University of Houston

Foundations of Security

COSC 6347

Final Exam Review

Author
K.M. Hourani

Based on Notes By
Dr. Aron Laszka

December 8, 2020

1 Advanced crypto

1.1 Advanced Cryptographic Primitives

- secure multiparty computation
- homomorphic encryption

1.2 Commitment Schemes

Commitment Problem

- Bob "calls" the coin flip (i.e., heads or tails)
- Alice flips the coin
- Bob wins if her call is correct, Alice wins otherwise

Can we prevent Alice from cheating even if the players are not in the same physical location?

Commitment Scheme

- Two phases
  1. commit: A chooses a value V, A sends a **commitment** of V to B
  2. reveal: A reveals the value of V
- Example: coin flipping
  1. commit: A flips a coin, A sends a commitment (i.e., coin is heads or tails) to B
  - B calls the coin flip (i.e., heads or tails)
  2. reveal: A reveals the value of the coin flip
- Requirements for commitment scheme
  - B cannot learn the value of V from the commitment
  - A can reveal only the originally chosen value for a commitment

Naive Attempt Using Hash Function

- H: cryptographic hash function
- If the set of possible values of V are small (e.g. "heads" or "tails"), B can learn V by simply trying all possible values

Secure Commitment Using Hash Function

- Collision-free hash function $\rightarrow$ A cannot cheat by finding V_1 and V_2 such that $H(r_1 \mid r_2 \mid V_1) = H(r_1 \mid r_2 \mid V_2)$
  - r_1 prevents pre-computation of colliding V_1 and V_2

1.3 Secret Sharing

- Problem: distribute a secret among N participants such that
  - any group of at least T participants can reconstruct the secret
  - no group of fewer than T participants can reconstruct any part of it
- Types
  - **unconditionally secure**: information-theoretically secure (unbounded attacker)
  - **conditionally secure**: typically more efficient

Special Case: $T = N$

- Unconditionally secure scheme:
  1. let the secret be a binary number S
  2. pick $N - 1$ random numbers R_1, R_2, …, R_{N-1} of the same length
  3. give each participant i, $i < N$, the number R_i

4. give the last participant the result of $S \oplus R_1 \oplus R_2 \oplus \ldots \oplus R_{N-1}$

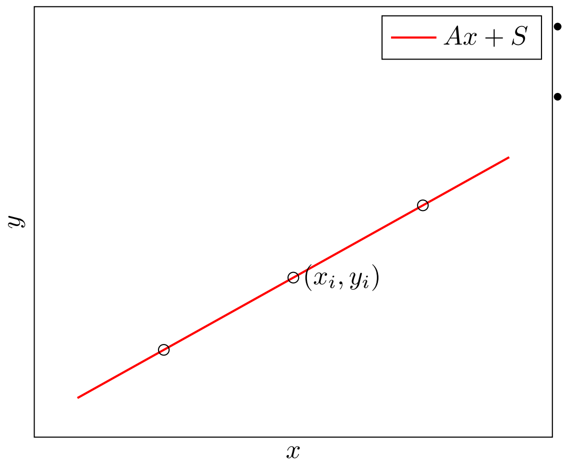
- N participants can reconstruct the secret by XORing their numbers:
  -

$$\begin{aligned} & R_1 \oplus R_2 \oplus \ldots \oplus R_{N-1} \oplus (S \oplus R_1 \oplus R_2 \oplus \ldots \oplus R_{N-1}) \\ & = (R_1 \oplus R_1) \oplus (R_2 \oplus R_2) \oplus \ldots \oplus (R_{N-1} \oplus R_{N-1}) \oplus S \\ & = S \end{aligned}$$

- $N-1$ participants can compute only $S \oplus R_i$, where i is the missing participant (or $R_1 \oplus R_2 \oplus \ldots \oplus R_{N-1}$ if the last participant is missing)

1.4 Shamir's Secret Sharing

- Proposed by Shamir in 1979
- Unconditionally secure
- Special case $T = 2$
  1. let the secret be a number S
  2. pick random number A
  3. let each participant's share be a random point on the line $Ax + S$



- $T = 2$ participants can reconstruct the secret since any two points define a line
- Single participant cannot learn the slope

General Case

- Arbitrary T:
  1. let the secret be a number S
  2. pick random numbers A_1, A_2, …, A_{T-1}
  3. let each participant's share be a random point from the curve

$$y = S + A_1 x + A_2 x^2 + \cdots + A_{T-1} x^{T-1}$$

- At least T points are necessary to define a polynomial of degree $T - 1$
- Example $T = 3$
  - secret is a parabola (i.e., $A_2 x^2 + A1x + S$)
  - there an infinite number of parabolas fitting two points
  - but three point define one uniquely

1.5 Secure Multiparty Computation

- Problem: N participants with private data d_1, d_2, …, d_N
  - participants would like to compute the value $F(d_1, d_2, \ldots, d_N)$ of a public function F over their private data

    - no participant i would like to reveal any information about its data d_i
- Requirements
    - **privacy**: no information is revealed about any private data (other than what is revealed by the public output)
    - **correctness**: public function is correctly computed
- Adversaries may be semi-honest (passive) or malicious (active)

2 WiFi security

2.1 Security Challenge

- Problem: no inherent physical protection
- **joining** a network does not require physical access
- radio transmissions are broadcast → anyone in range can **eavesdrop**
- **injecting** new messages or **replaying** old messages is possible
- **jamming** attacks against availability
- jamming and injecting messages can be combined into **tampering** attacks

2.2 Simple ”Solutions” for Access Control

Hidden SSID

- Association request must contain the SSID of the network
    - by default, the AP broadcasts it periodically in the beacon
- AP may be configured to **stop announcing the SSID** → SSID may be be used as a “password”
- However,
    - SSID must be hard to guess
    - every authorized user must know the SSID
    - **SSID can be easily eavesdropped** whenever an authorized station connects to the network → does not provide any security
- Tools are available for eavesdropping (e.g. Aircrack-ng)

MAC Address Based Filtering

- AP may be configured to **allow only devices with certain MAC addresses** to connect
    - MAC addresses of all authorized devices must be registered in advance
- However,
    - **MAC address is sent in plaintext** in every packet
    - many WLAN devices allow their MAC addresses to be changed → attacker can easily impersonate an authorized user

2.3 802.11 Security Standards

WEP

- security is based on a 40 or 104-bit secret key
    - WiFi ”password” shared by all users
- confidentiality: RC4 stream cipher
    - key is extended by a 24-bit IV, which is changed for each message → used as nonce to prevent key reuse problems
- integrity: encrypted CRC32 (Cyclic Redundancy Check) checksum
- access control: challenge-response between AP and station

WEP Design Flaws

- Authentication
  - **one-way authentication** (only for station) → AP can be impersonated
- Integrity protection
  - based on **error-detection code** (CRC32) instead of cryptographic hash → forging authentication tags is trivial
  - **no message replay protection**
- Key usage
  - **no session key**: long-term key used for all purposes (authentication, encryption, integrity protection)
  - **short nonce** (i.e., 24-bit IV) → danger of key reuse for stream cipher
    - busy network with 1000 packets per second reuses in less than 5 hours
  - vulnerable to Fluhrer-Mantin-Shamir Attack
    - In practice, WEP keys can be broken in a matter of minutes (or less) → WEP is **not secure**

2.4 WiFi Protected Access (WPA)

WPA

- Standard: 802.11i TKIP (Temporal Key Integrity Protocol
- Design goals: **fix the flaws of WEP** and be **compatible with legacy hardware**
- Overview
  - key usage: session key is established during a secure two-way authentication
  - confidentiality: RC4 encryption, but with **48-bit IV**, which is **mixed thoroughly** with the session key and source MAC address
    - prevents key reuse and the Fluhrer-Mantin-Shamir attack
  - integrity: 64-bit message integrity codes computed using Michael, which is **computationally very efficient** but provides only 20 bits of effective security
    - after wrong code, station is banned for a minute and needs to re-authenticate
  - Deprecated in later revisions of the standard
-

WPA-2

- Standard: **IEEE 802.11i**
- WPA 2 Devices can be certified by the Wi-Fi Alliance

Phases

1. Discovery
   - agree on what authentication method and ciphers to use
2. Authentication
   - may use an authentication server
   - create a master session key
3. Key management
   - derive keys for various purposes
4. Protected data transfer
5. Connection termination

Discovery Phase

- Goal: station and AP may support different sets of authentication methods and ciphers → they need to agree on which ones they will use
- **Authentication and key-management suite**: how to perform mutual authentication and derive fresh keys
  - IEEE 802.1X, pre-shared key (PSK), or vendor-specific
- **Cipher suite**: what ciphers to use for confidentiality and integrity
  - WEP, TKIP, CCMP, or vendor-specific
- Protocol
  1. AP can periodically **broadcast** its security capabilities using a **Beacon** (or station can ask for it using a Probe Request message)
  2. Station **specifies** an authentication and cipher suite in an **Association Request**
  3. if the AP **accepts** the specified suites, it sends an **Association Response**

Authentication Phase

- Goals:
  - mutual authentication:
    1) only authorized stations can use the network
    2) station is assured that it communicates with a legitimate network
  - generate **pairwise master key** (PMK)
- Approaches
  - Pre-shared key (PSK)
    - password is deployed on each station and the AP manually
    - PMK = PSK = generated from the password using a hash function
    - ideal for home and small office networks
  - IEEE 802.1X

Key-Management Phase

- Goals:
  - derive **pairwise transient keys** from the PMK
  - distribute **group keys**
- Pairwise transient key (PTK)
  - protecting data between station and AP
  - generated from PMK and the AP's and station's MAC addresses and nonces
- Group temporal key (GTK)
  - protecting multicast communication
  - group master key (GMK): generated randomly by the AP
  - distributed using the PTK

Protected Data Transfer Phase

- Standard defines two schemes: TKIP and CCMP
- TKIP: see WPA
- CCMP (Counter mode CBC-MAC Protocol)
  - based on the CCM (Counter with CBC-MAC) authenticated encryption mode
  - integrity: CBC-MAC based on AES encryption
  - confidentiality: AES encryption in counter (CTR) mode
  - same 128-bit key for integrity and confidentiality (from PTK)
  - 48-bit packet number to prevent replay attacks

IEEE 802.1X

- Standard for port-based network access control
- Entities
  - supplicant = station
  - authenticator = access point
  - authentication server
- Port-based: supplicant can access only the authentication server until the authentication succeeds
- Authentication server does not have to be implemented on the access point → little overhead for the access point

EAP Authentication Methods

- Extensible framework, not a specific authentication mechanism
- Example methods
  - EAP-TLS: based on public-key certificates
  - EAP-GPSK (Generalized Pre-Shared Key): based on secret keys shared by the client and the server, uses symmetric-key cryptography

3 IPSec

- Collection of protocols and mechanisms, standardized by the Internet Engineering Task Force (IETF) in a series of publications
- Provides
  - data confidentiality and integrity
  - source authentication (prevent address spoofing, i.e., sending from fake address)
  - protection against packet replay
- Below the transport layer (TCP or UDP) → transparent to applications
- End-to-end security between two hosts, a host and a network, or between two networks
- Example Applications of IPSec
  - Secure remote access over the Internet
  - Secure virtual private network

3.1 Transport Mode and Tunnel Mode

- Transport mode
  - protects the payload of the IP packet
  - typically host-to-host communication
- Tunnel mode
  - protects the entire IP packet by encapsulating it in the payload of a new IP packet
  - typically host-to-network or network-to-network communication

	Protocol	
	Authentication Headher (AH)	Encapsulating Security Payloads (ESP)
Modes	both transport and tunnel	
Provides	integrity, replay protection	integrity, confidentiality, replay protection
Protects	payload and IP header	payload

Authentication Header

- Services
  - data and origin integrity
  - replay-prevention
- Message authentication

  - computed from immutable fields of the IP header, AH header (except ICV), and original payload
  - algorithms: HMAC-MD5, HMAC-SHA-1, HMAC-SHA-2, ...

Encapsulating Security Payload

- Services: confidentiality, integrity (optional), replay prevention
- Encryption: AES-CBC, 3DES-CBC, ...
- Message authentication: HMAC-SHA-1, AES-GMAC, ...
- Authenticated encryption: AES-GCM

Combining Modes and Protocols

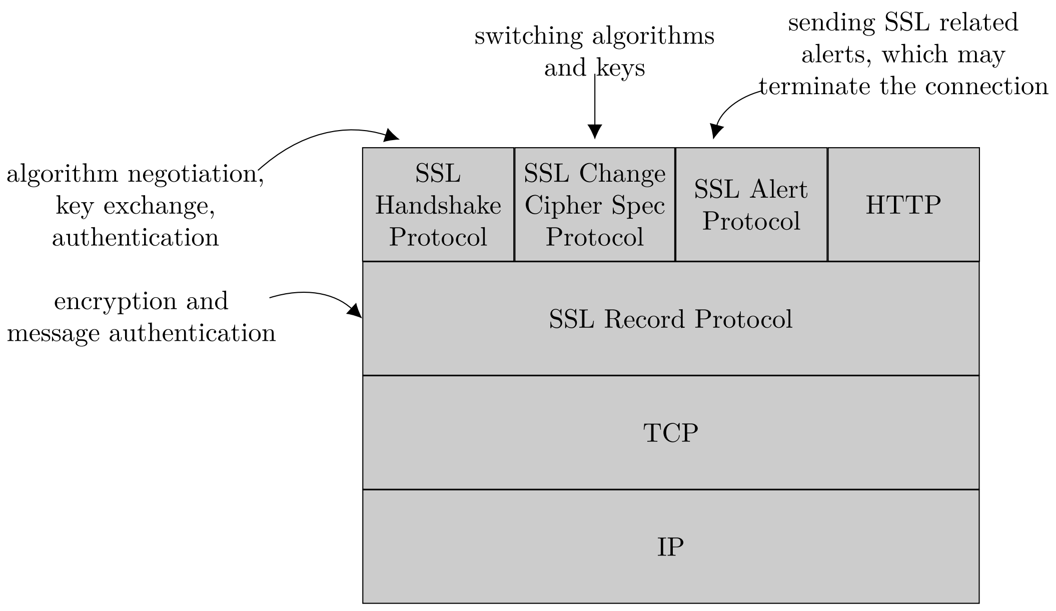
- Tunnel mode advantage: requires support only at the gateways
- Transport mode advantage: requires support only at the hosts
- AH advantage: authenticates some elements of the original header
- ESP advantage: protects both integrity and confidentiality
- Combining modes
  - IPSec tunnel can carry any IP packet $\rightarrow$ IPSec transport or tunnel packets can be sent through an IPSec tunnel
  - IPSec transport can protect any IP packet $\rightarrow$ IPSec transport or tunnel packets can be protected by outer IPSec transport
  - ...
  - can be nested to any depth
- Combination Examples
  1. AH in transport (for integrity) + ESP in transport (for confidentiality)
  2. IPSec packets over tunnel

4 SSL / TLS

Secure Socket Layer

- End-to-end security between two applications
- Endpoint applications can implement it without the help of the operating systems or any intermediate devices
- Developed by Netscape for securing HTTP $\rightarrow$ HTTPS = HTTP over SSL
- very widely used, not just for HTTP (e.g. FTP, POP3, IMAP)

Overview


switching algorithms and keys
sending SSL related alerts, which may terminate the connection
algorithm negotiation, key exchange, authentication
encryption and message authentication
SSL Handshake Protocol
SSL Change Cipher Spec Protocol
SSL Alert Protocol
HTTP
SSL Record Protocol
TCP
IP

SSL Record Protocol

- Security 528
  - confidentiality: symmetric-key encryption (AES GCM, Salsa20, . . .)
  - integrity: message authentication codes based on symmetric-key cryptography (HMAC-SHA256, . . .)
- Additional Services
  - fragmentation: fragment application data into records of at most 16 KiB
  - lossless compression: optional (default is no compression)

SSL Handshake Protocol

- Phase 1: establish security capabilities
  - client_hello:
    - highest SSL version supported by the client
    - nonce (timestamp + random value)
    - cipher suite: list of key-exchange methods, as well as encryption and MAC algorithms
    - compression method: list of supported compression algorithms
  - server_hello:
    - highest SSL version supported by both the client and the server
    - nonce
    - chosen cipher suite and compression method
- Phase 2: server authentication and key exchange
  - certificate (optional): X.509 certificate (may be a chain)
  - server_key_exchange (optional):
    - parameters for Anonymous or Ephemeral Diffie-Hellman exchange
    - public-key for RSA exchange if the certificate contains only a signing key
    - signed by the server (together with the nonces)
  - certificate_request (optional): ask client for an X.509 certificate
  - server_hello_done: server is finished
- Phase 3: client authentication and key exchange
  - certificate (optional): X.509 certificate if the server asked for a client certificate
  - client_key_exchange:
    - pre-master secret encrypted using public RSA key of the server

    - parameters for D-H exchange
  - certificate_verify (optional): digital signature of all previous handshake messages

Key Exchange Methods

- Goal: exchange or agree on a pre-master secret (PMS)
- RSA
  - client generates pre-master secret (PMS)
  - sends PMS to the server encrypted using RSA public-key encryption
- Diffie-Hellman protocol
  - anonymous D-H: basic D-H with no authentication
  - fixed D-H: D-H parameters of the server (X_A and Y_A) are fixed and Y_A is contained in a digital certificate
  - ephemeral D-H: D-H with authentication

SSL Change Cipher Spec Protocol

- Same for client and server
  - change_cipher_spec: signals that the communication party is switching to the negotiated cryptographic algorithms and keys
  - finished: hash value computed from the master secret and all handshake messages using HMAC with SHA hash function

Session Resume

- Authentication and key exchange are complex $\rightarrow$ result needs to be reusable
- Session
  - association between a client and a server
  - cipher suite, compression method, and master secret
- Connection
  - within a session
  - keys and IVs for encryption and message authentication
- Session ID: identifies a session
  - sent in ClientHello $\rightarrow$ may specify an existing session to be resumed
  - sent in ServerHello $\rightarrow$ server can accept resume by sending the same ID
- Session resume skips all messages in the Handshake after the ClientHello and ServerHello messages
  - new keys and IVs are generated from the nonces in the Hello messages

Transport Layer Security

- only minor differences compared to SSL 3.0:
  - pseudorandom function for generating keys and MAC is based on HMAC
  - variable length padding (may prevent traffic analysis)
  - other minor changes

HTTPS

- HTTP over SSL/TLS
- Between the web browser and server:
  - HTTP client = SSL client
  - HTTP server = SSL server
- Conventions

  - URL: `https://` instead of `http://`
  - default TCP port is 443 instead of 80
- HTTP request may be sent after SSL Finished messages
- Protected information
  - URL, contents of the document, browser forms, cookies, headers
  - page served over HTTPS may include elements retrieved using HTTP

5 DNSSEC

Domain Name System (DNS)

- Millions of domain names → distributed database
  - dynamic data ← IP address for a host may change
  - decentralized authority ← each name has an owner
- Hierarchical name space
  - domain: node in the DNS tree
  - for each domain, there is an authoritative server
- Authoritative server
  - responds to queries about the domains for which it is responsible
  - may refer to other authoritative servers for a subdomain

DNS Queries and Responses

- Transport protocol
  - UDP port 53
  - TCP for long responses (and some tasks between nameservers)
- Messages: query and reply
  - 16-bit identification field: match queries with replies
- Caching
  - received responses are cached by servers
  - each record has a Time-to-Live field, after expiry it must be queried again

DNS Weaknesses

- DNS responses are not authenticated
  - responses can be sent over UDP transport protocol → anyone can respond from a spoofed (i.e., fake) IP address to a query
- DNS is a key infrastructure
  - resolvers trust responses, users trust resolvers
  - by tampering with responses, an attacker may direct users to malicious websites or direct e-mail to malicious servers
- DNS cache poisoning
  - attacker sends malicious response to a DNS server, which caches it → malicious response is served to all clients using the server
  - attacker does not have to be man-in-the-middle
- Race to Respond First
  - If the attacker responds before the authoritative server, the DNS cache is poisoned with the malicious IP address
    * fake response is a single UDP packet from the spoofed (i.e., forged) IP address of the authoritative server

Domain Name System Security Extensions (DNSSEC)

- Set of extensions defined by the IETF to the DNS protocol

- Guarantees origin authenticity and data integrity of DNS replies
- Backwards compatible: responses can be interpreted by DNS servers and clients that do not support DNSSEC
  - of course, no guarantees are provided for these servers and clients
- Does not provide confidentiality
  - responses are only authenticated but not encrypted
- Based on public-key cryptography
  - every response is digitally signed

DNSSEC Public Keys and Signatures

- Signature algorithms: RSA-SHA1, RSA-SHA256, ECDSA-SHA256, ...
- Trust anchor
  - known public key for an authoritative nameserver
  - typically included in the operating systems
- Authentication chain

$$\begin{tikzcd}
\text{trust anchor} \arrow[dr, bend right] & & & \\
& \text{root} \arrow[r, "\text{signs}"] & \text{edu} \arrow[r, "\text{signs}"] & \text{uh.edu}
\end{tikzcd}$$

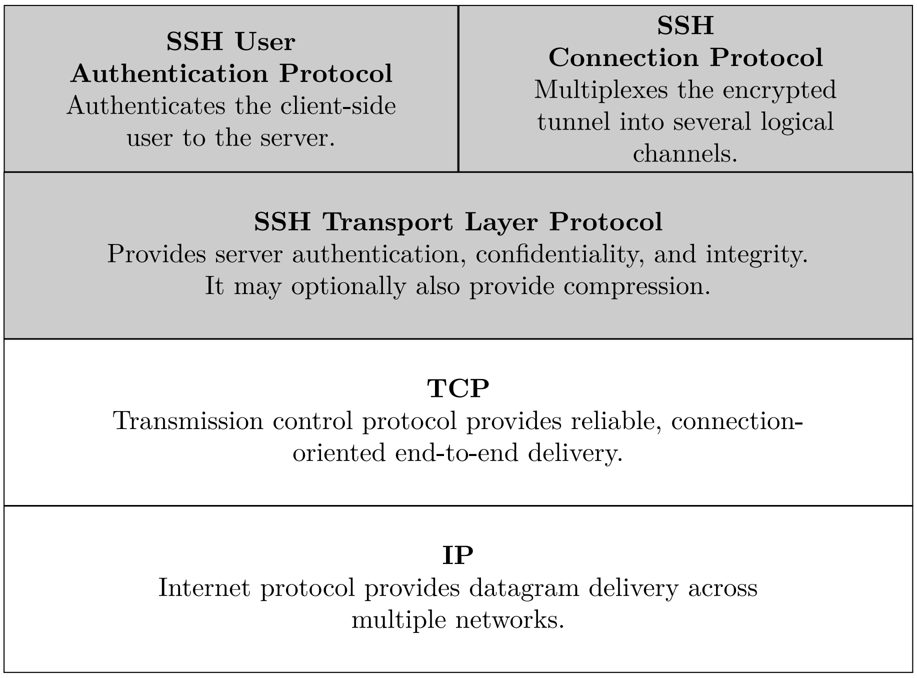
- Responses may include
  - RRSIG: digital signature for the contents of the response
  - DNSKEY: public key for a zone ← if the response delegates

DNS over HTTPS and TLS

- DNS over HTTPS (DoH)
  - DNS queries and responses are encoded into HTTPS requests and responses
    - ∗ provides integrity and confidentiality
    - ∗ server may use HTTP/2 Push to send records in advance
  - controversies and criticism: can impede traffic analysis for cybersecurity, may provide false sense of privacy, can impede traffic filtering by ISPs, ...
- DNS over TLS (DoT)
  - provides integrity and confidentiality

6 SSH

Secure Shell (SSH) Protocol Stack



SSH User Authentication Protocol
Authenticates the client-side user to the server.

SSH Connection Protocol
Multiplexes the encrypted tunnel into several logical channels.

SSH Transport Layer Protocol
Provides server authentication, confidentiality, and integrity. It may optionally also provide compression.

TCP
Transmission control protocol provides reliable, connection-oriented end-to-end delivery.

IP
Internet protocol provides datagram delivery across multiple networks.

SSH Transport Layer

- Packet Exchange
- Identification string exchange
  - both the client and the server send: `SSH-protocolVersion-softwareVersion`
- Algorithm negotiation (KEXINIT)
  - both the client and the server send the list of key exchange, encryption, MAC, and compression algorithms that they support
  - chosen one: first on the client's list that is also on the server's list
- End of key exchange (NEWKEYS)
  - start using the algorithms and keys

Server authentication

- servers signs a hash of all the earlier messages and the new symmetric key
- servers sends the signature and its public key to the client
- Client needs to verify the public-key sent by the server (i.e., verify that the public key belongs to the server host)
- Trust models
  - Certificate authority
    - client accepts public keys that are certified by a trusted CA
  - Local database
    - client has a list of known pairs of hosts and public-keys
    - typically, each user has a list stored in its home directory
      - default location: `~/.ssh/known_hosts`
- Known Hosts
  - First connection: verify and store host and public key
  - Subsequence connections: compare to stored key

SSH User Authentication Methods

- Passwords
  - client sends a username and a password
- Public key
  - client sends a public key and a signature based on the corresponding private key
  - server checks if the public key is acceptable and verifies the signature
  - typically, for every user account on the server host, there is a list of acceptable public keys stored at `~/.ssh/authorized_keys`
- Host based
  - assumes that the server trusts the client host $\rightarrow$ since the client host has already authenticated the user, the server only needs to verify the identity of the client host
  - client sends a signature based on the private key of the client host

7 E-mail security

Weaknesses

- Simple Mail Transfer Protocol
  - Outgoing emails can be tampered (integrity), eavesdropped (confidentiality), or forged (authenticity/integrity)

E-Mail Threats

- Spam: unsolicited messages sent in bulk
- E-mail scam: advance-fee scam (a.k.a. "Nigerian Prince" scam), job scam, ...
- Phishing: collecting sensitive information (e.g. passwords, credit card numbers) or delivering malware by impersonating a trusted entity
- Spear-phishing: phishing directed at specific targets (e.g. users)

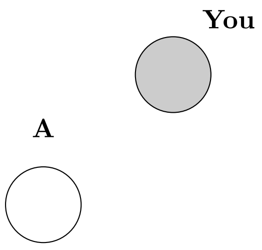
Pretty Good Privacy (PGP)

- General-purpose application for secure communication between users
  - confidentiality and integrity protection for files and e-mail
  - built on widely used asymmetric and symmetric-key cryptographic algorithms
  - communicating users know each other's public keys $\rightarrow$ trust
- IETF standard: OpenPGP
- Software
  - PGP went commercial in 1996
  - GnuPG is a free and open-source implementation of OpenPGP

PGP Key Management

- Each user may have multiple public-private key pairs $\rightarrow$ key identifiers are used to specify which key is used
- Key storage
  - private-key ring: user's own public-private key pairs
    - each entry has user identifier, key identifier, public key, encrypted private key
    - private key is encrypted using a passphrase
  - public-key ring: public keys of other users
    - each entry has user identifier, key identifier, public key, trust levels, and signatures from other users
    - public-keys can be verified directly (e.g. delivery on secure channel) or using the ”web of trust”

PGP Web of Trust



PGP Authentication

- Digital signature using the sender's private key
  - hash using MD5, SHA-1 or SHA-2, and then sign using RSA or DSA
- Message may be compressed after signature
  - ZIP, Bzip2, . . .

PGP Encryption

- Message may be compressed before encryption
  - ZIP, Bzip2, . . .
- Generate a new 128-bit random symmetric key for each message
  - encrypt the message with the symmetric key using a block cipher in CFB mode (3DES, Blowfish, AES, . . .)
  - encrypt the symmetric key with the recipient's public key using RSA or ElGamal

S/MIME

- MIME (Multipurpose Internet Mail Extension)
  - fixes the limitations posed by SMTP
  - New headers fields
    - Content-Type: type of message content
      - multipart type: body contains multiple parts, each having a header (e.g. images in HTML message, attachments)
      - simple types (e.g. text/plain, image/jpeg, text/html)
    - Content-Transfer-Encoding: how binary data is represented in 7-bit ASCII (e.g. Base64, quoted-printable)
- Secure / Multipurpose Internet Mail Extension (S/MIME)
  - security enhancement to the MIME e-mail format standard
  - similar to PGP (Pretty Good Privacy)
    - both S/MIME and PGP enable encrypting and signing messages
    - both have IETF standards
    - both support state-of-the-art algorithms (AES, RSA, SHA-2, ...)
    - S/MIME is likely to emerge as the industry standard for commercial and organizational use (e.g. Microsoft Outlook and Gmail support S/MIME)
    - PGP is likely to remain the choice for personal e-mail security

S/MIME Functionality

- Functions
  - Signed data: message is digitally signed, and both the signature and the message are encoded (using Base64 representation)
  - Clear-signed data: similar to signed data, but only the signature is encoded

    - Enveloped data: encrypted message content and encrypted content-encryption key (i.e., session key) for one or more recipients (encoded using Base64)
    - Signed and enveloped data: signing and encrypting may be nested
- MIME content types
    - application/pkcs7-mime: signed or enveloped data
    - multipart/signed: for clear-signed data, which contains a message and a signature (signature part is application/pkcs7-signature)

S/MIME Public Keys

- Public-key certificates
    - based on X.509 digital certificate format
    - similar to PGP, digital certificates are distributed manually
    - however, certificates may be signed by a CA
- Public keys are used for
    - verifying signatures
    - encrypting session keys
        * for enveloped data, the sender generates a random session key
        * session key is encrypted with each recipient's public key, and the message contents are encrypted (using symmetric-key crypto) with the session key
        * upon receiving the message, a recipient can decrypt the session key, and then decrypt the message contents using the session key

DKIM

- E-Mail Spoofing Problem

- Attackers may use e-mail with forged sender addresses for spam/phishing

- Limitations of PGP and S/MIME

    - depend on the sending and receiving users, who must install or configure software, share public keys, etc.
    - do not sign the message header, only the message contents

- DomainKeys Identified Mail (DKIM)

    - specification for signing e-mail messages
    - implemented on the servers, therefore transparent to the users

DKIM Signature and Verification

- Signature: Dkim-Signature header field
    - v: version
    - a: algorithm used for signature (RSA-SHA1, RSA-SHA256)
    - d: domain name
    - s: selector (if there are multiple public keys)
    - h: list of signed header fields
    - bh: hash of the body part of the message
    - b: signature in Base64
- Verification
    - receiving SMTP server queries record s._domainkey.d using DNS
    - nameserver returns the public key corresponding to the signing private key

E-Mail Authentication Solutions

- DomainKeys Identified Mail (DKIM)

- Sender Policy Framework (SPF)
  - another system for preventing e-mail spoofing
  - DNS record lists all authorized sending hosts (i.e., e-mail servers) for a domain → receiving server can verify
  - may be combined with DKIM
- Domain-based Message Authentication, Reporting and Conformance (DMARC)
  - built on top of DKIM and SPF, published using DNS record
  - enables domain owners to publish a policy (combination of DKIM and SPF) for verifying the legitimacy of e-mails from the domain

8 Authentication and access control

Authentication

- Authentication: reliably verifying the identity of someone or something
  - computer authenticates another computer
  - computer authenticates a user
- Types of authentication
  - one-way authentication or mutual authentication
  - one-time or establishing a session (e.g. combined with key exchange)
- Typical methods of computer authentication
  - cryptography-based
  - address-based

User Authentication

- Types of user authentication factors
  - knowledge: some secret known only by the user (e.g. password)
  - ownership: some physical object possessed by the user (e.g. bank card)
  - inherence: some physical characteristic of the user (e.g. fingerprint)
- Password-based user authentication
  - typical form of knowledge-based authentication
  - verifier stores the password in a database or file
  - often combined with cryptography-based approaches to protect the password from eavesdropping
  - password must be easy to remember but hard to guess

Password-Based Authentication

- Problem: easy-to-remember passwords are weak
  - Miller's law: number of objects an average human can hold in working memory is 7 ± 2
  - length of passwords that users can easily remember (i.e., not write down somewhere) is very limited
- Brute-force attack: password guessing
  - online: attacker must rely on the verifier to test the correctness of a password → verifier can limit the number of attempts (e.g. number of unsuccessful login)
  - offline: attacker can test the correctness of a password on its own

Password Storage

- Cleartext passwords are insecure
  - system administrators (and other local users) may easily read passwords
  - attackers who have compromised a system may be able to read passwords

- Users tend to reuse passwords → breach may affect other systems as well
- Store the cryptographic hash of the password
  - during authentication, the user enters the plaintext password, and the verifier computes its hash and compares it with the stored hash
  - attacker can perform offline guessing to recover the plaintext password
- Brute-forcing multiple hashed passwords
  - first, precompute a table of [password, hash] values for possible passwords
  - second, for each hashed password, look up the precomputed hash value

Salting

- Before hashing a password mix it with a salt value
  - both when the password is set and during verification
  - verifier stores: username, salt, H(password + salt)
- Salt value
  - randomly generated for each user account
  - may be stored in plaintext by the verifier
- Salt values do not have to be memorized → strong randomness
  - prevents precomputing hashes since the attacker cannot consider all possible salt values (different salt values require different precomputation)
  - also hides identical passwords, which would result in identical hashes
- However, it does not make guessing a single password harder (assuming that the attacker knows the salt)

Multi-Factor Authentication

- User is authenticated only after passing multiple independent authentication mechanisms
  - typically, each mechanism is built on a different type of factor (e.g. knowledge + possession), so it is independent of the other mechanisms
  - attacker must circumvent all authentication mechanisms to succeed
- Possession factors
  - disconnected token: not connected to the client computer, typically the user manually enters authentication data displayed by the token
  - connected token: physically connected to the client computer (e.g. USB token)
- Inherence factors
  - includes fingerprint, face, voice, or iris recognition

Access Control

- Access control (i.e., authorization): approving or rejecting access requests
- Abstractions
  - subjects: entities that can perform actions on the system
  - objects: resources to which access must be controlled
- Control access to objects based on a policy

Discretionary Access Control (DAC)

- Allows access rights to be propagated at the subjects' discretion
- Often implemented using the notion of owner
  - every object has an owner subject, who can set the permissions for that object
- Used by popular operating systems (e.g. Unix and Windows)
- Problem: non-malicious users are not necessarily trustworthy
  - phishing: subjects may be tricked into propagating their access rights to malicious entities

  - malware: malicious code running with a subject's credentials can disclose or modify sensitive information
  - large organizations working with sensitive data may need centralized control

Mandatory Access Control (MAC)

- Restricts the access of subjects to objects based on a system-wide set of rules
  - system-wide rules are set by a central authority (e.g. system administrator)
  - policy is mandatory → users do not have full control over access to the resources that they create
- Traditionally used for implementing multilevel security
  - objects have security classifications (e.g. ”Top Secret”, ”Secret”)
  - subjects have security clearances
- Available in some form on many modern operating systems
  - SELinux and AppArmor for Linux, and Mandatory Integrity Control for Windows
- May be combined with DAC: grant access only if both DAC and MAC permit the access

Access Control Models

- Access control list (ACL): list permissions for each object
  - for each object, list pairs of [subject, access right]
- Role-based access control (RBAC): row oriented
  - create a set of roles (e.g. based on real-world job functions), and assign a role (or roles) to each subject
  - for each role, list pairs of [object, access right]

Unix Access Control

- user: has a unique UID (special UID = 0 for root user)
- group (collection of multiple users): has a unique GID
- Access control abstraction
  - subject = process
    - has an effective UID and GID (as well as real and saved UIDs and GIDs)
  - object = file
    - has an owner (UID) and a group (GID), typically inherited from the process that created the file
    - almost everything is a file on a Unix system (regular files, directories, devices, Unix domain sockets, . . .)
  - Each file has 12 permission bits
    - read, write, and execute permission for owner, group, and others
    - set user ID (setuid), set group ID (setgid), sticky bits
  - When a process wants to read/write/execute a file,
    1. if effective UID = file owner → use read/write/execute permission for owner
    2. else if effective GID = file group → use read/write/execute permission for group
    3. else → use read/write/execute permission for others
  - For directories,
    - read means listing the contents of the directory
    - write means creating, renaming, and deleting files in the directory
    - execute means accessing the files (and directories) within the directory (must also have execute permission on all the parent directories)

Sticky, Set UID, and Set GID Bits

- Sticky bit

  - when set on a directory, files within that directory can be renamed or deleted only by their owners, the directory owner, or a superuser
  - for example, sticky bit is typically used on the /tmp directory
- Set UID bit
  - when set on an executable file, the effective UID of a process executing the file is set to the file owner UID
  - for example, set UID bit is typically used on the passwd command
- Set GID bit
  - when set on an executable file, the effective GID of a process executing the file is set to the file group GID
  - when set on a directory, new files created within will inherit the GID of the directory

9 Software vulnerabilities and countermeasures

Buffer Overflow

- Buffer overflow / buffer overrun: anomalous condition where a process tries to store data beyond the boundaries of a fixed-length buffer
  - extra data overwrites adjacent memory, which may lead to denial-of-service or arbitrary code execution
- Vulnerable programming languages
  - C, C++ and other ”lower-level” languages without bounds checking
  - “higher-level” languages, such as Java and C#, are generally not vulnerable
- Attacker can supply malicious (i.e., long) input
  - remotely through a network connection (e.g. specially crafted HTTP request)
  - locally (e.g. by sending a specially crafted file to the target user)

Typical Buffer Overflow Exploits

- Attacker’s goal: execute arbitrary code (i.e., code chosen by the attacker)
- execute arbitrary code (i.e., code chosen by the attacker)
- Stack-based exploitation (”stack smashing”)
  - overflow a local buffer
  - overwrite local variables, function return addresses, exception handlers, etc.
- Heap-based exploitation
  - overflow a dynamically allocated buffer
  - overwrite other data, function pointers, etc.
- Shellcode: small piece of code, which allows the attacker to control the compromised machine

Buffer-Overflow Exploit Countermeasures

- Compile-time – hardening new software
  - programming languages
  - safe functions and libraries
  - compiler extensions
- Run-time – protecting existing software
  - executable space protection
    * Lot of exploits build on injecting and executing malicious code
    * By separating the memory space of a process into executable and modifiable parts, code injection can be prevented
    * Problem: modern computer architectures do not separate code from data

        - Limitation: cannot fully protect programs that create and execute code at runtime (e.g. just-in-time compilers)
        - Circumventing Executable Space Protection
            - Attacker can re-use existing code from the memory space of the process for malicious purposes
            - Return-to-libc attack
            - for most processes, the standard C library is loaded into memory
            - attacker can change the return address of a function to point to the beginning of a function in the C library
            - common target: system function – takes as argument a string, and executes it as a system command with the privileges of the process
            - attacker has control over the stack → attacker can set up parameters for the C library function
    - address space layout randomization
        - In order to reliably jump to an exploited code, the attacker needs to know its address
        - Address Space Layout Randomization (ASLR)
            - randomly arrange the positions of the executable, the stack, and the heap in the process's address space
            - may prevent return-to-libc attacks
            - most operating systems (e.g. Windows, Linux) implement some randomization
        - Counter-countermeasures
            - information leakage (e.g. printf vulnerability)
            - random guessing (Heap spraying)

Integer Overflow

- Integer overflow occurs when an arithmetic operation leads to a value that is greater than the maximum value that can be stored
- Affects most languages - even some managed languages, such as Java and C#, are susceptible to integer overflow errors
- Exploiting integer overflow errors
    - calculating indexes into arrays
    - calculating the amount of space to allocate for a buffer
    - checking whether an overflow could occur

Input Validation

- Sources of input
    - user supplied files and terminal input
    - command line arguments
    - environment variables
    - function calls from other modules
    - network packets (web applications in detail later)
    - ...

Format String Vulnerabilities

- Vulnerable functions
    - `sprintf`: writes to buffer
    - `fprintf`: writes to file
    - other members of the printf family (e.g. `snprintf`)
    - `printk`: used in the Linux kernel
    - other functions that use format strings (e.g. `syslog`)
- Format placeholders

    - `%s` – string
    - `%d` – number (output in decimal format)
    - `%x` – number (output in hexadecimal format)
    - Special placeholder: `%n`
        - argument must be a pointer to a signed integer, where the number of characters printed so far will be written

- When variable is printed directly, introduces Vulnerability
    - `printf(name);`
    - if `name` is provided as `%d%d%d` then the top three stack values will be printed

- To prevent, use string formatting
    - `printf("%s%", name);`
    - if `name` is provided as `%d%d%d` then `"%d%d%d"` will be printed

Race Conditions

- Race condition
    - when results depend on the sequence or timing of uncontrollable events
    - for example, when the output of a software depends on how the operating system schedules the execution of multiple processes or threads
- Typically happens when interacting with
    - memory shared by multiple processes (or threads)
    - file system
    - signals and other interprocess communication mechanisms
- Race condition bugs and errors
    - happen when events do not occur in the intended order
    - typically very difficult to reproduce and debug
- Attack approaches
    - try multiple times (if possible)
    - slow down the target process
    - increase computational load on the machine
    - computational complexity attacks
- Preventing Race Conditions
    - Time of check to time of use
        - we cannot allow any changes in this interval
        - trying to make it short is not enough
    - Prevention techniques
        - work with file descriptors instead of filenames
        - rely on filesystem access checks
        - be careful with directories that are writable by everyone (e.g. `/tmp/`)
        - lock resources (files, databases, etc.)
        - look out for non-atomic operations (e.g. `num++`)
        - synchronization (e.g. semaphore, mutex)

10 Web vulnerabilities

File Inclusion Vulnerabilities

- Passing unvalidated user input to functions that load files
    - remote: webserver loads a file from another server
        - Attacker may run arbitrary PHP code on the vulnerable webserver
        - Prevention in PHP: disable allow_url_include

        · disabled by default in newer versions
        · prevents the inclusion of remote files
  - local: webserver loads a local file
    * Directory traversal
      · attacker may read any file (to which the webserver process has access)
    * Uploading PHP code to the webserver
      · enables the attacker to execute arbitrary code on the webserver
- Susceptible functions in PHP:
  - `include(file)`, `require(file)`, `include_once(file)`, `require_once(file)`
  - parse the content of the specified file (anything between <?php and ?> is interpreted, anything outside is sent to the output)
  - used to load PHP libraries, classes, etc.
- Similar functions in other languages
- Prevention: do not trust user input at all
  - always validate input before using it
  - do not use values directly (e.g. `switch` statement)

File Upload Vulnerability

- File upload form
- Attacker may
  - upload malicious PHP code `attack.php` → arbitrary code execution with `GET attack.php`
  - upload, for example, `.htaccess` (per-directory configuration file for Apache webserver) → circumvent access control
  - overwrite existing files

Command Injection

- Example: sendmail.php:

```
<?php $address = $_GET["email"];
system("mail $address -s 'Welcome to example.com!'< /var/www/welcome.txt");
?>
```

- exploit:
  - `GET sendmail.php?email=foo@bar.com; rm important_file;` → `system("mail foo@bar.com; rm important_file; -s ...`

SQL Injection Vulnerability

- Attacker may execute any statement
  - delete an entire table: `'; DROP TABLE users; -`
  - change the password of another user: `'; UPDATE users SET password='pwnd' WHERE email = 'victim@example.com'; -`
- Some databases even allow running system commands
  - example: Microsoft SQL Server `xp_cmdshell`
- Attempts to Prevent SQL Injection
  - Escape special characters
    * idea: add backslash \before characters ', ", \, and `NUL` to escape them
    * however, this is not as simple as it sounds
  - Secret table and column names
    * attacker can guess and test them
    * most databases have queries for reading the database and table schema

            · if query results are displayed, attacker can read the schema

- Preventing SQL Injection
  - Properly escape special characters
  - Object-relational mapping (ORM) framework
    - eliminates the need to develop potentially buggy and vulnerable custom code
  - Prepared / parameterized SQL statements
    - execute the same query or statement repeatedly, only changing certain parameter values between the executions
    - advantageous for both performance and security
      - statement needs to be parsed and optimized only once
      - statement cannot be changed using SQL injection

Cross-Site Scripting

- A cross-site scripting (XSS) vulnerability enables an attacker to inject client-side script code into pages generated by a web server
  - attacker can use an XSS vulnerability to run malicious scripts on clients (i.e., web browsers) that view the webpage, and steal their private information, credentials, send requests in their name, etc.
  - Types (note that there is also DOM-based XSS)

  - Stored XSS
    - attacker tricks the webserver into storing and serving a malicious script
    - malicious script is sent to clients visiting the webpage
  - Reflected XSS
    - attacker tricks a client into sending a specially crafted request to the server
    - malicious script is reflected back to the client as part of the response

- Preventing XSS
  - Validate and filter all input data on the webserver
    - even if they are not processed by the webserver
    - some web browsers also try to detect reflected XSS by testing if a script received from the server was sent in the request
  - Scripts may be specified in
    - script tag: `<script>`
    - event handlers: `onClick`, `onLoad`, `onSubmit`, ...
    - ...
  - Remove or encode all special HTML characters
    - encoding: `&lt;` instead of `<`, `&quot;` instead of `"`

Cross-Site Request Forgery

- A cross-site request forgery (CSRF) vulnerability enables an attacker to trick a user into sending malicious requests to a webpage
- Similar to reflected XSS, an attacker has many options for tricking the user into opening a malicious URL
  - sending link in e-mail or instant message
  - displaying link on some other website
  - ...
- Attackers might use a CSRF vulnerability to
  - open a backdoor on a server
  - initiate a wire transfer on an online banking website
  - ...
- May enable bypassing other access control methods
  - for example, if access to a server is limited to clients on the same local network

- Prevention
  - Webserver has to verify that the request originates from a page belonging to the website (i.e., user ”clicked” a link on the site, instead of opening a link from some other source)
  - Techniques
    - synchronizer / secret token
      - pages generated by the server include a secret input field
      - when a user performs an action on the website, the value of the secret field is sent in the request → server can verify origin of the request
    - referrer validation
      - HTTP requests may include a header field that identifies the address of the page that linked to the request resource
      - however, referrer might be missing (or, in some cases, forged)

11 Malware

- Malware (short for malicious software): software used to disrupt computer operations, gather sensitive information, or gain access to private systems
- Backdoor
  - secret entry point into a system or program that circumvents the usual security access procedures
  - maintenance hook: legitimate use for debugging and testing
  - asymmetric backdoor: can be used only by the developer, even if the implementation becomes public
  - may be introduced by a malicious compiler
- Logic bomb
  - code embedded in a legitimate program, which ”explodes” when certain conditions are met (e.g. time, presence of some files, hostnames)
  - may alter or delete data, system functionality, etc.
- Trojan Horse
  - an apparently benign application that has some hidden malicious functionality
  - Runs with and abuses the privileges of the victim user
  - May implement various kinds of malicious functionality
  - Typically spread using social-engineering techniques
    - providing free (but illegal) copies of commercial software
    - sending as an e-mail attachment
    - drive-by download: authorized by the user without understanding the consequences
- Virus
  - parasitic self-replicating malware
  - resident virus: remains in memory (e.g. as part of the OS), may overwrite interrupt handlers and other functions
  - Infection targets
    - boot sector: targets the boot sector or Master Boot Record of the infected host’s hard drive or removable media
    - executable files: targets binary executable files
    - documents (macro viruses): targets word processor and spreadsheet documents that support embedding macro programs (e.g. Microsoft Office)
  - Typical phases of operation
    1. dormant: virus is waiting for an external event
    2. propagation: virus places copies of itself into other executables and files
    3. triggering: virus is activated by some external event
    4. execution: virus executes the payload, which performs malicious actions

- Worm
  - runs independently, without a host program or file
  - Propagates a copy of itself to other systems
    - typically by exploiting vulnerabilities, such as common software vulnerabilities or weak passwords
    - over the Internet, over a local computer network, or through removable media (e.g. USB flash drives)
    - e-mail worm: propagating as e-mail attachments
  - Payload
    - either carried by the worm or downloaded from a server
    - may perform other tasks (e.g. spyware, ransomware, botnet)

Spyware, Adware, Cryptojacking, Randomware

- Spyware
  - collect sensitive information about the user
  - may record keystrokes, mouse clicks, browsing activity, etc.
  - attacker can use it to collect personal information (e.g. social security number, financial information, such as credit card numbers)
- Adware
  - display unwanted advertisement to the user (e.g. pop-up windows, injecting into web pages, ...)
  - legitimate use: advertisement-supported non-malicious software
- Cryptojacking
  - use the infected computer's resources to "mine" proof-of-work cryptocurrencies (e.g., Bitcoin) for the attacker
- Ransomware
  - holds a computer system or data hostage
  - disables access to the computer system or certain files on it (e.g., screen locking, browser locking)
  - may also threaten to publish the victim's files online
  - "scareware": threaten or deceive users (e.g., fake antivirus)
  - Payment
    - computer system or data is released upon payment
    - payment through Bitcoin, premium-rate text messages, wire transfers, etc.
  - Crypto-ransomware
    1. ransomware encrypts files using a random symmetric key
    2. symmetric key is encrypted using the attacker's public key and then deleted
    3. attacker decrypts the symmetric key in exchange for payment

Botnets

- collection of computers controlled through the Internet
  - legal application: distributed computing
  - illegal application: taking advantage of compromised "zombie" computers
- Zombie computers
  - laptop and desktop computers
  - IoT devices (e.g., remote cameras, home routers, DVRs)
- Typically used by attackers to
  - send spam e-mail or perform click fraud
  - collect personal information $\approx$ spyware
  - attack other systems (e.g., denial-of-service attack by flooding with requests)
- Architectures

  - Client-Server
    * publish digitally signed commands on websites, IRC channels, or other standard protocols
    * C&C servers are typically redundant, and often implemented on compromised computers
  - Peer-to-Peer
    * commands distributed over a peer-to-peer (P2P) network
    * no single (or few) point of failure → more resilient to shutdown or hijack

Rootkit

- enables unauthorized access to a computer system
- hides its own existence
- Typically installed by attackers after compromising a system
  - provides backdoor for easy reentry
  - keeps the compromise hidden from operators → persistent and stealthy compromise
- Installation
  - typically requires root (or administrator) access
  - may be performed manually or automatically
- Techniques for hiding
  - Rootkit must be installed and run on the compromised system → without hiding, it might be visible as a file or running process
  - Modifying system commands and applications
    * ls: lists files in a directory → modify to omit the rootkit files
    * ps: lists running processes → modify to omit the rootkit proce
  - Modifying or disabling event logging
    * system logs may provide evidence of an attack → remove log entries
  - Disabling anti-virus and anti-rootkit software
  - Modifying the kernel, system drivers, and other core parts of the OS
    * rootkits may tamper with any system call, security mechanism, etc.

Malware Detection

- "Arms race" between malware scanners and concealment techniques
- Signature-based malware scanner: detects known viruses based on patterns
- Polymorphic malware: "mutates" (i.e., changes) itself to avoid signature-based detection
- Encrypted malware
  - common technique for polymorphic malware
  - most of the malware is encrypted, except for a short code that can decrypt it
  - decryption code must be in plaintext → signature for self-decryption (very rare for non-malicious software)
- Metamorphic malware: changes the entire code upon each infection (including, e.g., decryption code)
- Heuristic rule-based malware scanner: identify code fragments commonly used by malware
- Behavior-based malware detection: detect behavior characteristic to malware (e.g., sending data to C&C servers)
- White-list known good programs, verify integrity

12 Secure development

compile-time buffer-overflow prevention

- Programming Languages and platforms that do not allow direct memory access typically prevent buffer overflows
  - C# and other managed .NET languages
  - Java, Python, PHP, Perl

    - However,
        * safety and security may come at the cost of lower performance (bounds checking can increase execution time)
        * native code invoked from these languages is still vulnerable (e.g. through Java Native Interface)
        * be careful with libraries that might rely on native code
        * there might be vulnerabilities in the interpreter/virtual machine of the language (e.g., JVM or CLR)
- Safe Functions

Unsafe	Less unsafe
`strcpy(char *src, char *dst)`	`strncpy(char *src, char *dst, size_t n)` (not really safe) `strlcpy(char *src, char *dst, size_t n)`

`strcat(char *s1, char *s2)`	`strncat(char *s1, char *s2, size_t n)`
	`strlcat(char *s1, char *s2, size_t n)`

`sprintf(char *str, char* format, ...)`	`snprintf(char *str, size_t n, char* format, ...)`
`gets(char *str)`	`fgets(char *str, int n, FILE *file)`

- Safe Libraries
    - e.g. String class `std::string`; mostly safe, but `[]` operator can cause issues
    - same with Array class `std::vector`
- Stack Canary Compiler Extension
    - at the beginning of the function, a special value is placed on the stack between the local variables and the return address
    - buffer overflows that reach the return address will necessarily overwrite the special value
    - before returning, the function checks if the value is intact

Secure Programming/Defensive Programming

- Techniques and approaches for improving the source code
- Avoiding typical errors that lead to common vulnerabilities
- assume nothing (e.g., input length)
- expect everything (e.g., errors and misuse)
- i.e., “be paranoid”
- Use safe functions (e.g., never use `gets`, use `fgets` instead), use safe libraries
- Do not use input in vulnerable functions directly (e.g. file inclusion, string output)
- Beware integer type conversions and promotion
- Do not allow changes between time of check and time of use to prevent race condition vulnerabilities
- Use prepared statements for SQL queries
- Sanitize user input before sending it to other systems (e.g., XSS)
- Validate the sources of HTTP requests to prevent CSRF
- ...

Input Validation

- blacklist
    - list known bad inputs
    - do not allow inputs that are on the blacklist
    - listing all bad inputs is difficult and error-prone
    - typically has low impact on usability
- whitelist
    - list known good inputs
    - allow only inputs that are on the whitelist
    - typically more secure

Code Analysis

- static
  - input: typically source code
  - finds vulnerabilities by considering possible executions of the source code
  - typically "white-box" approach
- dynamic
  - input: typically binary (compiled) code
  - finds vulnerabilities by executing the code with various inputs
  - typically "black-box" approach

- Static analysis may find the root cause of a problem, while dynamic analysis can detect only when the problem occurs

Taint Analysis

- attempts to identify which variables and objects can be modified by untrusted user input
  - if the source of the value of a variable is untrustworthy (e.g., network packets, user files, user input), then the variable is tainted
  - any variable whose value is derived from (e.g., through mathematical or string operators) from a tainted variable is also tainted
  - tainted variable passed to a vulnerable function without sanitization $\rightarrow$ potential software vulnerability
- Information flow analysis
  - assign a security level to each source
  - each object inherits the security levels of the sources and other objects from which it is derived
  - may be used to detect the leakage of confidential information
- In practice
  - static: analyze source code $\rightarrow$ rewrite source code if vulnerability is found
  - dynamic: keep track of tainted variables during run time $\rightarrow$ throw an exception

13 Detection

14 Isolation

15 Denial of Service attacks

16 Vulnerability scanners