is who he says he is?
    - how do we protect the ACL itself from modification?
    - how do you determine what resources a user can access?

      - would have to check every single ACL, inefficient
  - *pros*:
    - easy to find who can access a resource
    - easy to change permissions
  - *cons*:
    - hard to find which resources a subject can access
    - changing access rights requires getting to the object e.g. across the Internet, distributed systems
  - used by most modern systems:
    - e.g. Linux, Windows, Android
    - to prevent additional lookups, only check ACL on *first* open:
      - maintain metadata in file descriptor, which begins to act as a capability
      - not the safest
- utilizing **capabilities** is the second main mechanism for access control:
  - each subject keeps a set of data items that specify his allowable accesses
    - i.e. a set of tickets
  - possession of the capability for an object implies that access is allowed
  - capabilities *must* be unforgeable:
    - in single machine, OS is in charge of capabilities
    - what about networked systems?
  - in most systems, some capabilities allow creation of other capabilities
    - allows process to pass a restricted set of capabilities to a subprocess
      - much more difficult with ACLs
  - *pros*:
    - easy to determine what a subject can access
    - potentially faster than ACLs
    - easy model for transfer of privileges
  - *cons*:
    - hard to determine who can access an object
    - requires extra mechanism for revocation
    - in a network, need cryptographic methods to prevent forgery
- how can we revoke a capability?
  - destroy the capability
    - how can we find it?
  - revoke on use?
    - requires checking on use
    - essentially turning capability into ACL
  - generation numbers i.e. generations of capabilities?
    - requires updating non-revoked capabilities
    - needs another list of subjects
- in distributed access control:
  - ACLs still work OK:

        - provided we have a global namespace for subjects
        - and no one can masquerade
    - capabilities are more problematic:
        - relies on unforgeability
        - provided by cryptographic methods
            - prevents forging, not copying
- **role based access control** is an enhancement to ACLs or capabilities:
    - each user has certain roles he can take while using the system
        - at any time, the user is performing a certain role
    - give the user access to only those things required to fulfill that role
    - available in some form in most modern OSes
    - only helps if changing roles isn't trivial
        - typically requires secure authentication
    - *limitations*:
        - number of roles per user
        - disjoint role privileges
        - system administration overheads
        - usability and management problems
- whatever form it is, access control must be instantiated in actual code:
    - needs to check if a given attempt to reference an object should be allowed AKA a **reference monitor**
    - good reference monitors are critical for system security
    - *properties*:
        - correctness
        - proper placement
        - efficiency
        - simplicity
        - flexibility

Cryptography

- the goal of **cryptography** is to keep enciphered information secret:
  - encryption is the process of hiding information in plain sight through transformation:
    - transform the secret data into something else
    - make the secret hard for others to read
      - while making it simple for authorized parties to read
  - counters disclosure
  - can be used to provide integrity of data and origin
    - counters modification and masquerading
  - can be used to provide non-repudiation
    - counters repudiation of origin
- the basic component of cryptography is a **cryptosystem**:
  - sender S, receiver R, attacker O:
    - **encryption** makes the message unreadable or unalterable by O
    - **decryption** makes the encrypted message readable by O
    - rules for transformation called the **cipher**
  - 5-tuple (E, D, M, K, C)
  - $E : M \times K \to C$ is the set of **encryption functions**
  - $D : C \times K \to M$ is the set of **decryption functions**
  - M is the set of **plaintexts**
  - K is the set of **keys**:
    - most algorithms use a key (usually secret) to perform encryption and decryption
    - if you change only the key, a given plaintext encrypts to a different ciphertext
  - C is the set of **ciphertexts**
- desirable characteristics of ciphers:
  - amount of secrecy required should match labor to achieve it
  - freedom from complexity
  - simplicity of implementation
    - probability of error is lower
  - errors should not propagate e.g. consider if bits get flipped
  - ciphertext size should be same as plaintext size
  - encryption should maximize **confusion** i.e. plaintext and ciphertext relationship should be complex
  - encryption should maximize **diffusion** i.e. plaintext information is distributed throughout ciphertext

Cryptanalysis

- **cryptanalysis** is the process of trying to break a cryptosystem:
  - finding the meaning of an encrypted message without knowing the key
  - successful when you don't get garbage when decrypting
    - almost all messages will be garbage if the key is wrong, only $\frac{1}{2^N}$ are sensible
- forms of cryptanalysis:
  1. analyze an encrypted message and deduce its contents
  2. analyze one or more encrypted messages to find a common key
  3. analyze a cryptosystem to find a fundamental flaw
- types of attacks:
  1. ciphertext only:
     - no plaintext knowledge or details of algorithm
     - must work with probability distributions, patterns of common characters, etc.
     - hardest type of attack
  2. known plaintext
     - have matching sample of ciphertext and plaintext
  3. chosen plaintext e.g. differential cryptanalysis
     - clever choices of plaintext may reveal many details
  4. algorithm and ciphertext:
     - can use exhaustive runs of algorithm against guesses at plaintext
     - or try and brute force
     - or, in a **timing attack**:
       - have ability to watch algorithm encrypting and decrypting
       - some algorithms perform different operations based on key values
       - watch timing or observe power use to try to deduce keys
       - successful against some smart card crypto
  - in many cases, intent is to guess the key
- most cryptosystems are breakable:
  - some just cost more to break than others
  - the job of the cryptosystem designer is to make the cost infeasible or incommensurate with the benefit extracted

Symmetric Cryptosystems

- **symmetric** cryptosystems have the same key for encipherment and decipherment

    - i.e. there is a $D_k \in D$ such that $D_k(E_k(m)) = m$ for message m
    - *pros*:
        - encryption and authentication performed in a single operation
        - well-known and trusted ones perform faster than asymmetric key systems
        - doesn't require any centralized authority
            - key servers can help
    - *cons*:
        - makes signature more difficult
        - non-repudiation
        - key distribution
        - scaling
- **transposition** or **permutation** ciphers diffuse the data in the plaintext:
    - the letters are not changed, only rearranged
        - e.g. columnar transpositions, double transpositions
    - detected by comparing character frequencies with a model of the language
    - can be attacked by anagramming i.e. rearranging the ciphertext:
    - cannot be attacked by examining individual letter frequencies
        - could check frequencies of **digrams** i.e. pairs of letters
- **substitution** ciphers change characters in the plaintext:
    - decrypt by reversing the substitutions
    - e.g. in a Caesar cipher, we translate each letter a fixed number of positions in the alphabet:
        - simple, but no good diffusion or confusion
        - could attack using letter frequencies to figure out the offset
            - the more ciphertext we have, the easier the attack
    - a **monoalphabetic** cipher maps every character into another character in one alphabet
        - preserves the statistics of the underlying message
    - a **polyalphabetic** cipher uses multiple alphabets, obscuring the statistics:
        - if patterns aren't hidden well, we don't gain much
        - can be attacked by examining repetitions:
            - **index of coincidence** predicts the number of alphabets used to perform the encryption
            - requires lots of ciphertext
- there is a "perfect" substitution cipher, the **one-time pad**:
    - one that is theoretically and practically unbreakable without the key
        - and the key cannot be guessed, if we chose the key correctly
    - we use non-repeating keys, where we use a new substitution alphabet for *every* character:
        - substitution alphabets chosen purely at *random*, and these consti-

tute the key
        - e.g. flip a coin many times to create a key stream
      - *any* key was equally likely
      - *any* plaintext could have produced this message
      - no longer has the property that only one key gives a non-garbage answer
  - usually done in practice with bits, not characters
  - *pros*:
    - if key is truly random, provable that it cannot be broken
  - *cons*:
    - need one bit of key per bit of message
    - key distribution is painful
    - key synchronization is vital
    - good random number generator is hard to find
  - typically not used, very difficult in practice:
    - pads distributed with some other cryptographic mechanism
    - pads generated non-randomly
    - pads reused
- in **quantum cryptography**, we use quantum mechanics to perform cryptography:
  - mostly for key exchange
  - relies on quantum entanglement or indeterminacy
  - can also use quantum computers to break cryptography:
    - famously can potentially break RSA
      - but has no use in cracking AES
    - currently non-feasible in reality
- modern ciphers tend to use both transposition and substitution
  - hide text patterns and also hide underlying text characters
- the **data encryption standard (DES)** is a classic symmetric cryptosystem:
  - bit-oriented
  - uses both transposition and substitution i.e. is a product cipher
  - input, output, and key are each 64 bits AKA one **block** long
  - consists of 16 rounds:
    - each round uses a separate key of 48 bits
      - generated from the key block by dropping parity bits, permuting, and extracting 48
    - if the order in which the round keys is used is reversed, input is deciphered
    - input of one round is output of the previous round
    - right input half and round key are ran through a function f that produces 32 bits of out
      - output is XORed into left half, and halves are swapped
  - f takes the right half of the input, expands it, and XORs it with the

round key
    - ⋆ the resulting 48 bits are split into eight sets of six bits each
    - ⋆ each set is put through a substitution table called the S-box that produces four bits of output
    - ⋆ results are concatenated into a single 32-bit quantity, which is then permuted
  - used from 1976 to 2001 (until the release of AES) as an official cryptography standard
  - *weaknesses*:
    - ⋆ key length of 56 bits is too short
    - ⋆ had weak and semiweak keys
    - ⋆ S-boxes were classified, suggesting that the classification hid ways to invert the cipher
    - ⋆ S-boxes exhibited non-randomness
- the **advanced encryption standard (AES)** succeeded the DES:
  - another bit-oriented product cipher
  - can use keys of 128, 192, or 256 bits
  - operates on 128 bits of input, producing 128 output bits
  - initial state array is transformed over the rounds into the output
  - consists of between 10 and 14 rounds:
    - ⋆ round key for each round generated by rotating and substituting the words in the original key
    - ⋆ round key added into state array, substitutions performed rows are shifted, and columns are mixed
  - basic operations such as XOR allows for high-performance implementations
  - *advantages over DES*:
    - ⋆ larger keys and better round key generation
    - ⋆ S-box values are nonlinear and algebraically complex
    - ⋆ inputs bits are rapidly diffused
    - ⋆ no weak or semiweak keys
  - *weaknesses*:
    - ⋆ attacks work on version of AES using fewer rounds
    - ⋆ attacks get keys quicker than brute force, but not practical time
    - ⋆ unusable flaws often suggest presence of usable ones

Asymmetric Cryptosystems

- a new type of cryptography proposed in 1976 had different keys for encoding and decoding:
  - keys created in pairs:

        - one key is public and its complementary key must remain secret
        - if you want to send an encrypted message, encrypt with his public key, and only he can decrypt
    - thus, this public key system should meet the following conditions:
        1. computationally easy to encode or decode given the key
        2. computationally infeasible to derive the private key from the public key
        3. computationally infeasible to determine the private key from a chosen plaintext attack
    - typically based on either NP-complete problems or hard mathematical problems e.g. finding factors
    - *vs. symmetric cryptosystems*:
        - easier authentication
            - no need to distribute a shared key
        - nicer scaling properties
            - each user just needs a key pair
- new challenge is publishing public keys in a trustworthy manner:
    - security depends on using the right public key
    - need high assurance a given key belongs to a particular person
    - needs some sort of **key distribution infrastructure**
- quick authentication with public keys:
    - to sign a message, simply encrypt it with your own private key
        - only you know the private key, so no one else could create that message
    - everyone knows the public key, so everyone can check the claim
    - solves some issues with shared key authentication
- ideally, we want to use both symmetric and asymmetric cyrptography:
    - public key used to "bootstrap" symmetric communication
    - e.g. RSA to authenticate and establish a session key
        - use AES with that session key for the rest of the transmission
- the notable **RSA** cryptosystem was introduced in 1978:
    - most popular public key algorithm, in wide use
        - has withstood much cryptanalysis
    - an exponentiation cipher based on factoring large numbers
    1. given two large prime numbers p and q, the **totient** $\phi(n)$ of $n = pq$ is the number of numbers less than n with no factors in common with n
        - alternatively, $\phi(n) = (p-1)(q-1)$
    2. choose an integer $e < n$ that is relatively prime to $\phi(n)$
    3. find a second integer such that $ed \; mod \; \phi(n) = 1$
    - public key is (e, n) and the private key is d
        - i.e. functions of a pair of 100-200 digit prime numbers
    - recovering plaintext without private key is supposedly equivalent to factoring product of the prime numbers

  - *vs. AES*:
    - AES is much more complex
      - but only arithmetic, logic, and table lookup
    - RSA uses exponentiation to large powers
      - much more computationally expensive
    - RSA key selection also more expensive
- elliptic curve cryptography:
  - another math problem
  - can give good security with much smaller keys
  - often used for small devices
- attacking public key systems:
  - nobody uses brute force attacks of checking 2^{2048} keys
  - instead, attack the mathematical relationship between public and private key

Checksums and Signatures

- in some cases, secrecy isn't necessary, but authentication is required:
  - data must be guaranteed to be unchanged
  - important for long-lived data
- desired signature properties:
  - unforgeable
  - verifiable
  - non-repudiable
  - cheap to compute and verify
  - non-reusable
  - no reliance on trusted authority
- signatures with shared key encryption require a trusted third party:
  - third party needed so receiver cannot forge the signature
  - instead, third party checks validity with secret keys shared with them
- with public keys:
  - signer can simply encrypt the document with his private key
  - receiver decrypts with signer's public key
  - no trusted third party needed, but receiver must be certain he has the right public key
  - to save on computation, or if we don't need encryption, just sign a checksum only
- a **checksum** or **message digest** is used to check against tampering:
  - e.g. parity bit is a simple checksum
  - should meet the following conditions:
    1. checksum is easy to compute

2. computationally infeasible to find the input from a checksum value
3. computationally infeasible to find another different input that gives the same checksum value:
    * by the pigeonhole principle, several messages *must* produce the same checksum
    * ideally, the hashes of all possible messages will be evenly distributed over the possible checksums

- **HMAC** is a generic term for an algorithm that uses a keyless hash function and a cryptographic key to produce a keyed hash function:
  - used in public key systems to validate data is unchanged in transit
  - without the key, anyone can change the data and recompute a digest

Ciphers In Practice

- some issues can arise when using cryptosystems in practice:
  - messages can be precomputed
    * in a small set of possible plaintexts, an attacker can use a "forward search" to precompute and compare ciphertexts
  - blocks can be misordered:
    * e.g. over a network, parts of a message can be deleted replayed or reorded
    * can checksum the entire message or have a sequence number in each block
  - statistical regularities
    * independence of parts of ciphertext can give information relating to the structure of the message, even if it is unintelligible
  - type flaw attacks
    * exploiting the structure or components of messages
- ciphers will often divide a message into a sequence of blocks:
  - can encipher each block with the same key, or use a nonrepeating stream of key elements AKA **stream ciphers**
  - **block ciphers** work on a given sized chunk of data at a time
- stream ciphers:
  - how can we generate a random, infinitely long key?
    * an algorithm is used to create the new key
      · e.g. RC4 cipher creates a changing, supposedly unpredictable, key stream
    * can use shift registers, or even obtain the key from the plaintext or ciphertext
  - *pros*:
    * speed of encryption and decryption

            - each symbol encrypted as soon as its available
        - low error propagation:
            - errors affect only the symbol where the error occurred
            - depends on cryptographic mode
    - *cons*:
        - low diffusion, each symbol separately encrypted
        - susceptible to insertions and modifications in the middle of a stream cipher
        - not good match for many common uses of cryptography
            - can mitigate some issues with proper cryptographic mode
- block ciphers:
    - most common Internet cryptography done with block ciphers
    - *pros*:
        - good diffusion
        - immunity to insertions
    - *cons*:
        - slower
        - worse error propagation
- we have a bunch of data to encrypt using the same cipher and key:
    - block ciphers have limited block size and stream ciphers just keep going
    - if we encrypt naively:
        - two blocks with identical plaintext encrypt to the same ciphertext!
        - each block of data was independently encrypted with the same key
        - we used the wrong **cryptographic mode** i.e. way of applying a particular cipher!
- cryptographic modes:
    - a combination of cipher, key, and feedback
    - in **electronic codebook (ECB)** mode, simply perform block cipher encryption block by block
    - in **cipher block chaining (CBC)** mode, a group of related encrypted blocks are tied together:
        - hides that two blocks are identical, foiling insertion attacks
        - the encryption version of the previous block is used to encrypt this block by XORing them together
            - adding feedback into the encryption
        - however, we have to fix the first block:
            - use **initialization vectors (IV)**
            - XOR a random string with the first block
            - ensures encryption results are always unique
    - cipher-feedback mode and output-feedback mode both convert block to stream cipher

Key Management

- it doesn't matter how strong the algorithm is if the keys are insecure:
  - proper use of keys is crucial
  - ciphers don't get cracked often, but keys get leaked all the time
- if algorithm is otherwise completely secure, strength depends on key length:
  - singe the only attack is a brute force attempt
  - however, with longer keys, encryption costs more and is slower
  - some algorithms have defined key lengths only
- **perfect forward secrecy** means that the compromise of any one session key will not compromise any other
  - keys get divulged, so minimize the resulting damage
- key lifetime consideration:
  - long-lived keys are more likely to be compromised
  - more data is exposed
  - easier cryptanalysis
  - more resources attackers can devote to breaking it
  - even old keys can be found in multiple places after being destroyed:
    - e.g. caches, virtual memory, freed file blocks, stack frames, etc.
    - need to zero out the key value
- key lifetime examples:
  - symmetric session keys:
    - e.g. keys for specific communications sessions should be changed often
    - avoid storing them permanently
  - long term symmetric keys:
    - e.g. disk encryption
    - safe storage is critical
  - private asymmetric keys:
    - long-term storage as well
    - safe storage is critical
- storing a user's keys:
  - permanently on machine
    - machine can be cracked
  - difficult to remember keys
    - hash keys from passwords or passphrases
  - smart cards
  - key servers
- key secrecy breaches:
  - private keys are often shared:
    - for convenience

    - to share expensive certificates
    - don't know any better
  - entire security of public key system depends on the secrecy of the private key

Key Exchange

- an **interchange key** is associated with a principal i.e. user
  - while a **session key** is associated with the communication session itself
- the first hurdle to overcome is transmitting the session key:
  - session key must be encrypted when it is exchanged
    - in order to exchange, may need a trusted third party

Symmetric Key Exchange

- simple symmetric key exchange:
  1. A asks third party C to start a session with B
  2. C sends to A the session key encrypted with A's key, followed by the session key encrypted with B's key
  3. A sends to B the session key encrypted with B's key
  - note that A's key and B's key are keys shared with them and the trusted third party
    - symmetric, not public keys
  - vulnerable to a man-in-the-middle attack, before the following minor changes:
    - encrypt request with A's key
    - include identity of other participant in response from C
  - however, still compromised using repeating messages
- types of security protocols:
  - **arbitrated protocols** involve a trusted third party
  - **adjudicated protocols** involve a trusted third party, after the fact
  - **self-enforcing protocols** do not involve a third party
- Needham-Shroeder protocol:
  - another symmetric key exchange and authentication protocol
  - uses **nonces** or randomly generated numbers to defend against replay attacks
  1. A sends to third party C: A's name, B's name, and a nonce r_1
  2. C sends to A: A's name, B's name, r_1, the session key, A's name plus the session key encrypted with B's key, all encrypted with A's key:
     - i.e. $\{A||B||r_1||k_{session}||\{A||k_{session}\}_{k_B}\}_{k_A}$
     - A is now sure of who they are talking to, and the nonce assures against replay attacks

3. A sends to B: A's name and the session key, encrypted with B's key
    - B now knows who they are talking to
4. B sends to A: another nonce r_2 encrypted with the session key
5. A sends to B: $r_2 - 1$ encrypted with the session key

- cannot be easily compromised with repeated messages:
    * still possible for old session keys to be cracked by attackers, and B's challenge to A can be forged
    * in this case, can add timestamps to further counter repeats, which requires synchronized clocks
        - e.g. Kerberos protocol with tickets

- global clocks and timestamps:
    - often hard to obtain a globally synchronized set of clocks
        * attacker can attack clocks as well
    - in a suppress-replay attack, attacker can intercept and replay if the sender's clock is behind
    - clock solutions:
        1. rely on clocks that are fairly synchronized and hard to tamper with e.g. GPS signals
        2. make all comparisons against the same clock
- Otway-Rees protocol:
    - avoids timestamps
    - uses an integer n to associate all messages with a particular exchange
    1. A sends to B: n, A's name, B's name, and r_1 plus n plus A's name plus B's name encrypted with A's key
        - i.e. $\{n||A||B||\{r_1||n||A||B\}_{k_A}\}$
    2. B sends to third party C: n, A's name, B's name, r_1 plus n plus A's name plus B's name encrypted with A's key, and r_2 plus n plus A's name plus B's name encrypted with B's key
        - i.e. $\{n||A||B||\{r_1||n||A||B\}_{k_A}||\{r_2||n||A||B\}_{k_B}\}$
    3. C sends to B: n, r_2 and the session key encrypted with A's key, and r_2 and the session key encrypted with B's key
    4. B sends to A: n, and r_1 and the session key encrypted with A's key
    - goal is to prevent replay attacks
- Bellare-Roagaway protocol:
    - considers authentication and symmetric key exchange different problems
        * protocol only provides the key exchange, trusted server sends to both parties
    1. A sends to B: A's name, B's name, r_1
    2. B sends to third party C: A's name, B's name, r_1, r_2
    3. C sends to B: session key encrypted with B's key, keyed hash of A's name, B's name, r_1, and the session key encrypted with B's key
        - keyed hash utilizes the user's interchange key

4. C sends to A: same as (3), but with A's keys

Public Key Exchange

- conceptually, public keys makes exchange keys very easy:
    - A sends to B: the session key encrypted with B's public key
        - attacker can easily forge message
    - (revised) A sends to B: A's name and the session key encrypted with A's private key, all encrypted by B's public key
        - after receiving the message, B can use A's public key to obtain the session key
- **man in the middle** attack:
    - occurs when A has to first obtain B's public key
    1. A asks C for B's public key
        - attacker intercepts and asks C themselves for B's public key
    2. C responds to attacker with B's public key
    3. attacker sends to A their own public key
    4. A sends to B the session key encrypted with the attackers public key
        - attacker intercepts again and sends to B themselves the session key encrypted with B's public key
    - no binding of identity to a public key
        - to resolve, need to look at management of cryptographic keys
- Diffie-Hellman key exchange:
    - securely exchange a key:
        - without previously sharing any secrets
        - no public key available or symmetric key
        - using an insecure channel
    - first two parties need to agree on a large prime n and a number g
        - n, g don't need to be secrets, typically predefined in their software
    1. A chooses a large random integer x and sends B $X = g^x \mod n$
    2. B chooses a large random integer y and sends A $Y = g^y \mod n$
    3. A computes $k = Y^x \mod n$
    4. B computes $k' = X^y \mod n$
        - $k = k' = g^{xy} \mod n$
    - but nobody else can compute k, k'!
        - others know n, g, X, Y, but not x, y
        - knowing X, Y gets you nothing, unless you compute the discrete logarithm to obtain x or y
            - believed to be hard
        - typically, x, y are just the users private keys
    - D-H guarantees that two parties share a secret:
        - but it doesn't guarantee who those two parties are
        - how does A know whether the Y she heard was sent by B?

    - D-H does not authenticate the parties
  - authentication in any key distribution is a core Internet problem, TC/IP does no authentication!
    - however, D-H is used all the time

Key Generation

- a sequence of **random** numbers is a sequence such that an observer cannot predict any x_k even if all the previous numbers are known:
  - requires physical source of randomness or noise, e.g. background radiation, electromagnetic phenomena, biometrics, disk drive delay
    - done in the background AKA **gathering entropy**
  - on the other hand, a sequence of **pseudorandom** numbers is a sequence generated by an algorithm intended to simulate random numbers
    - need statistical properties and non-reproducibility
- pseudorandom generators:
  - how good is that generator?
    - don't use `rand`
  - linear congruential generator $x_k = (ax_{k-1} + b) \cos n$ has been broken, as well as polynomial congruential generator
  - the outputs of a **strong mixing function** depend on some nonlinear function of all input bits e.g. SHA:
    - best generator algorithms
    - one approach is to continue to hash old ones to produce new keys
      - does not have perfect forward secrecy, and depends on strength of the has algorithm

Key Infrastructures

- how can we guarantee the true owner that a public key belongs to?
  - need a trusted third party or authority to sign some sort of **certificate** binding an identity to a cryptographic key
    - or some kind of central server
  - but now we need to distribute the third party's public key… which needs to be verified by an additional certificate?
    - there is no universally trusted single authority
    - does everyone need the public keys for all certificate authorities?
- **key servers** are machines whose job it is to distribute keys to other machines:
  - clients can authenticate themselves to the server
  - server can authenticate itself to the clients

    - bootstrapping and transitive trust issue
    - not the popular solution
- certificate is essentially a copy of a public key together with an identity signed by a trusted authority:
    - usually has an expiration date
    - presentation of the certificate alone serves as authentication of your public key
    - problems during certification process:
        - what measures did CA take before issuing?
        - how long is certificate valid for?
        - is CA's own certification still valid?
        - who is trustworthy enough to be at the top of the hierarchy?
        - what do we do when keys are compromised?
        - user may have different standards than the CA
    - revocation is a general problem for keys, certificates, etc.
        - how does the system revoke something related to trust, in a network environment?
        - related to revocation problem for capabilities
        - one approach is OCSP, an online system that indicates if certificates have been revoked
            - used in different ways by different OSes and browsers
    - typically, most attackers do not break in using certificate validity:
        - not the weakest link
        - but now being exploited, mostly by sophisticated adversaries
- **Merkle's tree authentication**:
    - keeps public keys and their associated identities as data in a file
        - uses checksums to detect data integrity breaches
    - keys and identities in the file are organized into a tree structure
        - hash of the entire file is the **root**
    - during validation, can traverse an **authentication path** on the tree to verify the checksums
        - if the root value does not matched, an identity / key pair has been compromised
    - *pros*:
        - creates certificates without using public key signatures
        - suggests natural hierarchies
    - *cons*:
        - requires entire file
        - any changes requires wide redistribution
- a **certificate authority (CA)** is an entity that issues certificates:
    - there is no one CA for the entire Internet
    - CAs could be organized into a single hierarchy:
        - single CA at the top supplies certificates for the next layer, etc.

    * in practice however, we rely on large numbers of independent certifying authorities, each of which may have its own internal hierarchy
  - for new certificates by an unknown CA, the certificates also contain that authority's certificate
  - in reality, most OSes or browsers come with a set of "pre-trusted" certificate authorities (sometimes around hundred certificates):
    * system automatically trusts certificates they sign
      · usually no hierarchy
    * if not signed by one of those, present it to the user

Authentication

- generally, **authentication** is the binding of an identity to a subject:
  - e.g. process, machine, human user
  - physically identify through credentials, recommendation, knowledge, location, etc.
    - these all have cyber analogs
    - but, authentication is done over a network, even if the party is human
      - everything is converted to digital signal
    - in addition, identity might not be rechecked
  - more general than authentication in cryptography
    - access control only works if you have good authentication
  - authorization is determining what someone can do
  - there is a certain set of specific information with which entities prove their identities
    - can be passwords, biometrics, etc.
  - importantly, there is another set of information which the system stores that is used to validate the authentication information from the user
    - i.e. the complementary information
- the simplest authentication mechanism is a **password**, often a sequence of characters:
  - i.e. authenticated by what you know
  - complement can simply be the password in plaintext:
    - instead, should hash the password into a complement using a *one-way* function
    - retrieving the password file does not allow you to log in to the system
  - password selection:
    - random selection of passwords
      - strength of the pseudorandom generator
    - computer-generated pronounceable passwords
      - less strong, but easier to remember
    - user selected passwords
      - should avoid names, dictionary words, keyboard patterns, short passwords, etc.
    - graphical passwords
  - typically, passwords are **salted** by adding random data before the password is hashed:
    - random number need not be secret
    - just different for different users

        * makes dictionary attacks much more difficult
        * similar to nonces and initialization vectors
    - passwords have an aging issue:
        * can be cracked over time
        * should change passwords periodically
        * one-time passwords invalidate immediately
        * many systems ask for password once, trading security for convenience
- proper use of passwords:
    - sufficiently long
    - contains non-alphabetic characters
    - ungeuessable
    - changed often
    - never written down
    - never shared
- attacks:
    - in an **offline dictionary attack**, the attacker knows the complementation functions and stored complementary information:
        * e.g. has the encrypted password file
        * repeatedly guesses different passwords and applies the functions
        * real dictionary attacks use probability of words being used as passwords
    - in an **online dictionary attack**, the attacker guesses directly into the system, without other previous knowledge:
        * with **backoff**, systems increase the time between interactions with more tries
        * with **disconnection**, the connection is broken after a number of failures
        * with **disabling**, the account is disabled
        * with **jailing**, the user gets false access to a limited part of the system
            · can also **honeypot** the system with false data to trap attackers
    - modern machines are very fast, so even with salting, huge dictionaries can be checked against encrypted passwords:
        * GPUs excel at password cracking
        * even salted, hashed passwords are not safe
- password management:
    - limit login attempts:
        * prevents dictionary attacks "over the wire"
        * lock account, slow down, etc.
    - encrypt passwords:
        * store unencrypted passwords as briefly as possible e.g. no temp files
        * same with password attempts into a log file, etc.
        * passwords should be sent over HTTPS

  - protect the password file
    * make dictionary attacks more difficult
  - for forgotten passwords, should generate new passwords
    * site should never be able to send back forgotten passwords, implies that there is a way to decrypt encrypted passwords
  - transporting new passwords:
    * generally sent encrypted via email or text message
    * both are compromisable
    * some banks reuire surface mail
  - user passwords:
    * using same vs. different passwords for sites
    * security vs. usability
    * password vaults, write down passwords
- another authentication mechanism is **challenge / response**:
  - authenticate based on questions you can answer correctly i.e. what you know
    * e.g. security questions, or smart card
  - can ask for different information every time
    * or challenge the hardware to perform something e.g. encrypt it with a unique key
  - security depends on encryption of the challenge
  - question is too hard to answer without special hardware, or too easy for intruders to spoof the answer
  - smart card details:
    * cryptography should be performed only on smart card
    * user should enter password into card
  - *cons*:
    * if lost or stolen, can't authenticate, and maybe someone else can
    * susceptible to sniffing attacks
    * requires special hardware
- **biometrics** is another mechanism based on who you are:
  - fingerprints, voice patterns, retinal patterns, etc.
  - to authenticate, allow system to measure physical characteristics
    * biometrics converted to digital
  - interplay vs. false positive and false negatives:
    * more sensitivity means lower false positive rate, but also higher false negative rate
    * the **crossover error rate (CER)** is the point where the rates meet
    * for usability, false negatives are very undesirable
  - *good use cases*:
    * use them for authentication with clean readings
    * when biometric readers themselves are secure
    * when attacks are rare or difficult

        - together with other authentication
    - *poor use cases*:
        - working off low-quality / noisy readings
        - finding "needles in haystacks"
        - when biometric reader is easy to bypass or spoof
            - anything across a network is suspect
    - *cons*:
        - requires very special hardware
        - not as foolproof as you might think
        - generally not helpful for authentication programs or roles
        - many physical characteristics vary too much for practical use
- authentication by where you are
    - requires sufficient proof of physical location and ability to tie a device at that location to its messages
- **multifactor authentication**:
    - something you know + something you have
        - at least one factor needs to be non-replayable
    - e.g. PIN + ATM card, password + phone
    - either can go wrong for a false negative
    - are the factors really orthogonal?
    - are both factors non-trivial?
    - is one factor likely to suffer a catastrophic break?

Operating Systems

- what does the OS protect?
  - authentication for operating systems
  - memory protection e.g. buffer overflows
  - IPC protection e.g. covert channels
  - stored data protection e.g. full disk encryption
- the OS provides the lowest layer of software visible to users:
  - close to hardware, often with complete hardware access
  - OS flaws compromise all security at higher levels
  - OS controls memory, scheduling, devices, other resources
  - systems may be single user, multiple user, embedded with no human user
    - all still require OS security
  - almost all other security systems must assume a secure OS at the bottom
- security *depends* on running the right OS and version, not altered by an attacker:
  - i.e. **trusted computing**
  - need trusted hardware that makes sure the boot program behaves and runs the right OS:
    - AKA **security enclaves**
    - hardware implementation is challenge, often has known flaws
  - the **trusted platform module (TPM)** is special hardware designed to improve OS security:
    - proves OS was booted with a particular bootstrap loader using tamperproof hardware and cryptographic techniques
    - provides secure key storage and crytpo support
    - checks signatures of the OS etc.
    - bootloader and users can request TPM to verify applications or OS
    - not *guaranteed* security, but creates a chain of transitive trust
  - TPM hardware is widely installed, but not widely used:
    - e.g. Microsoft Bitlocker, secure Linux boot loader
    - Microsoft's SecureBoot is another build software alternative that only boots systems with pre-arranged digital signatures
- authentication and authorization in OS:
  - OS must *authenticate* all user requests
  - human users log in locally and remotely, and processes run on their behalf
  - once authenticated, requests must be *authorized*
  - remote user authentication timeline:
    1. user authenticates via password, public key crypto, sometimes a

particular process, etc.
2. successful login creates a primal process under ID of logged in user
3. OS ties a process control block to the process with owner ID
4. process can fork off more processes
    * invoking system calls checks owner IDs through **reference monitors**

  * special system calls can change a process's ID
- how often should OS perform authorization?
  * passing operations through reference monitors add overhead
  * balance between overhead and necessary authorization
  * e.g. only on first check, incrementally, periodically, etc.

- protecting memory:
  - memory contains executable code, copies of permanently stored data, and temporary process data
  - virtual memory provides a logical separation of processes:
    * for error containment more so than security
    * main memory divided into page frames, every processes has an address space divided into logical pages
    * each process is given a table, and all addressing goes through the page table at the hardware level
    * a process shouldn't be able to name other processes' pages
  - security issues of page frame reuse:
    * OS switches ownership of page frames as necessary
    * when a process acquires a new page frame, can the process read the old page frame data?
  - need to clean page:
    * e.g. zero on deallocation, zero on reallocation, zero on use, clean pages in background
    * Linux zeroes on reallocation, Windows cleans in background
  - **buffer overflow** is one of the common causes for compromises of operating systems:
    * process messing with its own memory, running different code by changing the function return address:
      · i.e. choosing what gets written into the instruction pointer
      · programs often run on behalf of others, so this is dangerous
    * can be interpreted as a flaw in OS input processing, programming languages, or even programmer training
    * **stack overflow** is a kind of buffer overflow intended to alter the contents of the stack
    * **heap overflow** does not offer the direct ability to jump to arbitrary code (heap is mostly non-executing), but potentially quite dangerous
  - fixing buffer overflows:

-
    -
        - write better code
        - use programming languages that prevent them
        - add OS controls that prevent overwriting the stack
        - put things in different places on the stack
        - don't allow execution from places in memory where overflows occur
- protecting interprocess communications:
    - OS provides various kinds of IPC e.g. messages, semaphores, shared memory, sockets
    - *possible exploits*:
        - convince system process is another process
            - an authentication problem
        - can break into another process's memory
            - handled by page tables
        - forge a message from someone else
            - OS tags IPC with identities
        - eavesdrop on someone else who gets the secret
            - related to page reuse and internal OS buffers
    - mostly secure, but hard for certain scenarios:
        - bug in the OS
        - not a single machine
            - depends on strong authentication and authorization
        - OS has to prevent cooperating processes from sharing information
        - process wants to communicate with another process, but OS has been instructed to prevent that e.g. mandatory access control
    - in **covert channels**, we use something not ordinarily regarded as a communications mechanism to actively attempt to deceive the OS:
        - e.g. disk activity, page swapping, time slice behavior, use of a peripheral device
        - only need to send 0's and 1's
        - very difficult to detect
- stored data protection:
    - files are a typically shared resource
    - data stored on disk is subject to many risks:
        - if OS protections are bypassed, how can we protect data?
        - store data in encrypted form
    - *issues*:
        - when does cryptography occur?
            - which files, explicitly or implicitly, how long decrypted, where does it existed in decrypted form
        - where does the key come from?
            - human user, file system, smart card, disk hardware, where and how long do we store

        * what is the granularity of cryptography?
            - disk, file system, block
    - *practicality*:
        * for improper users, why not just use access control
        * no point in hiding from OS
        * for data transfers, encrypt while in transit
        * someone who physically accesses the device not using the OS
            - only relevant attack that encryption protects against
    - in full disk encryption:
        * all data on the disk is encrypted
            - data is encrypted and decrypted as it enters and leaves disk
        * prevents improper access to stolen disks
        * could be done in hardware or software

Network Security

- degree of locality:
  - some networks are very local e.g. Ethernet
  - benefits:
    - physical locality
    - small number of users and machines
    - common goals and interests
  - other networks e.g. Internet are very non-local
    - many users and sites share bandwidth
- network media e.g. wires, cables, telephone lines can be physically protected
  - satellite links and radio links have more limited *physical* protection possibilities
- implication of protocol type:
  - protocol defines a set of rules that will always be followed
  - specific attacks exist against specific protocols
- threats to networks include wiretapping, impersonation, confidentiality and integrity attacks, DoS attacks:
  - **passive wiretapping** is listening in illicitly on conversations
  - **active wiretapping** is injecting traffic illicitly
  - **packet sniffers** can listen to all traffic on a broadcast medium
  - wiretapping on wireless is often just putting up an antenna
  - message can be read or even altered at intermediary gateways and routers
    - typically requires access to part of the path the message takes
  - in denial of service, legitimate users are prevented from doing their work by flooding the network or corrupting routing tables or flooding routers or destroying key packets
    - all-inclusive nature of the Internet makes basic access trivial, universality of IP makes this easy
- SYN flood attack:
  - attacker uses initial request and response to start enough TCP sessions to fill a table that is used to keep track of connections at the server
    - sends a bunch of SYN requests, without acknowledging the SYN/ACK
  - prevents new real TCP sessions
    - server cannot delete half-open connections in case we have a slow, real client
  - can defend with SYN cookies and firewalls along with large tables
  - SYN cookie approach:
    - when table is almost full, server sends back a SYN/ACK, *without*

creating a new table entry, that contains a cookie
    - cookie value is a secret function of various information e.g. client/server address and port, timer
    - store the cookie as the sequence number itself!
        - no need to change the protocol to support cookies
    - server doesn't need to save cookie values
    - slows down attacker greatly since he would need to create full connections to actually take up space in the table

- **distributed denial of service (DDoS)** attacks:
  - send a large volume of packets from a large number of distributed machines
    - no need to target a particular exploit like TCP tables
  - distribution harnesses multiple machines and makes defenses harder
  - if more packets sent that can be handled by target e.g. link or server, service is denied
    - could be pure flooding, or overwhelming of CPU or memory resources, direct or reflected
  - *complications*:
    - high availability of compromised machines
    - Internet is designed to deliver traffic
    - IP spoofing allows easy hiding
    - distributed nature makes legal approaches hard
    - attackers can mimic normal packets
  - *defense approaches*:
    - overprovisioning
    - dynamic increases in provisioning
    - filtering
    - traffic redirection e.g. content delivery networks
    - reducing volume of attack
    - none of these are totally effective

Traffic Control Mechanisms

- in **source address filtering**, we filter out some packets because of their source address value:
  - AKA ingress or egress filtering, address assurance
  - usually because we believe their address to be spoofed
  - router knows what network it sits in front of:
    - filter outgoing packets with source addresses not in its range
    - prevents users from spoofing other nodes' addresses, but not from spoofing each others
  - can also be done in the other direction, as packets leave the Internet and enter a border router

        * only prevents spoofed IPs that are in the local network (these packets should have never left the local network, so we can safely drop them)
- other forms of filtering e.g. worm signatures, unknown protocol identifiers, unallocated IP addresses, local use addresses only
    - can also redirect packets to a special filtering site on the edge of the network:
        * expressively designed to deal with DDoS attacks with aggressive filtering criteria
        * incurs serious delay penalties
- realistic limits on filtering:
    - little filtering possible in Internet core:
        * packets handled too fast
        * backbone providers typically don't want to filter
    - filtering near edges is also limited in terms of possibility, affordability, what router owners will do
- many routers can place **limits** on the traffic they send to a destination:
    - limits defined flexibility
    - often not good enough to differentiate good and bad traffic
- to better hide traffic characteristics, we can use **padding** to add extra traffic
    - fake traffic must look like real traffic
- similarly, use ability to control message routing to conceal the traffic in the network
    - i.e. using **onion routing** to hide who is sending traffic to whom for anonymization purposes
- a **firewall** is a machine to protect a network from malicious external attacks:
    - running special software to regulate network traffic and control entry and exit points
        * examines each incoming packet and decide to let the packet through or not
    - a form of security called **perimeter of defense**
    - breaching the perimeter compromises all security
    - part of the solution, but not the entire solution
        * i.e. defense in depth by combining different defenses
    - *types*:
        * filtering gateways AKA screening routers
        * application level gateways AKA proxy gateways
        * reverse firewalls
- filtering gateways filter based on packet header information:
    - IP addresses can always be spoofed
        * firewall should not always trust packet headers
    - can filter based on ports to drop packets sent to little-used ports
    - *pros*:

    * stateless
    * fast, cheap, flexible, transparent
- *cons*:
    * limited capabilities
    * dependent on header authentication
    * generally poor logging
    * may rely on router security