T7. Securitatea în Mediile Cloud și Mobile. Provocări specifice pentru aplicațiile cloud.
T8. Securitatea în Mediile Cloud și Mobile. Securitatea aplicațiilor mobile

1. Introducere

Conținutul prelegerei

- Definiția și importanța securității aplicațiilor în mediile cloud și mobile
- Diferențele principale dintre securitatea cloud și securitatea mobilă
- Exemple de incidente de securitate în cloud și mobile (ex. scurgeri de date în servicii cloud, malware mobil)

Mediul Cloud și Cloud Computing reprezintă tehnologii esențiale care permit stocarea, procesarea și accesul la resurse informatice prin internet. Aceste tehnologii sunt fundamentale în era digitală, oferind scalabilitate, flexibilitate și eficiență.

Securitatea cloud se referă la ansamblul de politici, proceduri, instrumente și tehnologii pentru a proteja utilizatorii, datele sensibile, aplicațiile și infrastructura în mediile de tip cloud. Altfel spus, securitatea cloud urmărește să prevină accesul neautorizat, furtul sau pierderea datelor și întreruperile serviciilor în sistemele și aplicațiile găzduite în cloud. Un concept fundamental în acest domeniu este modelul de responsabilitate partajată – furnizorul de cloud asigură securitatea infrastructurii de bază, dar clientul este responsabil de protejarea datelor și aplicațiilor proprii. De exemplu, chiar dacă un provider cloud oferă firewall și criptare la nivel de rețea, o configurare greșită sau o aplicație vulnerabilă instalată de client poate duce totuși la o breșă de securitate.

Securitatea mobilă vizează protejarea dispozitivelor mobile (smartphone-uri, tablete) și a datelor stocate sau accesate de acestea împotriva amenințărilor cibernetice. În esență, securitatea mobilă înseamnă aplicarea unor măsuri care să întărească apărarea dispozitivului și a aplicațiilor împotriva riscurilor precum breșe de date, supraveghere neautorizată, ransomware sau alte atacuri malițioase. Acest domeniu include atât securitatea hardware și a sistemului de operare (ex: blocarea dispozitivului cu cod/PIN sau date biometrice, actualizarea periodică a sistemului de operare), cât și securitatea aplicațiilor mobile instalate (ex: permisiuni acordate aplicațiilor, criptarea comunicațiilor efectuate de acestea).

Importanța securității în mediile cloud și mobile este tot mai mare având în vedere adoptarea pe scară largă a acestor tehnologii. Milioane de utilizatori și organizații își stochează datele în

cloud și utilizează zilnic aplicații mobile pentru servicii bancare, comunicare, sănătate, etc. Statisticile arată că peste 83% din populația globală deține un smartphone, iar aplicațiile mobile au devenit adesea unica interfață prin care utilizatorii accesează servicii digitale. Această popularitate atrage însă și atenția infractorilor cibernetici. Un studiu al serviciilor de informații indică faptul că aplicațiile mobile reprezintă principala țintă, fiind folosite în 86% din atacurile cibernetice ca vectori de atac. Totodată, tot mai multe organizații își migrează infrastructura și datele către cloud pentru eficiență și scalabilitate. În consecință, securitatea devine o preocupare centrală: o breșă într-un mediu cloud poate expune volume mari de informații sensibile, iar o vulnerabilitate într-o aplicație mobilă populară poate compromite confidențialitatea a milioane de utilizatori.

În concluzie, atât cloud-ul, cât și ecosistemul mobil aduc beneficii imense, dar și riscuri specifice. Securitatea informației în aceste medii nu mai poate fi tratată ca o reflexie târzie, ci trebuie integrată de la început în arhitectură și dezvoltare.

Cloud Computing se referă la livrarea de servicii de computing — inclusiv servere, stocare, baze de date, rețelistică, software, analiză de date și inteligență artificială — prin intermediul internetului ("cloud") pentru a oferi inovații mai rapide, resurse flexibile și economii de scară.

Beneficii ale Cloud Computing

- **Scalabilitate**: Resursele pot fi mărite sau micșorate rapid în funcție de cerințe.
- **Cost-eficiență**: Plătiți doar pentru resursele consumate.
- **Accesibilitate**: Acces de oriunde, oricând, atâta timp cât există o conexiune la internet.
- **Reziliență**: În caz de defectare a hardware-ului, serviciile și datele sunt gestionate de furnizorii de cloud pentru a asigura continuitatea.

Considerații de Securitate

- **Protecția datelor**: Implementarea criptării și politici de securitate robuste.
- **Conformitate**: Asigurarea că serviciile de cloud respectă reglementările locale și internaționale (de exemplu, GDPR) (General Data Protection Regulation) este Regulamentul general privind protecția datelor adoptat de Uniunea Europeană (2018)
- **Gestionarea accesului**: Utilizarea autentificării multifactor și a permisiunilor bine definite.

2. Provocări specifice pentru aplicațiile cloud

2.1. Modele de servicii cloud și implicații de securitate

- **IaaS (Infrastructure as a Service)** – responsabilități partajate (ex. AWS, Google Cloud, Microsoft Azure). Resurse de computație brute, precum servere virtuale și stocare.
- **PaaS (Platform as a Service)** – protecția codului și a datelor (ex. Google App Engine, Heroku). Seturi de servicii și instrumente de dezvoltare care permit dezvoltatorilor să creeze aplicații web sau mobile.
- **SaaS (Software as a Service)** – securitatea utilizatorilor finali (ex. Google Workspace, MS Office 365). Software-uri disponibile prin abonament, gata de utilizat pentru utilizatori.

Exemple Practice

1. **IaaS**: **Amazon Web Services (AWS) EC2 -** Companiile pot închiria mașini virtuale și pot utiliza propria lor platformă pentru a rula aplicații, gestionând scalabilitatea în funcție de necesități.
2. **PaaS**: **Google App Engine** - Companiile de dezvoltare software pot construi și găzdui aplicații web fără a gestiona infrastructura proprie, concentându-se doar pe inovația software și design-ul aplicației.
3. **SaaS**: **Microsoft Office 365** - Utilizatorii pot accesa suite de productivitate (Word, Excel, Outlook etc.) prin abonament, fără a instala software-ul pe dispozitivele locale, beneficiind de actualizări și suport continuu.

2.2. Principalele amenințări și vulnerabilități

- **Acces neautorizat și identități compromise**

  Compromiterea conturilor administrative sau a cheilor de acces cloud poate avea consecințe grave. Problemele legate de Identity and Access Management (IAM) s-au aflat în topul amenințărilor cloud (locul #2 în 2024). Atacatorii pot fura sau ghici credențiale slab securizate, pot exploata lipsa autentificării multifactor sau a rotației periodice a cheilor de acces, obținând control asupra resurselor cloud ale victimei. Un exemplu comun este phishing-ul pe conturi de cloud – utilizatorii sunt păcăliți să-și dezvăluie parolele, permițând ulterior atacatorilor să acceseze consolele cloud.

  - Ex. atacuri prin phishing asupra conturilor cloud
  - **Soluție**: Implementarea autentificării multi-factor (MFA)

- **Configurări greșite ale serviciilor cloud**

  Configurarea necorespunzătoare a resurselor cloud (de ex. bucket-uri de stocare setate public din greșeală, permisiuni excesive) reprezintă una dintre cele mai mari amenințări. Conform Cloud Security Alliance, erorile de configurare și control al schimbărilor ocupă locul #1 între riscurile de securitate cloud din 2024.

  O singură setare eronată poate expune întregi baze de date sau sisteme către internet, facilitând accesul neautorizat.

  - Ex. baze de date expuse public (MongoDB, Elasticsearch)
  - **Soluție**: Scanarea regulată și automatizată a configurațiilor
- **Atacuri de tip malware și DDoS asupra aplicațiilor cloud**

  Deși infrastructura cloud robustă atenuează multe atacuri de tip DoS, acestea rămân o amenințare – de exemplu, un atac masiv de tip **Distributed Denial of Service** poate perturba temporar accesul la o aplicație SaaS. În plus, există **malware în mediul cloud**, precum scripturi de cryptojacking (minare de criptomonedă) injectate în mașini virtuale sau imagini container nesigure. Astfel de malware poate consuma resursele și poate duce la costuri crescute sau la compromiterea integrității serviciilor cloud.

  - Ex. atacurile asupra platformelor de streaming și gaming
  - **Soluție**: Utilizarea WAF (Web Application Firewall) și a protecției DdoS

- **Scurgeri de date și conformitate**
  - Ex. breșele de securitate din servicii de stocare cloud (ex. Amazon S3 buckets expuse)
  - **Soluție**: Criptarea datelor la repaus și în tranzit, auditare regulată

- **Riscuri de tip insider și erori umane**

  O parte din incidentele cloud provin din interior – un administrator care configuraază greșit un firewall, un dezvoltator care lasă chei secrete în cod public, sau un angajat rău

intenționat. Complexitatea mediului cloud face ca eroarea umană să fie o cauză majoră a breșelor. De asemenea, un insider cu acces legitim poate exfiltra date dacă politicile de acces nu sunt bine segmentate pe principiul *principiului minimului privilegiu*.

2.3. Măsuri de securitate pentru aplicațiile cloud

Aplicațiile care rulează în cloud trebuie să fie securizate împotriva atacurilor online.

Ex. Dezvoltatorii unei aplicații de editare video bazate pe cloud folosesc teste de securitate ale codului în faza de dezvoltare pentru a identifica și remedierea vulnerabilităților, cum ar fi injecțiile SQL sau problemele de scriptare trans-site (XSS).

- **Autentificare și autorizare** (IAM - Identity and Access Management)

  Securitatea începe cu asigurarea că doar utilizatorii autorizați au acces la resursele cloud.

  - Ex. Controlul accesului pe baza principiului minim "least privilege".
  - Ex. O companie de producție video utilizează un sistem de autentificare multi-factor (MFA) pentru a asigura că accesul la fișierele video de pe platforma de cloud este limitat la personalul autorizat. Acest lucru poate include combinarea unei parole cu o verificare prin SMS sau aplicație de autentificare.
- **Criptarea datelor**

  Criptarea este esențială pentru protecția datelor sensibile, atât în repaus cât și în tranzit.

  - Ex. Utilizarea TLS pentru comunicații și criptare AES-256 pentru datele sensibile.
  - Ex. Un serviciu de streaming criptează toate fișierele video înainte de a le încărca în cloud. Aceste fișiere sunt apoi decriptate dinamic doar în momentul redării de către utilizatorii finali. Acest proces asigură că datele sunt protejate împotriva accesului neautorizat pe parcursul transmiterii și stocării.

- **Monitorizare și detecție de incidente**

Configurațiile nesigure ale serviciilor cloud pot crea vulnerabilități.

  - Ex. Utilizarea AWS GuardDuty, Azure Security Center pentru detecție anomalii
  - Ex. O companie de broadcast folosește un instrument de gestionare a configurațiilor pentru a automatiza și monitoriza setările de securitate ale infrastructurii sale cloud. Acest lucru include limitarea porturilor deschise și utilizarea grupurilor de securitate pentru a controla fluxul de trafic către și dinspre instanțele cloud.

- **Conformitatea și Auditul**
  Respectarea standardelor industriale și a reglementărilor privind protecția datelor este crucială.
  Ex. O platformă de distribuție media se asigură că este conformă cu GDPR pentru a proteja datele personale ale utilizatorilor europeni. Ei efectuează audituri regulate ale securității pentru a verifica și documenta conformitatea cu aceste reglementări.

- **Securitatea la Nivelul Fizic și al Rețelei**
  Protejarea infrastructurii fizice și a rețelei este de asemenea vitală.
  Ex. Centrul de date care găzduiește serverele cloud pentru o companie de producție media are măsuri stricte de securitate fizică, inclusiv camere de supraveghere, garduri de securitate și controale biometrice la intrare.

- **Soluții de backup și recuperare**
  - Ex. Backup automatizat și politici de disaster recovery

Prin implementarea riguroasă a acestor măsuri de securitate, companiile din domeniul media pot minimiza riscurile asociate cu gestionarea și distribuția conținutului în cloud. Aceasta le permite să își protejeze atât propriile active, cât și pe cele ale clienților lor.

---

3. Securitatea aplicațiilor mobile

3.1. Principalele amenințări, riscuri și atacuri asupra aplicațiilor mobile

- **Malware mobil**

  Dispozitivele mobile și aplicațiile lor sunt ținta unei game variate de malware: troieni bancari, spyware, ransomware mobil etc. Malware-ul mobil se răspândește adesea în aplicații legitime. Atacatorii pot crea clone ale unor aplicații populare sau pot insera cod malițios în aplicații reale reambalate, păstrând aparent funcționalitățile originale. De exemplu, aplicații aparent inofensive (jocuri, utilitare) pot conține troieni care, odată instalați, colectează date personale, parole sau chiar preiau controlul dispozitivului (ex. trimit SMS-uri premium, fură contacte etc.). Un raport Kaspersky a raportat peste 33 de milioane de atacuri cu malware și adware pe mobile doar în 2023, evidențiind amploarea fenomenului.

- **Vulnerabilități ale platformelor (sistemelor de operare)**

  Atât Android, cât și iOS au avut vulnerabilități de securitate care, dacă nu sunt remediate prin actualizări, pot fi exploatate de atacatori pentru a obține privilegii sporite pe dispozitiv. Escaladarea privilegiilor prin exploit-uri (cunoscute popular ca jailbreak pe iOS sau root pe Android) le permite atacatorilor să scape de mecanismele de protecție impuse de sistemul de operare și să controleze aproape complet dispozitivul. Un dispozitiv compromis astfel poate duce la accesul la toate aplicațiile și datele sale. În plus, deblocarea neautorizată a bootloader-ului sau utilizarea de versiuni modificate de OS pot dezactiva elemente cheie de securitate oferite de producător.

- **Atacuri asupra codului sursă și inginerie inversă**

  Din perspectiva unui dezvoltator, o amenințare serioasă este posibilitatea ca aplicația mobilă să fie analizată și modificată de persoane neautorizate. **Ingineria inversă** a aplicațiilor (decompilarea codului, analiza logicii interne) poate dezvălui secrete precum chei de API, algoritmi sau vulnerabilități care pot fi apoi exploatate. De asemenea, un atacator poate crea versiuni falsificate ale aplicației (adiționând malware) distribuindu-le ca atare. Un exemplu notabil: în 2019, cercetători de securitate au descoperit aplicații populare reambalate cu adware în magazine neoficiale. Alte atacuri includ **hooking** (interceptarea apelurilor de funcții ale aplicației la runtime, pe dispozitive compromise),

depanarea abuzivă sau rularea aplicației în emulator pentru a o analiza. Fără mecanisme de protecție, aplicațiile mobile pot fi astfel disecate și transformate în vectori de atac.

- Ex. Decompilarea unei aplicații Android folosind APKTool
- **Soluție**: Obfuscarea codului cu ProGuard sau R8

- **Amenințări asupra API-urilor backend ale aplicațiilor mobile**

  Aproape orice aplicație mobilă modernă comunică cu un server sau serviciu cloud prin intermediul API-urilor. Dacă aceste API-uri nu sunt securizate adecvat, atacatorii pot exploata aplicația ca pe **o poartă de acces către server**. De exemplu, în absența verificării stricte a autentificării și autorizării, un atacator poate invoca direct API-urile folosite de aplicație (fără să treacă prin interfața acesteia) pentru a accesa sau modifica date pe server. Un tip de atac frecvent este **BOLA (Broken Object Level Authorization)** – atacatorul modifică identificatori (ID-uri de resurse) în apelurile API (capturate eventual prin inginerie inversă a aplicației) și obține datele altor utilizatori dacă serverul nu verifică riguros drepturile de acces.

- **Atacuri Man-in-the-Middle (MitM) pe conexiuni nesigure**

  Dispozitivele mobile se conectează frecvent la rețele Wi-Fi publice (cafenele, aeroporturi) care pot fi nesecurizate. Un atacator poate intercepta traficul necriptat sau chiar crea un punct de acces Wi-Fi fals pentru a realiza un atac de tip MitM, interceptând comunicațiile victimei. Dacă o aplicație mobilă nu folosește protocoale sigure de criptare a traficului (ex: TLS/HTTPS) sau nu validează corect certificatele, datele transmise (autentificări, mesaje, tranzacții) pot fi citite și chiar modificate de intervenientul aflat în mijloc. Chiar și **aplicațiile mobile care folosesc TLS** pot fi vulnerabile la MitM dacă utilizatorii au dispozitive compromise (ex: cu root/jailbreak) unde un malware instalează un certificat rădăcină fals.

  - Ex. Capturarea traficului nesecurizat cu Wireshark
  - **Soluție**: Folosirea certificate pinning și HTTPS strict

- **Phishing și inginerie socială pe mobil**

  Phishing și inginerie socială pe mobil: Telefoanele mobile facilitează noi vectori de phishing – de la SMS-uri (SMiShing) care îndeamnă utilizatorii să acceseze link-uri malițioase, la notificări push și aplicații de mesagerie folosite pentru distribuirea de escrocherii. Ecranele mici și interfețele simplificate pot face mai dificil pentru utilizatori să verifice legitimitatea unui site sau a unui e-mail, crescând șansele de succes ale phishing-ului. Un exemplu comun este un SMS pretins de la o bancă ce solicită "actualizarea" datelor de cont; dacă utilizatorul atinge link-ul și introduce credențialele, acestea ajung direct la atacatori. Astfel de atacuri pot duce la compromiterea conturilor și furt de identitate.

- **Furtul de date prin permisiuni excesive**
  - Ex. Aplicații care solicită acces la contacte, locație, SMS
  - **Soluție**: Principiul minimului necesar (least privilege)

În ansamblu, mediul mobil se caracterizează printr-o suprafață de atac largă – aplicațiile, sistemul de operare, rețeaua și chiar utilizatorul (prins în capcane de inginerie socială). O aplicație mobilă este considerată de încredere doar dacă îndeplinește un set minim de cerințe de securitate, cum ar fi: să nu conțină funcții ascunse malițioase, să nu colecteze date peste ceea ce e necesar declaratelor funcționalități, să cripteze comunicațiile ce transmit informații personale și să nu prezinte vulnerabilități cunoscute**.**

3.2. Tehnici de securizare a aplicațiilor mobile

- **Practici de securizare a codului sursă**

O aplicație mobilă sigură începe cu scrierea unui cod securizat. Câteva practici esențiale includ:

- **Validarea intrărilor și gestionarea erorilor:** Orice date preluate de la utilizatori sau de la rețea trebuie validate riguros pentru a preveni injecții de cod (SQL, comenzi sistem, XML etc.) sau atacuri XSS (dacă aplicația afișează conținut web). De asemenea, codul ar trebui să gestioneze elegant erorile, fără a divulga informații sensibile prin mesaje de debug sau stack trace în clar.

- **Evitarea vulnerabilităților cunoscute:** Dezvoltatorii trebuie să urmeze ghiduri de secure coding specifice limbajului (de ex. evitarea funcțiilor nesigure de tip `strcpy` în C/C++, folosirea de statement-uri parametrizate pentru interogări SQL, neutralizarea caracterelor speciale în generarea de HTML pentru a preveni XSS etc.). Pentru aplicațiile mobile care rulează parțial cod nativ (C/C++), atenția la erori de memorie (buffer overflow, use-after-free) este crucială.

- **Protejarea secretelor în cod:** Aplicațiile mobile nu ar trebui să conțină chei sau parole hardcodate în codul sursă. Chiar dacă acestea sunt ofuscate, un atacator determinat ce practică ingineria inversă le poate extrage. Cheile de API sau alți secreți trebuie stocați pe server sau furnizați aplicației prin mecanisme securizate (ex: server de configurație securizat, servicii de tip keystore protejate de platformă). Un exemplu negativ celebru este expunerea cheilor API ale serviciului Twitter în aplicații terțe acum câțiva ani, ceea ce a permis abuzul interfețelor de către aplicații neoficiale.

- **Ofuscarea codului și protecția binarelor:** Pentru a îngreuna ingineria inversă și modificarea neautorizată a aplicațiilor, dezvoltatorii pot folosi tehnici de ofuscare (prin care codul compilat devine mai dificil de citit sau decompilat). De asemenea, includerea unor mecanisme de auto-protecție la rulare (RASP – Runtime Application Self-Protection) poate ajuta la detectarea dacă aplicația rulează într-un mediu compromis (ex: pe un device cu root) și la luarea unor măsuri (refuzarea execuției unor funcții sensibile). Scopul este de a preveni atacurile de tip binary attacks (ce includ atât inginerie inversă cât și code tampering, adică modificarea aplicației, de exemplu pentru a elimina restricții sau protecții).

- **Eliminarea informațiilor de debug și a permisiunilor nenecesare:** În faza de producție, aplicația nu trebuie să conțină jurnale de debug detaliate, string-uri sensibile sau alte informații care ar putea ajuta un atacator. De asemenea, aplicația ar trebui să solicite utilizatorului doar permisiunile de care are nevoie efectiv. Orice permisiune suplimentară (acces la contacte, la locație, la microfon etc. care nu este absolut necesară

funcționării) nu face decât să mărească impactul în cazul unui compromis și să ridice probleme de confidențialitate.

Implementarea acestor practici reduce șansele ca aplicația să conțină vulnerabilități exploatabile. Totodată, este recomandată integrarea unor unelte automate de analiză statică a codului (SAST) în procesul de dezvoltare, pentru a detecta din timp eventuale probleme de securitate (injection, buffer overflow, utilizare de API nesigure etc.). De exemplu, rularea unui scaner de securitate precum SonarQube sau Veracode pe codul sursă poate evidenția porțiuni vulnerabile ce trebuie remediate.

- **Gestionarea identității și accesului utilizatorilor**
    - **Autentificare sigură:** Aplicația ar trebui să autentifice utilizatorii folosind metode sigure și, preferabil, standardizate. Evitarea creării de la zero a unui sistem de autentificare este indicată – în schimb se pot folosi servicii OAuth 2.0 sau OpenID Connect oferite de terți (Google, Facebook, Apple) sau un server propriu de identitate bine testat. Dacă aplicația implementează autentificare tradițională cu utilizator și parolă, atunci parolele trebuie să nu fie stocate local niciodată în clar. De regulă, aplicația mobilă trimite credențialele prin conexiune criptată serverului, care validează și emite un token (ex: un token JWT – JSON Web Token) ce va fi folosit ulterior pentru acces. Asigurarea unei autentificări corecte implică și ședințe (sesiuni) bine gestionate – tokenurile de autentificare să aibă expirare rezonabilă, să poată fi revocate de pe server în caz de compromitere și să fie legate de dispozitiv (poate prin includerea unui identificator de dispozitiv în token).

    - **Autorizare strictă:** Autorizarea se referă la controlul acțiunilor pe care un utilizator (autentificat) le poate efectua. Aplicațiile mobile trebuie să implementeze controale de autorizare atât local (ascunderea/dezactivarea funcțiilor la care un anumit rol de utilizator nu ar trebui să aibă acces), cât mai ales pe partea de server/API. Chiar dacă aplicația nu afișează un buton anume pentru un utilizator fără privilegii, un atacator priceput ar putea încerca totuși apelul API corespunzător acelui buton. De aceea, serverul trebuie să verifice drepturile (rolul, permisiunile) asociate fiecărui token sau fiecărei cereri. O eroare frecventă de autorizare este insecure direct object reference, când API-urile permit accesul la obiecte (ex: detalii de

cont cu un anumit ID) fără a verifica dacă acel obiect aparține utilizatorului autenticat curent.

- **Managementul sesiunilor și token-urilor:** În locul sesiunilor pe bază de cookies (specifice aplicațiilor web), aplicațiile mobile folosesc adesea token-uri (precum JWT) în antetele HTTP pentru autentificare. Aceste token-uri trebuie protejate – de exemplu, stocate în mod sigur pe dispozitiv (în Android se poate folosi Keystore-ul securizat, iar pe iOS Keychain-ul) și transmise numai prin conexiuni criptate. De asemenea, este recomandată implementarea mecanismelor de reîmprospătare a token-urilor (refresh tokens) cu expirare scurtă pentru token-urile de acces, astfel încât chiar dacă un token este furat, el să fie valabil doar o perioadă limitată.

- **Autentificare multifactor (MFA):** Pentru acțiuni sensibile (ex: autentificare pe conturi financiare, schimbare de parole, tranzacții bancare) ar trebui oferită sau chiar impusă autentificarea cu doi factori. Pe mobil, un al doilea factor uzual este codul OTP primit prin SMS sau generat de o aplicație precum Google Authenticator. Alte metode includ utilizarea amprentei sau a recunoașterii faciale ca factor suplimentar (de obicei integrat ca metodă de deblocare a aplicației după autentificarea inițială). MFA reduce drastic riscul compromiterii contului în cazul în care parola este aflată de un atacator, deoarece acesta nu va avea și al doilea factor.

- **Politici de parole și securizarea parolelor:** Deși pe mobil parolele sunt dificil de gestionat din cauza tastaturii mici (utilizatorii pot alege parole simple), aplicația ar trebui să impună o politică minimă de complexitate (lungime minimă, combinație de caractere) și să evite practicile nesigure (precum cerința de schimbare frecventă a parolei, care adesea duce la parole mai slabe). Pe server, parolele trebuie stocate hasurate cu un algoritm puternic (ex: bcrypt, Argon2) și sărate (salt) unic pentru fiecare utilizator, pentru a preveni atacurile de tip rainbow table.

Un sistem robust de autentificare și autorizare asigură că numai utilizatorii legitimi accesează aplicația și că fiecare utilizator are acces doar la resursele permise. Multe breșe apar din cauza ignorării acestor aspecte – OWASP include Autentificarea Insecure printre primele riscuri la aplicațiile mobile, evidențiind probleme precum neidentificarea utilizatorului când ar trebui sau menținerea incorectă a identității între sesiuni.

- **Protecția datelor în tranzit și în repaus**
  Aplicațiile mobile manipulează frecvent date sensibile (date personale, token-uri de autentificare, informații financiare etc.), de aceea este esențial ca aceste date să fie protejate atât în tranzit (când sunt transmise între dispozitiv și server), cât și în repaus (stocate pe dispozitiv sau pe server).
    - **Date în tranzit (comunicații de rețea):** Toate comunicațiile dintre aplicația mobilă și serviciile backend trebuie să fie criptate prin protocoale sigure. Standardul de facto este HTTPS peste TLS (Transport Layer Security). Dezvoltatorii trebuie să se asigure că biblioteca HTTP folosită în aplicație verifică validitatea certificatului TLS prezentat de server (pentru a preveni un atacator să se interpună cu un certificat fals). Versiunile vechi și nesigure precum SSLv3 sau TLS 1.0 nu ar trebui utilizate. Ideal, aplicația poate implementa și **certificate pinning**, adică să accepte doar certificatul (sau cheia publică) al/a serverului de încredere predefinit – astfel, chiar dacă dispozitivul are instalată o autoritate certificatoare malițioasă, aplicația refuză conexiuni către server care nu prezintă certificatul așteptat. Protejarea datelor în tranzit previne interceptarea lor în clar prin atacuri de tip MitM. Conform OWASP, comunicarea nesigură (criptare absentă sau incorectă) este una din cele mai mari vulnerabilități la mobile – de aceea, practic toate aplicațiile serioase criptează acum traficul. Un exemplu negativ notoriu a fost, în trecut, transmiterea credențialelor sau a datelor de autentificare în text clar; astăzi astfel de practici ar fi imediat depistate de testele de securitate și corectate

    - **Date în repaus (stocare locală):** Dacă aplicația stochează date pe dispozitiv (în memorie flash, în sistemul de fișiere sau într-o bază de date locală), aceste date trebuie securizate împotriva accesului neautorizat. O regulă de bază **este să nu se stocheze parole sau date extrem de**

sensibile pe dispozitiv, decât dacă este absolut necesar – și chiar și atunci, să fie criptate. Platformele mobile oferă mecanisme de stocare sigură: pe iOS, Keychain-ul permite stocarea criptată a cheilor/tokennurilor protejată hardware (Secure Enclave); pe Android, **Android Keystore** oferă funcționalități similare (cheile stocate nu părăsesc modulul de criptare hardware). Pentru date mai voluminoase (ex: o bază de date SQLite locală), se pot folosi biblioteci de criptare precum SQLCipher (care criptează transparent baza de date cu o parolă/cheie). Un alt aspect este prevenirea **scurgerilor neintenționate de date** – de exemplu, asigurarea că datele sensibile nu ajung în backup-urile necriptate ale dispozitivului, în memoria cache sau în jurnale de sistem. Dezvoltatorii ar trebui să marcheze corect flag-urile de securitate (ex: pe Android, pot marca un fișier ca fiind **Word-Readable = false** pentru a nu putea fi accesat de alte aplicații, sau pot opta să nu includă anumite fișiere în backup-ul cloud al dispozitivului). OWASP evidențiază **stocarea nesigură a datelor** ca risc major la aplicații mobile (ex: salvarea de tokenuri JWT în preferințe nesecurizate sau baze de date fără criptare).

- **Criptografie puternică și corectă:** Atât pentru datele în tranzit cât și pentru cele în repaus, folosirea criptografiei este eficientă doar dacă se aplică algoritmi și implementări solide. Algoritmi slabi sau implementați necorespunzător pot crea un fals sentiment de securitate. De exemplu, utilizarea unui algoritm de criptare casnic sau a unei chei prea scurte poate fi la fel de riscantă ca lipsa oricărei criptări. OWASP a introdus categoria *Insufficient Cryptography* pentru cazurile în care dezvoltatorii încearcă să cripteze, dar o fac greșit, de exemplu, folosesc un mod de operare nesigur (ECB) sau o cheie hardcodată comună pentru toți utilizatorii. Recomandarea este să se folosească algoritmi standard (AES 256 pentru simetric, RSA/ECC pentru asimetric, SHA-256/512 pentru hashing, PBKDF2/bcrypt/Argon2 pentru derivarea cheilor din parole), implementați de biblioteci consacrate ale platformei (CommonCrypto sau CryptoKit pe iOS, javax.crypto pe Android/Java, librării .NET etc.), în loc să se încerce “reinventarea roții” criptografice.

- **Protecția datelor sensibile în memorie:** O provocare suplimentară este că, și dacă datele sunt stocate criptat pe disc, la momentul folosirii ele trebuie decriptate în memorie. Dezvoltatorii ar trebui să șteargă din memorie aceste date cât mai curând posibil după utilizare (de ex., setând manual la zero buffer-ele care au conținut date sensibile) și să evite păstrarea lor în memorie mai mult decât e necesar. De asemenea, pe Android, se poate seta flag-ul FLAG_SECURE pentru ferestrele aplicației care afișează informații sensibile, astfel încât conținutul să nu poată fi capturat în screenshot-uri sau vizualizat în lista de aplicații recente.

În rezumat, criptarea și protecția datelor sunt elemente indispensabile ale securității mobile. Lipsa criptării sau implementarea slabă a acesteia a fost menționată de experți drept una dintre principalele amenințări la adresa aplicațiilor mobile. O aplicație sigură va transmite și stoca datele astfel încât, chiar dacă comunicațiile sunt interceptate sau dispozitivul ajunge pe mâna altcuiva, informațiile să rămână confidențiale.

- **Criptarea datelor – implementare în C/C++, Java, C# și Python**

  Vom ilustra cum se poate implementa criptarea datelor folosind algoritmul simetric **AES (Advanced Encryption Standard)** în diferite limbaje de programare folosite adesea la dezvoltarea aplicațiilor sau componentelor server pentru mobil. Scopul este de a evidenția existența bibliotecilor de criptografie și simplitatea relativă a utilizării lor corecte, comparativ cu tentația periculoasă de a implementa propriile algoritmi.

  C/C++ – exemplu de criptare AES

  În C și C++, cea mai folosită bibliotecă pentru criptografie este **OpenSSL**, care oferă API-uri pentru AES. Exemplul de mai jos prezintă criptarea unui text folosind AES-128 în modul ECB (Electronic Codebook) cu OpenSSL:

```
c
#include <openssl/aes.h>
#include <string.h>
int main() {
    // Cheie de 128 biți (16 octeți)
    unsigned char key[16] = {
        0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,
```

```
        0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF
    };
    // Textul clar (exemplu)
    unsigned char plaintext[] = "MESAJ_SECRET!";
    // Buffer pentru textul criptat (aceeași dimensiune cu
    // plaintext rotunjită la 16 bytes)
    unsigned char ciphertext[16];

    AES_KEY enc_key;
    // Inițializare cheie de criptare AES (128 biți)
    AES_set_encrypt_key(key, 128, &enc_key);
    // Criptarea unui bloc de 16 octeți
    AES_encrypt(plaintext, ciphertext, &enc_key);

    // La acest punct, ciphertext conține textul criptat în
    // format binar.
    // (Într-o utilizare reală, probabil am converti
    //ciphertext la hex sau base64 pentru a fi
    transmis/stocat.)
    return 0;
}
```

Explicații: În exemplu, definim o cheie de 128 de biți și un text clar. Folosim `AES_set_encrypt_key` din OpenSSL pentru a pregăti cheia de criptare, apoi `AES_encrypt` pentru a cripta un bloc de date. Trebuie notat că modul ECB folosit aici este nu recomandat în practică pentru date mai mari, deoarece criptează bloc cu bloc fără amestec, putând lăsa tipare detectabile în textul cifrat. În codul real, ar trebui să utilizăm un mod mai sigur, precum CBC (Cipher Block Chaining) împreună cu un vector de inițializare (IV) aleator, sau GCM (Galois/Counter Mode) care oferă și autentificarea datelor. De asemenea, este important să *umplem (padding)* textul la dimensiunea blocului de 16 bytes pentru AES, dacă nu este deja multiplu – OpenSSL oferă și API-uri de nivel mai înalt (bazate pe EVP) care se ocupă de padding și moduri.

Java – exemplu de criptare AES

Java are incluse în JDK librării de criptografie (Java Cryptography Architecture – **JCA**) care simplifică mult operațiile de criptare. Mai jos, criptăm un șir de caractere folosind AES-128-CBC:

```
java
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.util.Arrays;
import java.nio.charset.StandardCharsets;

public class CryptoExample {
    public static void main(String[] args) throws Exception
{
        // Cheie simetrică de 16 bytes pentru AES-128
        byte[] keyBytes =
"0123456789ABCDEF".getBytes(StandardCharsets.UTF_8);
        SecretKeySpec key = new SecretKeySpec(keyBytes,
"AES");
        // IV (vector de inițializare) de 16 bytes - ales
//aleator în practică
        byte[] ivBytes =
"AAAABBBBCCCCDDDD".getBytes(StandardCharsets.UTF_8);
        IvParameterSpec iv = new IvParameterSpec(ivBytes);

        // Inițializare cifru AES în modul CBC cu padding
//PKCS5
        Cipher cipher =
Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key, iv);

        String plaintext = "Mesaj super secret";
        byte[] ciphertext = cipher.doFinal(
plaintext.getBytes(StandardCharsets.UTF_8) );

        System.out.println("Text clar: " + plaintext);
        System.out.println("Text criptat (hex): " +
bytesToHex(ciphertext));
    }
```

```
    // Utilitar pentru a transforma un array de octeți
//într-un șir hexazecimal
    private static String bytesToHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for(byte b: bytes){
            sb.append(String.format("%02X", b));
        }
        return sb.toString();
    }
}
```

Explicații: Codul creează o cheie secretă de 128 biți dintr-un șir ASCII de 16 caractere și un IV fictiv (în practică IV-ul trebuie generat aleator și de obicei transmis împreună cu textul criptat). Se obține un obiect `Cipher` pentru algoritmul "AES/CBC/PKCS5Padding" – modul CBC cu padding standard PKCS#5 – apoi se inițializează în modul de criptare cu cheia și IV-ul. Metoda `doFinal` aplică automat padding și realizează criptarea datelor, rezultând un tablou de octeți cu textul criptat. La final, imprimăm textul criptat în hexazecimal. Într-o aplicație reală, acest text criptat ar putea fi trimis către server sau stocat local, urmând ca pentru decriptare să se folosească aceeași cheie și IV (sau IV-ul transmis). Avantajul JCA este că dezvoltatorul nu trebuie să se ocupe manual de fiecare pas (padding, operare pe blocuri); totuși, trebuie să aibă grijă la gestionarea cheilor și IV-urilor (să fie securizate și sincronizate între emițător și receptor).

C# (C Sharp) – exemplu de criptare AES

Platforma .NET oferă clasa **Aes** în spațiul de nume `System.Security.Cryptography` pentru a realiza criptarea AES. Un exemplu simplu care folosește AES-CBC cu un IV generat automat:

```
csharp
using System;
using System.Security.Cryptography;
using System.Text;
class CryptoExample {
    static void Main() {
        string plaintext = "Secretul lui Polichinelle";
```

```
        // Inițializare obiect AES cu cheia și IV generate
//implicit
        using (Aes aesAlg = Aes.Create()) {
            aesAlg.KeySize = 128; // putem specifica 128,
//192 sau 256
            aesAlg.GenerateKey();
            aesAlg.GenerateIV();
             ICryptoTransform encryptor =
aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
            byte[] plaintextBytes =
Encoding.UTF8.GetBytes(plaintext);
byte[] cipherBytes =

encryptor.TransformFinalBlock(plaintextBytes, 0,
plaintextBytes.Length);
      Console.WriteLine("Text clar: {0}", plaintext);
      Console.WriteLine("Text criptat (Base64): {0}",
Convert.ToBase64String(cipherBytes));
        }
    }
}
```

Explicații: Codul folosește `Aes.Create()` pentru a obține o instanță a algoritmului AES cu parametri implicit siguri (mod CBC, padding PKCS7). Cheia și IV-ul sunt generate aleator (`GenerateKey()` și `GenerateIV()`), putând alternativ să fie setate manual (`aesAlg.Key = ..., aesAlg.IV = ...`). Apoi se creează un obiect `encryptor` pe baza cheii și IV-ului actual, și se apelează `TransformFinalBlock` pentru a cripta datele (efectuând și padding-ul necesar). Rezultatul, `cipherBytes`, este afișat ca text în baza64. Într-o aplicație reală, am transmite probabil și IV-ul alături de textul criptat, deoarece este necesar la decriptare (fără IV nu putem decoda mesajul criptat). .NET oferă și metode convenabile precum `write()` pe un `CryptoStream` dacă dorim să criptăm fluxuri de date (de exemplu, fișiere mai mari). Ca bună practică, folosim blocul `using` pentru a ne asigura că obiectul Aes este eliminat din memorie imediat după utilizare, ștergând astfel și datele sensibile (cheia) din memorie.

Python – exemplu de criptare AES

În Python, putem folosi biblioteca cryptography (care la rândul ei folosește OpenSSL în spate) sau PyCryptodome. Vom demonstra cu biblioteca cryptography utilizarea unui Fernet (simetric, bazat pe AES128 in CBC cu HMAC pentru integritate):

```
python
from cryptography.fernet import Fernet
# Generare cheie simetrică random
key = Fernet.generate_key()
f = Fernet(key)
plaintext = b"Date ultra-secrete"
ciphertext = f.encrypt(plaintext)
print("Text clar:", plaintext)
print("Text criptat:", ciphertext)
print("Text decriptat:", f.decrypt(ciphertext))
```

Explicații: Fernet este o schemă de criptare simetrică autenticată furnizată de bibliotecă care include tot ce e necesar (genera automat un IV, folosește AES-128-CBC intern și atașează un HMAC-SHA256 pentru a asigura integritatea și autenticitatea mesajului criptat). În exemplu, generăm o cheie random de 32 bytes, criptăm un mesaj și apoi îl decriptăm pentru a verifica. Rezultatul `ciphertext` este un token în formă URL-safe base64 care include atât IV-ul cât și HMAC-ul. Avantajul folosirii unei biblioteci de nivel înalt precum Fernet este că reduce riscul de a uita vreo etapă (de ex. verificarea integrității sau folosirea unui IV aleator). Dacă dorim un control mai detaliat, putem folosi direct modul AES din `cryptography.hazmat`:

```
python
from cryptography.hazmat.primitives.ciphers import Cipher,
algorithms, modes
from cryptography.hazmat.primitives import padding
from os import urandom
key = urandom(32)  # cheie de 256 biți
iv = urandom(16)   # IV de 128 biți
cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
encryptor = cipher.encryptor()
padder = padding.PKCS7(128).padder()  # bloc de 128 biți
pentru AES
```

```
plaintext = b"Exemplu"
padded_data = padder.update(plaintext) + padder.finalize()
ciphertext = encryptor.update(padded_data) +
encryptor.finalize()
```

Acest fragment arată pașii manual pentru AES-CBC în Python: generăm cheie și IV, configurăm un cifru, facem padding manual (bloc de 128 biți) și criptăm. Este mai supus erorilor, de aceea pentru majoritatea aplicațiilor se preferă Fernet sau alte abstracții.

Concluzie la criptare: Indiferent de limbaj, dezvoltatorii au la dispoziție biblioteci solide care implementează algoritmi standard. Este important să le folosească corect (de ex., să nu folosească același IV static mereu, să includă și mecanisme de verificare a autenticității datelor criptate – cum e HMAC sau moduri de operare autentificate ca GCM). Implementarea proprie a algoritmilor sau utilizarea de scheme triviale (ex: "criptarea" datelor prin XOR cu o cheie) trebuie evitate, deoarece adversarii le vor sparge ușor. În schimb, criptarea bine implementată asigură confidențialitatea datelor utilizatorilor și integritatea comunicațiilor, fiind un pilon esențial al securității aplicației mobile.

- **Autentificare și autorizare securizată (OAuth, JWT, MFA)**

  Aplicațiile mobile moderne se bazează adesea pe servicii web și API-uri care necesită autentificare. În locul mecanismelor clasice pe bază de sesiuni web, ecosistemul mobil preferă protocoale token-based și standarde precum OAuth2. Să detaliem aceste concepte și modul de utilizare:

  - **OAuth 2.0:** Este un protocol de autorizare utilizat pe scară largă pentru a permite unei aplicații (denumită client) să acceseze resurse în numele unui utilizator pe un serviciu (provider) fără a expune parola utilizatorului aplicației terțe. În context mobil, OAuth este folosit, de exemplu, când o aplicație vă permite "Login with Facebook/Google/Apple". Aplicația mobilă redirecționează utilizatorul către pagina provider-ului (de ex. Google) unde acesta se autentifică și își dă acordul ca aplicația să acceseze anumite informații (profil basic, email etc.). La final, aplicația primește un **token de acces** (un șir opac) pe care îl va folosi la apelurile către API-ul provider-ului pentru a obține datele aprobate. Pentru implementare, dezvoltatorii pot folosi SDK-urile oficiale ale provider-ilor sau biblioteci

OAuth. Un aspect important de securitate este **redirectionarea și interceptarea token-urilor**: aplicația trebuie să folosească URL-uri de redirecționare securizate (ex: un custom URL scheme pe mobil sau preferabil standardul **PKCE** – Proof Key for Code Exchange – care adaugă un layer de securitate la schimbul de coduri OAuth, prevenind interceptarea codului de autorizare de către o altă aplicație).

- **JWT (JSON Web Token):** Este un format de token autonom, care conține într-o formă compactă (JSON codificat Base64) informații despre identitatea unui utilizator și eventualele sale permisiuni, semnate digital. JWT-urile sunt folosite frecvent atât în cadrul OAuth2/OpenID Connect (de exemplu, ID Token-ul returnat de OpenID Connect este un JWT care conține datele de profil ale utilizatorului), cât și de propriile API-uri ale aplicației pentru autentificare. Un JWT tipic conține trei părți: antet (specifică algoritmul de semnătură), payload (datele – ex: `"sub": "utilizator123", "role": "admin", "exp": 1680000000` ca timestamp de expirare) și semnătura generată cu o cheie secretă sau cheie privată. Avantajul JWT este că serverul poate verifica autenticitatea token-ului **fără** să fie nevoie de o stocare server-side a sesiunilor – verificarea se face calculând semnătura și comparând-o. Într-un flux tipic, după ce utilizatorul se autentifică (fie clasic cu user/parolă, fie via OAuth), serverul emite un JWT pe care aplicația îl va trimite ulterior cu fiecare cerere. Este esențial ca JWT-urile să fie **semnate** cu o **cheie puternică** și să aibă o perioadă de viață limitată (de obicei câteva zeci de minute pentru token-ul de acces). De asemenea, dacă un JWT conferă acces la operațiuni critice, se poate folosi și criptarea conținutului (JWE – JSON Web Encryption), deși adesea semnarea e suficientă, datele nefiind secrete ci doar autoritative. Un exemplu de utilizare: O aplicație de banking mobilă, după logare, primește un JWT care atestă identitatea clientului și drepturile sale; acel JWT este trimis în antetul **Authorization: Bearer <token>** la fiecare apel (ex: obținere sold, efectuare plată), iar serverul validează semnătura JWT-ului și verifică dacă nu a expirat și dacă utilizatorul are dreptul să facă acea acțiune. Mai jos este un mic exemplu Python care arată cum s-ar putea genera și verifica un JWT folosind biblioteca PyJWT:

```
python
import jwt
secret_key = "cheie_secreta_foarte_lunga"  # cheia de
semnare
# Creare token JWT
payload = {"sub": "user123", "role": "admin"}
token = jwt.encode(payload, secret_key, algorithm="HS256")
print("JWT emis:", token)

# Verificare/decodare token JWT
try:
    decoded = jwt.decode(token, secret_key,
algorithms=["HS256"])
    print("JWT valid. Conținut:", decoded)
except jwt.InvalidTokenError:
    print("Token invalid sau modificat!")
```

- Acest cod produce un token JWT semnat HMAC-SHA256 și apoi îl decodifică, validând semnătura cu aceeași cheie. În practică, cheia secretă ar trebui să fie stocată numai pe server (nu în aplicație) – aplicația mobilă doar transmite JWT-ul primit de la server.

- **MFA (Autentificare multi-factor):** Am menționat anterior, merită reiterat ca parte din strategia de autentificare sigură. MFA adaugă un nivel suplimentar: pe lângă ceva ce utilizatorul **știe** (parola), se cere ceva ce **are** (telefonul pentru a primi un cod sau a genera un cod) sau ce **este** (biometrie – amprentă/faceID). Pe mobil, integrarea MFA poate fi făcută nativ (ex: folosirea API-urilor de biometrie ale sistemului de operare ca al doilea factor, sau trimiterea unui OTP prin SMS către telefon – deși SMS-ul are propriile vulnerabilități, este totuși mai bun decât nimic). O metodă modernă este utilizarea aplicațiilor de autentificare (Google Authenticator, Microsoft Authenticator) care generează coduri TOTP sincronizate temporal cu serverul. Implementarea TOTP pe server se face conform

standardului HOTP/TOTP (algoritmi și parametri specificați de IETF), iar pe client utilizatorul doar introduce codul afișat. În ansamblu, oferirea opțiunii de MFA reduce foarte mult riscul ca un atacator să preia controlul asupra contului doar cu o parolă compromisă.

- **Sesiuni de utilizator și logout:** În aplicațiile mobile, de obicei sesiunile persistă (utilizatorul rămâne logat mult timp pentru a nu-i cere parola frecvent). Totuși, trebuie prevăzut un mecanism de logout (manual sau forțat de server). Dacă se suspectează un compromis (ex: serverul detectează activitate suspectă sau utilizatorul își resetează parola), serverul poate invalida token-urile active (păstrând la server o listă de token-uri revocate sau un timestamp de ultim logout și incluzând în JWT data emiterii – astfel știe să refuze token-urile emise înainte de ultimul logout). De asemenea, aplicația ar trebui să permită utilizatorului să își șteargă datele locale și token-urile la logout, astfel încât, dacă altcineva obține acces fizic la telefon, să nu găsească sesiunea deschisă.

În concluzie, autentificarea și autorizarea într-o aplicație mobilă modernă gravitează în jurul token-urilor și standardelor web (OAuth2, JWT), punând accent pe experiența utilizatorului (login federat, one-tap login) dar în același timp pe securitate (semnături, MFA). Adoptarea acestor standarde este recomandată și de organisme internaționale – de exemplu, NIST recomandă autentificarea cu doi factori pentru acces la date sensibile și folosirea de token-uri în locul sesiunilor tradiționale pentru aplicații distribuite, reducând suprafața de atac a furtului de sesiune.

- **Securitatea API-urilor mobile și metode de protecție împotriva atacurilor**

Majoritatea funcționalităților aplicațiilor mobile sunt susținute de servicii web (API-uri RESTful sau GraphQL, de obicei) care rulează pe servere în cloud. Securitatea acestor API-uri backend este o parte integrantă a securității aplicației mobile, deoarece o aplicație mobilă oricât de bine scrisă nu poate oferi securitate dacă serverul cu care comunică este vulnerabil. Iată aspectele cheie și contramăsurile privind securitatea API-urilor pentru mobile:

- **Autentificare și autorizare pe API-uri:** Așa cum s-a discutat, fiecare apel de la aplicația mobilă către server ar trebui autentificat (de ex. printr-

un token în antet). Serverul trebuie să verifice acel token la fiecare cerere. **Niciun endpoint sensibil nu trebuie lăsat neprotejat.** Un caz celebru de eșec este breșa suferită de operatorul telecom Optus în 2022, când un API public nu solicita deloc autentificare, permițând oricui a descoperit URL-ul să acceseze date confidențiale**.** *Lecție:* orice API care furnizează date legate de utilizatori sau operațiuni critice trebuie să necesite un token valid și să efectueze controale de autorizare (dacă utilizatorul X cere datele utilizatorului Y, să fie refuzat). În practică, se pot folosi gateway-uri API care să centralizeze autentificarea/autoritatea (ex: API Gateway AWS, Apigee etc.), aplicând politici unificate.

- **Validarea intrărilor (input sanitization) la nivel de API:** Chiar dacă aplicația mobilă impune anumite validări (ex: câmpul "username" acceptă doar litere și cifre, sau lungimea maximă 50), serverul nu trebuie să se bazeze pe acestea. Un atacator poate fabrica manual cereri HTTP către API (folosind un tool precum Postman sau interceptând traficul de la aplicație și modificându-l) și poate trimite input malițios. Așadar, serverul trebuie să valideze toate datele primite: parametri de URL, corpul JSON al cererii, anteturile. Astfel se previn atacurile de tip SQL Injection, NoSQL/ORM Injection, Command Injection sau chiar Deserializare Insecure (dacă API-ul primește obiecte serializate). OWASP API Security Top 10 evidențiază Injection și BOLA ca prime riscuri, deci contramăsurile sunt: filtrarea și validarea input-urilor și verificarea riguroasă a autentificării/autorizării pe fiecare obiect accesat.

- **Protecții împotriva atacurilor de forță brută și automatizate:** Un API expus poate fi ținta unor atacuri automate: scripturi care încearcă combinații utilizator/parolă (password spraying), sau încearcă mii de token-uri sperând ca unul să fie valid, sau pur și simplu bombardează API-ul pentru a-i satura resursele (DoS). Pentru a atenua aceste riscuri, se implementează rate limiting (limitarea numărului de cereri pe unitatea de timp de la o anumită adresă IP sau pe un anumit token de acces). De exemplu, serverul poate permite maxim 5 încercări de autentificare per minut de la același IP, sau maxim 100 de cereri la un endpoint într-un

minut. O altă măsură este introducerea de delays progresive la autentificări eșuate (ex: după 3 parole greșite pentru un cont, serverul așteaptă 30 secunde înainte de a permite următoarea încercare). Pentru protecția contra bot-ilor automați, uneori se folosesc și CAPTCHA la anumite acțiuni, deși pe mobil acestea afectează experiența, astfel că alternative precum soluțiile invisible reCAPTCHA sau verificări comportamentale sunt preferate.

- **Comunicare securizată și integritate:** Pe lângă criptarea TLS deja discutată, o tehnică suplimentară folosită de unele aplicații pentru a se asigura că cererile provin de la aplicația lor oficială și nu de la un client fals este utilizarea unui API secret sau a unui HMAC al cererii. De exemplu, unele servicii includ în documentația API o metodă de semnare HMAC: atât clientul cât și serverul cunosc un secret (cheie) și clientul trimite în antet o semnătură HMAC calculată peste corpul cererii sau peste anumiți parametri și timestamp. Serverul recalculează semnătura și, dacă nu corespunde, respinge cererea. Acest mecanism împiedică tampering-ul (modificarea) cererilor în tranzit și poate descuraja utilizarea neautorizată a API-ului (atacatorul ar trebui să fure cheia secretă din aplicație mai întâi, ceea ce ofuscarea codului poate îngreuna). Totuși, ținând cont că secretul trebuie să existe în aplicație la un moment dat, aceste scheme nu sunt infailibile – un atacator sofisticat îl poate extrage prin inginerie inversă, deci ele se folosesc mai mult ca layer suplimentar de securitate, nu ca unicul mijloc.

- **Protecția contra atacurilor pe nivelul de transport și rețea:** Aici intră TLS pinning (discutat), dar și utilizarea de protocoale suplimentare cum ar fi DNS over HTTPS sau includerea serviciilor anti-MiTM. De asemenea, pentru a evita ca un atacator să reprogrameze aplicația să trimită datele spre un alt server, se pot implementa controale pe serverul autentic – de exemplu, verificarea la nivel de aplicație server că User-Agent-ul sau pattern-urile traficului corespund aplicației legitime (deși acestea pot fi imitate). O abordare modernă este folosirea de servicii de atestare a aplicației (precum SafetyNet/Play Integrity pe Android, sau App Attest pe iOS) care permit serverului să știe dacă cererea vine de la o aplicație

nemodificată rulând pe un dispozitiv necompromis. Astfel de soluții (incluzând și cele comerciale, ex. soluții de mobile app attestation ca Approov) generează un atestat pe care aplicația îl trimite la server; serverul îl validează cu furnizorul (Google/Apple sau altul) și capătă încredere sporită în cerere. Aceasta este însă o zonă avansată, de obicei utilizată de aplicații cu risc înalt (ex: aplicații bancare) deoarece implementarea și costul pot fi ridicate.

- **Jurnaliere și monitorizare pe API:** Serverele ar trebui să înregistreze accesul la API-uri (cel puțin evenimente importante ca autentificări, cereri eronate, rate-limit triggers, răspunsuri 4xx/5xx), iar aceste loguri să fie monitorizate. O creștere bruscă a numărului de erori de autentificare sau a cererilor la un anumit endpoint poate indica un atac în curs (ex: un atac de forță brută sau un fuzzing după vulnerabilități). Monitorizarea în timp real, eventual cu sisteme de detectare a intruziunilor (IDS) pentru API (există concepte de Web Application Firewall – WAF – adaptate pentru API-uri) poate ajuta la blocarea automată a unor adrese IP sau sesiuni suspecte.

Exemplu practic de atac și apărare: Să considerăm cazul T-Mobile 2023. Atacatorii au descoperit un API al unui operator de telefonie (T-Mobile) care, deși protejat de autentificare, a fost accesat cu un token valid furat de la un alt serviciu. Timp de ~6 săptămâni, atacatorii au extras date personale la ~37 de milioane de clienți, deoarece API-ul respectiv furniza informații precum nume, telefon, adresă email fără verificări suplimentare. În acest caz, s-a speculat că limitele de rată au fost fie absente, fie prea permisive, permițând un volum atât de mare de cereri fără alarmare. Ca răspuns, compania a trebuit să investească masiv în îmbunătățiri de securitate. **Morala:** Chiar și cu autentificare, un API care oferă date sensibile trebuie monitorizat și limitat corespunzător; de asemenea, datele sensibile ar trebui minimizate (dacă nu e necesar ca un API să returneze 5 câmpuri personale, mai bine să nu o facă). Un alt exemplu este cel menționat cu Optus 2022, unde greșeala a fost flagrantă – lipsa autentificării pe un API public a dus la expunerea datelor personale a ~10 milioane clienți. Acel incident a evidențiat încă o dată importanța măsurilor fundamentale: autentificare pe toate endpoint-urile și izolarea sistemelor interne de cele publice (un API public ar trebui, dacă e posibil, să nu dea direct acces la baza de date cu clienți reali, ci la o zonă segmentată cu un subset de date).

Pe scurt, securitatea API-urilor mobile este strâns legată de securitatea aplicației: o aplicație sigură trebuie să comunice cu un server sigur. Practicile de dezvoltare securizată a serviciilor web (valabil și pentru aplicații web) se aplică și în cazul aplicațiilor mobile: autentificare robustă, autorizare la nivel de obiect, validare intrări, criptare, rate limiting, monitorizare. Organizațiile ar trebui să urmeze ghiduri precum OWASP API Security Top 10 pentru a se asigura că API-urile lor rezistă la încercările de atac.

- **Exemple practice și cod sursă**

Pentru a consolida conceptele discutate, vom examina câteva exemple practice, inclusiv fragmente de cod, care ilustrează implementarea unor mecanisme de securitate și studii de caz ale unor atacuri reale sau simulate, împreună cu metodele de prevenție.

- **Implementări de criptare și autentificare în diverse limbaje**

Criptare simetrică (AES) în diferite limbaje: În secțiunea mai sus am prezentat fragmente de cod pentru C/C++, Java, C# și Python care demonstrează folosirea bibliotecilor de criptografie pentru a cifra date. Observăm un aspect comun: în loc să scriem manual algoritmul AES, folosim API-uri de nivel înalt oferite de biblioteci consacrate. Acest lucru reduce riscul erorilor. De exemplu, în Java și C# padding-ul este gestionat automat de bibliotecă (PKCS5/7), dezvoltatorul trebuind doar să specifice modul și eventual IV-ul. În C/C++, OpenSSL necesită puțin mai multă atenție (ex: dezvoltatorul trebuie să se asigure că datele sunt aliniate la bloc și să aleagă modul de operare), însă tot pune la dispoziție rutine standard. Morala practică: indiferent de limbaj, există suport pentru criptografie puternică – utilizarea acestor mecanisme standard este o bună practică esențială (și adesea și o cerință legală, cum ar fi conformitatea cu standardele de protecție a datelor).

Autentificare cu JWT – exemplu practic în Python: Fragmentul de cod de mai sus a arătat cum se poate genera un JWT și verifica semnătura acestuia folosind biblioteca PyJWT. Într-o aplicație reală Python (să zicem partea de server a unui API scris în Flask sau FastAPI), am folosi asta în felul următor: la autentificare (endpoint /login) dacă userul și parola sunt corecte, serverul ar face **jwt.encode({"sub": user_id, "role": rol}**, secret) și ar trimite token-ul către aplicație. Apoi, la fiecare alt endpoint securizat, am avea un decorator/middleware care interceptează cererea, citește header-ul Authorization Bearer, face **jwt.decode(token, secret)** și dacă validează, permite execuția mai departe, altfel returnează 401 Unauthorized. Astfel se implementează rapid un sistem de sesiune stateless. Trebuie menționat că, dacă se folosește un JWT semnat cu o cheie simetrică (HMAC) partajată între servicii, toate componentele care

verifică tokenul trebuie să păstreze secretul. Alternativ, se poate folosi un JWT cu algoritm asimetric (RS256) – caz în care serverul de autentificare semnează cu cheia privată, iar alte servicii verifică cu cheia publică (eliminând nevoia de a avea secretul peste tot). Această arhitectură este comună în sistemele enterprise cu microservicii.

Hasharea parolelor – exemplu în C#: Deși nu am inclus cod mai sus, este util de subliniat cum se face corect stocarea unei parole. În C#, clasa RNGCryptoServiceProvider poate genera un salt aleator, apoi Rfc2898DeriveBytes (PBKDF2) poate fi folosit pentru a genera un hash securizat al parolei + salt. Exemplu concept:

```
csharp

using(var rng = new RNGCryptoServiceProvider()) {

    byte[] salt = new byte[16];

    rng.GetBytes(salt);

    var pbkdf2 = new Rfc2898DeriveBytes(parolaIntrodusa,
salt, 10000);

    byte[] hash = pbkdf2.GetBytes(32); // 32 bytes hash

    // stocăm in DB: salt-ul și hash-ul (de ex. concatenat
//sau în coloane separate)

}
```

Apoi la verificare, se recalculează hash-ul cu saltul stocat și se compară. Această practică asigură că, dacă baza de date este compromisă, atacatorul nu va putea obține ușor parolele utilizatorilor (dat fiind că fiecare e hashu-ită cu salt propriu și un număr mare de iterații care fac bruteforce-ul foarte lent). Deși acest subiect ține mai mult de partea de server, are relevanță directă pentru securitatea aplicațiilor mobile deoarece utilizatorii adesea folosesc aceeași parolă în mai multe locuri – deci compromiterea parolei într-o aplicație slab protejată poate duce la acces neautorizat și în aplicația mobilă securizată, dacă parolele coincid.

Conectarea și autorizarea prin OAuth – exemplu conceptual: Un exemplu practic este integrarea "Login with Google" într-o aplicație mobilă. Folosind biblioteca oficială Google Sign-In, aplicația mobilă redirecționează utilizatorul către ecranul de autentificare Google, primește apoi un token de autentificare Google. Acest token este de obicei un ID Token (JWT) care conține e-mailul utilizatorului confirmat de Google. Aplicația poate trimite acel token serverului său; serverul validează semnătura JWT-ului cu cheia publică Google și extrage email-ul. Dacă acel email corespunde unui cont din aplicație, consideră utilizatorul autentificat. Acest flux implică mai puține responsabilități de securitate pentru aplicație (nu mai stochează parole direct), dar introduce nevoia de a valida corect tokenul Google (inclusiv verificarea audienței – că tokenul e emis pentru clientul nostru – și a timpului de expirare). Este o bună practică ca aplicațiile mobile să folosească astfel de servicii OAuth/OIDC bine testate, mai ales când vorbim de autentificare de masă, tocmai pentru a evita erori de implementare locală.

- **Exemple concrete de protecție a API-urilor mobile**

Să analizăm un scenariu și contra-măsurile aferente:Scenariu:

Avem o aplicație de social media care are un API endpoint **/api/v1/updateProfile** ce primește un obiect JSON cu datele de profil ale utilizatorului și le actualizează în baza de date. În mod normal, aplicația mobilă apelează acest endpoint după ce utilizatorul editează profilul în interfață. În spate, se apelează cu metoda PUT sau POST, includând un token de autentificare.

Posibil atac: Un atacator încearcă să folosească acest endpoint pentru a modifica profilul altui utilizator. Metodă: capturează cererea API legitime a sa (de exemplu cu un proxy gen OWASP ZAP) – vede că este de forma **PUT /api/v1/updateProfile** cu corp **{"user_id":123, "name":"New Name", ...}.** Își dă seama că dacă schimbă user_id în altă valoare, poate poate actualiza profilul altcuiva. Încearcă cu **user_id=124** și trimite cererea cu tokenul său. Dacă serverul nu verifică autorizarea la nivel de obiect, atunci profilul utilizatorului 124 va fi suprascris cu noile date – un exemplu de atac BOLA (Broken Object Level Authorization).

Măsuri de protecție aplicate: Pentru a preveni așa ceva, pe server, implementarea endpoint-ului trebuie să ignore orice **user_id** venit în corp (sau chiar să nu îl ceară deloc) și să folosească ID-ul din tokenul de autentificare al cererii. Astfel, updateProfile va aplica modificările doar pentru utilizatorul autentificat asociat token-ului prezent. În plus,

ar putea exista o verificare: dacă un utilizator încearcă să-și atribuie privilegii mai mari (ex: trimite un parametru "role":"admin" într-un endpoint de update profil), serverul să ignore câmpurile nepermise sau să refuze cererea. Asemenea verificări de whitelist al câmpurilor modificabile pot preveni escaladarea privilegiilor prin API.

Alte măsuri: Endpoint-ul respectiv ar trebui să aibă și un rate limit, pentru a preveni un script care încearcă să modifice masiv (sau să ghicească ID-uri). De asemenea, dacă datele de profil includ câmpuri ce pot conține HTML (de ex. o biografie cu formatare), ar trebui igienizate pentru a evita stocarea unui script care să fie apoi servit altor utilizatori (atac XSS stocat via API).

Exemplu de utilizare a unui WAF/API Gateway: Să zicem că aplicăm un API Gateway în fața serverului de social media. Acesta poate detecta anumite tipare – de exemplu, dacă cineva accesează secvențial 100 de ID-uri de profil diferite (încercare de extragere masivă a datelor de profil ale utilizatorilor), gateway-ul poate bloca temporar acel client. Totodată, gateway-ul poate asigura transformarea HTTP -> HTTPS, astfel încât serverul intern să nu fie accesibil pe HTTP direct. În practică, soluții ca AWS API Gateway sau Azure API Management oferă astfel de capabilități out-of-the-box, reducând volumul de muncă pe partea de implementare securizată a serverului.

Protejarea cheilor de API publice în aplicație: Uneori aplicația mobilă folosește API-uri de la terți (ex: o hartă Google Maps) ce necesită o cheie API. Această cheie, dacă este expusă, ar putea fi folosită abuziv de alții (ex: consumând cota plătită de API). Pentru a preveni asta, platformele oferă opțiuni de restricționare – de exemplu, cheile Google Maps pot fi restricționate să funcționeze doar din aplicația noastră (identificată prin semnătura digitală a pachetului) și numai pentru anumite endpoint-uri. Astfel, dacă un atacator extrage cheia din aplicație și încearcă s-o folosească într-o altă aplicație sau de pe serverul lui, cererile vor fi refuzate.

Bune practici în cod: nu lăsați chei de API în cod în clar; folosiți, dacă e posibil, servicii proxy securizate pentru a vă ascunde cheile secrete. De exemplu, dacă aplicația mobilă trebuie să comunice cu un serviciu terț, puteți alege ca aplicația mobilă să trimită cererea la propriul server, iar acesta, având cheile necesare, să interogheze serviciul terț și să returneze rezultatul mobilului. Aceasta crește ușor latența, dar sporește securitatea (cheia nu mai ajunge în aplicație, deci nu poate fi furată de acolo).

- **Studii de caz relevante privind atacuri și metode de protecție**

Pentru a înțelege mai bine cum se traduc conceptele teoretice în situații reale, vom discuta succint câteva studii de caz:

Studiu de caz 1: Breșa de la Optus (2022) – API neautentificat. Optus, un furnizor mare de telecom din Australia, a suferit o breșă masivă când un atacator a descoperit un API expus public care nu cerea autentificare și returna datele clienților (inclusiv nume, adrese, numere de documente). Exploatarea a fost trivială și a dus la compromiterea datelor a ~10 milioane persoane**. Cauza:** eroare de configurare/proiectare (lipsa autentificării și expunerea directă a datelor sensibile). **Contramăsuri:** După incident, compania a segmentat mai strict API-urile interne vs publice, a implementat autentificare uniformă și a revizuit ce date sunt expuse prin API. De asemenea, acest incident a declanșat discuții despre reglementări mai dure (guvernul a cerut schimbări, evidențiind importanța conformității cu standarde de securitate).

Studiu de caz 2: Atac asupra aplicației Starbucks (2014) – stocare nesigură. O versiune veche a aplicației mobile Starbucks stoca numele de utilizator și parola în clar, în text, în fișierele de log ale aplicației (și token-ul de autentificare) pe dispozitiv. Un atacator care avea acces la telefon (sau malware-ul) putea extrage aceste credențiale cu ușurință. **Soluția:** Starbucks a remediat rapid, eliminând logarea datelor sensibile și securizând stocarea token-ului (mutând-ul într-o zonă criptată a dispozitivului). Acest caz a evidențiat necesitatea revizuirii codului pentru a detecta asemenea scurgeri neintenționate și adoptarea principiului nu loga nimic sensibil.

Studiu de caz 3: WhatsApp – criptarea end-to-end (2016). Deși nu e un "atac" propriu-zis, merită menționat momentul în care WhatsApp a integrat criptare end-to-end pentru toate mesajele. Anterior, mesajele trimise puteau fi interceptate (dacă un atacator realiza MitM sau serverul era compromis). După implementare, fiecare mesaj este criptat cu cheia publică a destinatarului, putând fi decriptat doar de acesta (cheile private fiind stocate doar pe dispozitive). Rezultat: chiar dacă serverele WhatsApp ar fi compromise, atacatorul nu ar putea citi mesajele. Acesta este un exemplu pozitiv de proiectare orientată pe securitate și confidențialitate. O lecție aici este utilizarea criptografiei end-to-end când natura aplicației o permite, pentru a minimiza încrederea necesară în infrastructura intermediară. Din perspectiva implementării, acest lucru a implicat gestionarea complexă a cheilor pentru sute de milioane de utilizatori (protocolul Signal

utilizat de WhatsApp se ocupă de distribuția cheilor și rotația lor), arătând că securitatea avansată este fezabilă la scară mare când devine prioritate.

Studiu de caz 4: Vulnerabilitatea API Instagram (2017). O vulnerabilitate în API-ul Instagram a permis atacatorilor să obțină adrese de email și numere de telefon ale unor conturi de utilizatori celebri. Exploatarea a implicat o lipsă de limitare la interogarea unui endpoint care returna aceste date pe baza user-id-ului. Atacatorii au putut face scraping pe scară largă deoarece nu exista un throttle adecvat și nici detectare de pattern anormal. După incident, Facebook (proprietarul Instagram) a închis acel endpoint și a introdus mecanisme anti-scraping mai stricte. Morala este similară cu cele de mai sus: orice API care poate furniza date personale trebuie protejat nu doar de acces neautorizat, dar și de extrageri masive neobișnuite chiar de către utilizatori autorizați (ex: un cont compromis care începe să acceseze datele tuturor prietenilor).

Fiecare dintre aceste exemple ne arată atât greșeli comune (stocare în clar, lipsa autentificării, validări insuficiente, lipsa limitelor) cât și modul în care ar fi putut fi prevenite prin aplicarea principiilor discutate în capitolele anterioare. Ideal, organizațiile ar trebui să învețe proactiv din astfel de incidente (ale altora) și să își auditeze propriile aplicații mobile și cloud pentru a identifica dacă suferă de probleme similare.

- **Bune practici pentru securitatea aplicațiilor mobile**

În această secțiune finală de conținut vom sintetiza bunele practici și ghidurile recunoscute în industrie care ar trebui urmate pentru a asigura securitatea aplicațiilor mobile, precum și metode de testare și monitorizare.

- **Ghiduri și standarde internaționale relevante**

  - **OWASP (Open Web Application Security Project):** oferă resurse valoroase pentru securitatea mobile. Două dintre cele mai importante sunt OWASP Mobile Top 10 – o listă a primelor 10 riscuri de securitate pentru aplicații mobile (ultima versiune majoră fiind din 2016, cu o actualizare în lucru pentru 2024), și **MASVS (Mobile Application Security Verification Standard)** – un standard detaliat ce conține cerințe de securitate împărțite pe categorii (arhitectură, gestionarea datelor, criptografie, autentificare, rețea, cod etc.) pentru aplicațiile mobile. Recomandarea este ca dezvoltatorii să consulte MASVS ca checklist în

timpul dezvoltării, iar evaluatorii de securitate să folosească OWASP Mobile Security Testing Guide (MSTG), care descrie metodologii de testare pentru fiecare categorie de vulnerabilitate. Urmând aceste ghiduri, se pot evita majoritatea greșelilor comune. De exemplu, MASVS impune ca nicio informație sensibilă să nu fie logată, ca toate conexiunile să folosească TLS corect, ca aplicația să valideze starea de jailbreak/root dacă e cazul și multe altele – practic sumând tot ce am discutat anterior, într-o formă organizată și verificabilă.

- **NIST (National Institute of Standards and Technology):** NIST, prin publicațiile speciale, oferă linii directoare pentru securitate. În domeniul mobil, NIST SP 800-163 (Vetting the Security of Mobile Applications) este un ghid ce propune un proces de evaluare a securității aplicațiilor mobile, de la planificare, dezvoltare, testare, până la distribuție și mentenanță. De asemenea, NIST are și documente pentru securitatea dispozitivelor mobile (ex: SP 800-124 despre gestionarea securității dispozitivelor). Un alt reper este NIST Cybersecurity Framework, care nu este specific mobile, dar ale cărui principii (Identify, Protect, Detect, Respond, Recover) se aplică și aici. Un aspect scos în evidență de NIST și aplicabil direct aplicațiilor mobile este necesitatea criptării datelor sensibile și a utilizării MFA pentru autentificare – aliniat cu tot ce am discutat.

- **Regulamentul GDPR (General Data Protection Regulation):** Deși este o reglementare privind protecția datelor personale în UE, GDPR are implicații asupra modului în care aplicațiile mobile (care procesează date personale ale utilizatorilor) trebuie dezvoltate. Principiul de "privacy by design" impune ca dezvoltatorii să ia în calcul protejarea datelor încă din faza de concepție a aplicației. Practic, pentru securitate, asta înseamnă: stocarea minimului necesar de date personale (principiul minimizării), protejarea acestora prin criptare atât la tranzit cât și la repaus, obținerea consimțământului clar pentru prelucrări, și implementarea de mecanisme de ștergere la cerere a datelor. Un exemplu: dacă o aplicație mobilă colectează poziția GPS a utilizatorului, conform GDPR trebuie să

informeze clar și să ceară acordul, și totodată să securizeze transmisia acestor date către server și stocarea lor. În caz de breșă de securitate ce expune date personale, GDPR impune notificarea autorităților și a utilizatorilor în termen scurt, și pot exista amenzi severe. Așadar, conformarea cu GDPR nu este doar o chestiune legală, ci devine și un motor pentru a implementa măsuri solide de securitate în aplicații, astfel încât probabilitatea unei breșe să fie minimă.

- **ISO 27001 și securitatea dezvoltării software:** ISO/IEC 27001 este un standard internațional pentru sistemele de management al securității informației (ISMS). Deși nu se referă specific la aplicații mobile, o organizație certificată 27001 va avea politici și controale pentru dezvoltarea securizată a software-ului. De exemplu, controlul A.14.2 din ISO 27001 se ocupă de securitatea în procesele de dezvoltare și suport, incluzând efectuarea de evaluări de vulnerabilitate, protejarea codului sursă, principiul celor patru ochi (cod review), managementul configurațiilor și al librăriilor terțe etc. Urmând ISO 27001, companiile se asigură că au un cadru managerial care obligă la respectarea bunelor practici tehnice. În context mobil, asta ar putea însemna: să existe ghiduri interne de programare sigură pentru mobile, scanări periodice de securitate (SAST/DAST) programate, gestionarea accesului la codul sursă (ex: codul sursă al aplicației mobile este protejat și nu oricine îl poate modifica) și planuri de răspuns la incidente (vom detalia la 5.3).

Pe scurt, respectarea acestor standarde și ghiduri asigură un nivel baseline de securitate. OWASP oferă direcția tehnică granulară, NIST și ISO oferă cadrul general și bune practici de proces, iar GDPR (și alte reglementări, ex: PCI-DSS dacă aplicația procesează plăți) impun cerințe specifice axate pe protecția datelor. O aplicație mobilă dezvoltată în conformitate cu aceste repere va fi mult mai rezilientă și va inspira încredere utilizatorilor și partenerilor.

- **Tehnici de testare a securității aplicațiilor mobile**

Testarea este crucială pentru a verifica eficacitatea măsurilor de securitate implementate. O abordare cuprinzătoare implică mai multe tipuri de teste:

Analiză statică de securitate (SAST): presupune examinarea codului sursă sau a codului bytecode al aplicației fără a-l executa, pentru a descoperi vulnerabilități. Există instrumente automate capabile să identifice probleme ca injecții, buffer overflow, utilizare de API nesigur (ex: detectează dacă developerul folosește o funcție de criptare depreciată) și chiar pattern-uri specifice mobile (ex: API de Android interzis). De exemplu, MobSF (Mobile Security Framework) poate analiza un APK Android sau un IPA iOS pentru a evidenția permisiuni excesive, endpoint-uri hardcodate, string-uri sensibile în cod etc. Integrarea SAST în CI/CD (integrare continuă) face ca noile vulnerabilități introduse de programatori să fie prinse imediat, înainte de a ajunge într-o versiune live. Cum menționam, OWASP recomandă includerea SAST în procesul de dezvoltare.

Analiză dinamică (DAST) și testare de penetrare: Aici testăm aplicația în execuție pentru a vedea cum se comportă și dacă putem exploata ceva. Pentru partea de client (aplicația pe telefon), se pot folosi emulatori sau dispozitive reale cu instrumente de debugging (adb, Xcode Instruments) pentru a monitoriza ce face aplicația: transmite date în clar? stochează fișiere sensibile? scrie în log? Un test dinamic include și interceptarea traficului cu un proxy (Burp Suite, OWASP ZAP) – dacă reușim (sau nu, în caz de pinning) să interceptăm comunicația, putem încerca să modificăm parametri și să vedem reacția serverului (astfel identificăm eventual BOLA, injection etc. pe API). Testarea dinamică a partei de server (API) este la fel ca un test de penetrare web: scanăm endpoint-urile (eventual folosind și un scanner automat de vulnerabilități), verificăm autentificarea, încercăm inputuri malițioase, testăm limitările. Uneori, se organizează testări de tip "mobile bug bounty", invitând cercetători externi să descopere probleme contra recompensă – un mod eficient de a afla vectori neașteptați de atac.

Analiza de cod binar și revers engineering (pentru aplicația mobilă): Un tester va încerca să decompileze aplicația (folosind apktool, Jadx pentru Android, sau instrumente pentru iOS dacă are pachetul, deși iOS e mai dificil fără jailbreak). Scopul este de a vedea dacă găsește chei ascunse, stringuri sensibile, logică de securitate slabă. De exemplu, s-ar uita dacă există cod în aplicație de tipul **if(isRooted()) { /* allow anyway**

***/ }** – ceea ce ar indica toleranță la rulează pe device compromis. Sau ar verifica integritatea implementării criptografiei – dacă observă că aplicația folosește o parolă fixă pentru criptarea unei baze de date, ar putea extrage acea parolă și apoi citi baza de date. Acest tip de test asigură perspectiva unui atacator care a compromis mediul mobil și încearcă să profite de cunoștințe extrase din aplicație.

Fuzzing și testare de reziliență: Fuzzing-ul presupune trimiterea unui volum mare de date semi-aleatorii către aplicație sau API, pentru a vedea dacă se blochează sau se comportă ciudat – indicând potențiale vulnerabilități de tip buffer overflow sau gestionare incorectă a erorilor. Pe mobil, fuzzing-ul se aplică mai ales la nivel de API sau eventual la interfața UI (de ex. Monkey tool pe Android care generează mii de interacțiuni random să vadă dacă aplicația crapă – nu e direct securitate, dar asigură stabilitate, ceea ce contează în disponibilitate). Pentru partea de API, fuzzing-ul parametrilor (ex: trimitere de JSON-uri malformate, foarte mari, structuri recursive) poate dezvălui erori de implementare pe server.

Testare specifică platformei: Aici intră lucruri ca verificarea implementării App Transport Security pe iOS (setările din Info.plist, că aplicația nu permite conexiuni ne-secure), verificarea setărilor de proguard/ofuscare pe Android (dacă APK-ul are cod ofuscat sau nu), verificarea că aplicația respectă Android Secure Coding Standard (de ex. nu folosește moduri de criptare neaprobate). Există și servicii de scanare automatizată oferite de Google Play (Play Protect) sau Apple App Store – ele nu sunt transparente, dar prind uneori probleme (de ex. dacă o aplicație folosește o librărie cunoscută ca vulnerabilă/banată, e respinsă). Ca atare, e de dorit ca înainte de a trimite la store, dezvoltatorii să ruleze propriile teste pentru a nu fi surprinși.

Audituri terțe și certificări: Pentru aplicații din domenii critice (bancar, medical), uneori se solicită audit independent. O firmă specializată face un penetration test complet și oferă un raport. În unele cazuri, există certificări formale – de exemplu, în sectorul plăților mobile, standardul PCI Mobile Payment Acceptance Security Guidelines ar putea fi relevant dacă acceptă carduri direct; sau scheme de certificare guvernamentale (Common Criteria) pentru anumite aplicații mobile de securitate.

Un plan de testare ar trebui să combine metodele de mai sus. OWASP MSTG oferă capitole întregi pentru testarea datelor, rețelei, criptografiei, autentificării, interfației, codului etc., cu exemple de proceduri. E recomandat ca echipele să folosească MSTG ca listă de verificare în

timpul evaluărilor interne. De asemenea, includerea testelor de securitate în pipeline (DevSecOps) ajută la descoperirea timpurie a problemelor.

- **Monitorizarea și răspunsul la incidente de securitate**

  Chiar și cu toate măsurile preventive și testele, este esențial ca organizațiile să aibă un plan pentru situația în care totuși apare un incident de securitate.

  Monitorizarea aplicațiilor mobile în producție: În contextul mobil, monitorizarea se împarte între monitorizarea la nivel de aplicație (client) și la nivel de server:

  - **Pe partea de client,** posibilitățile sunt mai limitate, dar unele aplicații includ mecanisme de telemetrie care pot indica un atac. De exemplu, o aplicație poate detecta și raporta către server dacă rulează pe un dispozitiv jailbreak/root (caz în care riscul de atac local crește). Sau poate trimite evenimente dacă constată repetate erori de conexiune suspecte (ce ar putea indica un MitM în care cineva prezintă certificate invalide). Există soluții de tip Mobile Threat Defense (MTD) care se pot integra în aplicații sau la nivel de device management și care monitorizează continuu integritatea device-ului și a aplicațiilor (ex: dacă este instalat un malware cunoscut pe telefon, MTD poate alerta sau bloca aplicațiile corporative). Astfel de soluții (ex: Microsoft Intune, Zimperium zIPS) sunt folosite mai ales în mediu enterprise, unde angajații folosesc aplicații mobile ale companiei.

  - **Pe partea de server/API,** monitorizarea este similară cu cea a oricărei aplicații web: logarea accesului (cine, ce a accesat, de unde), logarea erorilor de aplicație, alertele la evenimente cheie. Se pot folosi sisteme de tip SIEM (Security Information and Event Management) care să agrege loguri din multiple surse și să detecteze anomalii. De exemplu, dacă de pe un anumit cont de utilizator mobil se fac în 5 minute 5000 de cereri (ceea ce un om cu telefonul nu ar putea face), SIEM poate declanșa o alertă că acel token poate a fost folosit scriptic – semn de posibilă automatizare malițioasă sau cont compromis. Monitorizarea ar trebui să acopere și performanța – uneori un spike de CPU sau memorie pe server poate indica un atac DoS sau o exploatare (ex: cineva încearcă un exploit și provoacă leak de memorie).

Plan de răspuns la incidente (Incident Response - IR): Nicio apărare nu e perfectă, așa că trebuie pregătit un Incident Response Playbook, conform principiului "Assume Breach". Conform NIST, etapele sunt *Detectare – Analiză – Izolare – Remediere – Recuperare.* Aplicat la aplicații mobile:

- *Detectare:* Cineva (sistem automat sau utilizator) raportează un incident – de exemplu, se observă date anormale în sistem sau apare pe internet o listă de date care par a proveni din aplicație (semn că s-a produs o exfiltrare). Sau poate chiar echipa descoperă un bug sever de securitate în codul unei versiuni lansate.

- *Analiză și triere:* Se confirmă incidentul, se evaluează impactul (câți utilizatori, ce fel de date afectate, modul de atac folosit dacă se cunoaște).

- *Izolare/Atenuare:* Aici intră acțiuni precum: dezactivarea temporară a unor funcționalități sau servere ca să oprești sângerarea. De exemplu, dacă un endpoint API e vulnerabil, poate îl oprești până rezolvi sau pui un WAF în față cu o regulă care blochează exploit-ul. Dacă un certificat a fost compromis, revoci acel certificat (toate conexiunile implicate vor eșua până la actualizare). Dacă aplicația mobilă însăși are o vulnerabilitate critică (ex: versiunea 2.3 permite login fără parolă din greșeală), poți forța utilizatorii să actualizeze – de exemplu, serverul poate refuza cererile de la versiunea respectivă cu un mesaj de upgrade required.

- *Remediere:* Odată controlat incidentul imediat, echipa de dezvoltare lucrează la un patch. În context mobil, asta înseamnă adesea lansarea unei noi versiuni a aplicației (pentru vulnerabilități client) sau a serverului (pentru vulnerabilități API). Aici lucrurile pot fi complicate de procesul de publicare în store și de faptul că nu toți utilizatorii fac update imediat. În cazuri grave, se pot trimite push notifications sau emailuri tuturor utilizatorilor pentru a-i instiga să facă update urgent, evidențiind importanța. Dacă datele au fost compromise, remedierea implică și suport pentru utilizatori – ex: resetarea parolelor, invalidarea token-urilor (forțarea relogării tuturor) etc., în funcție de natura incidentului.

- *Comunicare incident:* Parte a răspunsului este și notificarea părților interesate. Conform GDPR (dacă e cazul), trebuie anunțați utilizatorii afectați și autoritatea în max 72h de la constatarea breșei. Chiar și dacă nu e o cerință legală, transparența e considerată o bună practică – companiile tind acum să informeze utilizatorii "ce s-a întâmplat, ce date au fost expuse, ce măsuri să ia utilizatorii și ce face compania pentru a preveni pe viitor". Un exemplu pozitiv e când o companie oferă și asistență, de ex: monitorizare gratuită a creditului dacă date financiare au fost expuse.

- *Învățare și îmbunătățire:* După incident, echipa ar trebui să facă o analiză post-mortem și să îmbunătățească procesele. Dacă a fost o eroare de programare, poate adaugă un caz de test în suitele automatizate să nu se mai repete. Dacă a fost o breșă de API, poate implementează scanări de securitate mai frecvente sau training suplimentar pentru dezvoltatori.

Un aspect special în mobil: dacă un serviciu terț pe care îl folosește aplicația e compromis, trebuie reacționat. Exemplu: Să zicem că aplicația mobilă se bazează pe un serviciu de push notification oferit de o terță parte, și acel serviciu e atacat în așa fel încât cineva trimite notificări malițioase userilor (ex: phish). Chiar dacă nu e vina directă a aplicației, trebuie să reacționeze – poate dezactivează temporar push-urile sau informează utilizatorii să ignore anumite mesaje. Acest lucru este menționat și de experți: planul de răspuns trebuie să acopere incidentele la parteneri externi, nu doar cele interne

In concluzie, a avea un plan de monitorizare și răspuns este la fel de important ca a preveni atacurile inițial. Măsurile de securitate reduc riscul, dar nu-l pot elimina 100%. O organizație matură din punct de vedere al securității aplicațiilor va ști nu doar să construiască aplicații sigure, dar și să detecteze și limiteze rapid eventualele breșe, minimizând daunele pentru utilizatori și pentru sine.

• Concluzii și recomandări

Rezumat al principalelor lecții:

Securitatea în mediile cloud și mobile reprezintă o provocare complexă, dar gestionabilă prin aplicarea consecventă a principiilor fundamentale de securitate cibernetică adaptate particularităților acestor medii. Am început prin a defini securitatea cloud ca un set de

măsuri pentru protecția utilizatorilor, datelor și infrastructurii cloud, respectiv securitatea mobilă ca ansamblul de măsuri ce întăresc apărarea dispozitivului și aplicațiilor mobile contra diverselor riscuri. Am evidențiat importanța acestor domenii prin adoptarea masivă – miliarde de utilizatori depind de ele – și am arătat că, în lipsa securității, consecințele pot fi grave (breșe de date, fraude financiare, atingerea vieții private).

Analizând amenințările, am identificat paralelisme dar și diferențe între cloud și mobil: dacă pentru cloud, top riscuri sunt erori de configurare, gestionare de identități și API-uri nesigure, în zona mobilă malware-ul, phishing-ul, vulnerabilitățile de platformă și atacurile asupra aplicațiilor și API-urilor sunt predominante.. Atacatorii profită atât de slăbiciuni tehnice (un server neprotejat, o criptare absentă), cât și de erori umane (furnizarea de credențiale prin phishing, neatenția la permisiuni etc.).

Securitatea aplicațiilor mobile, punctul central al discuției, presupune o abordare end-to-end: scriere de cod sigur (validări stricte, fără secrete hardcodate, protecții anti-tamper), gestionarea identității cu mecanisme moderne (OAuth, JWT, MFA), criptarea datelor atât în comunicații cât și stocare locală (folosind algoritmi puternici și biblioteci standard), precum și securizarea interfețelor API prin autentificare, autorizare și monitorizare atentă. Am ilustrat cum se implementează practic criptarea în diverse limbaje, demonstrând că instrumentele necesare sunt la îndemâna oricărui programator – deci nu există scuze tehnice pentru a nu cripta datele sensibile. De asemenea, am subliniat importanța autentificării robuste: integrarea MFA, folosirea token-urilor de acces cu viață scurtă, invalidarea lor la nevoie, toate aceste măsuri îngreunează enorm munca unui potențial intrus.

Exemplele practice ne-au arătat atât ce nu trebuie făcut (chei API expuse, endpoints fără autentificare, parole în clar) cât și ce trebuie făcut (limite de rată, pinning de certificate, atestare a aplicației, patch management rapid). Am studiat cazuri ca breșa Optus – un exemplu clar de eșec al principiilor de bază, sau cazul T-Mobile unde, deși exista un sistem de securitate, monitorizarea deficitară a permis extragerea datelor pe ascuns. Fiecare incident a venit cu lecții: securitate la nivel de design (nu adăuga securitatea ulterior), principiul celui mai slab verigă (o singură portiță neasigurată compromite întregul lanț), nevoia de audit continuu și actualizare.

Recomandări cheie: În lumina celor discutate, iată câteva recomandări concrete pentru dezvoltatorii și arhitecții de aplicații mobile și cloud:

1. Adoptarea unui framework de securitate încă din faza de proiectare: Înainte de a scrie codul, identificați cerințele de securitate. Folosiți standarde ca OWASP MASVS ca listă de cerințe. Practica threat modeling (modelarea amenințărilor) poate fi foarte utilă: imaginați-vă cum ar încerca cineva să atace aplicația și asigurați contramăsuri pentru acele scenarii.

2. Practica "secure-by-default" și minimizarea expunerii: Configurați serviciile cloud cu minimele permisiuni necesare (zero trust approach), activați criptarea peste tot unde este posibil (de exemplu, multe servicii cloud permit criptarea automată a stocării – activați-o). Pentru aplicații mobile, dacă o funcționalitate nu este esențială, mai bine eliminați-o decât să lase o poartă (ex: dacă nu aveți nevoie ca aplicația să ruleze pe device rooted, puteți bloca asta). Defaulturile de securitate ale platformelor sunt de obicei bune – nu le relaxați fără motiv.

3. Managementul dependințelor și actualizărilor: Țineți la zi nu doar sistemele de operare de pe servere, ci și bibliotecile terțe folosite în aplicația mobilă. Vulnerabilități cunoscute în biblioteci populare apar frecvent (ex: o vulnerabilitate într-un SDK de publicitate sau analytics folosit în aplicație poate expune milioane de utilizatori). Folosiți instrumente de SCA (Software Composition Analysis) pentru a identifica componentele și versiunile vulnerabile și actualizați-le prompt.

4. **Testare continuă și audit independent:** Introduceți verificări de securitate la fiecare versiune. Nu lansați actualizări majore de funcționalitate fără un nou set de teste de penetrare. Invitați periodic echipe externe să auditeze aplicația – ele pot descoperi probleme la care dezvoltatorii, obișnuiți cu propria lor creație, devin orbi. Explorați opțiunea de bug bounty, dacă aveți o comunitate mare de utilizatori – aceasta poate oferi un feedback valoros și proactiv.

5. Monitorizare activă post-lansare: Implementați logare suficient de detaliată pe server (dar atenție la confidențialitate – nu logați date personale inutil). Folosiți alerte automate pentru evenimente critice (multiple login eșuate, accesări masive, erori de sistem). Pregătiți un plan de reacție și exersați-l (simulați un incident pentru a vedea cum răspunde echipa). După cum subliniază experții, speră la ce e mai bun, dar pregătește-te pentru ce e mai rău– astfel, dacă se întâmplă ceva, veți acționa rapid și coerent, reducând impactul.

6. Educația echipei și a utilizatorilor: Nu în ultimul rând, investiți în educație. Echipa de dezvoltare ar trebui să fie la curent cu ultimele vulnerabilități și tehnici de apărare. Standardele OWASP, NIST etc., nu trebuie doar citite, ci și înțelese și integrate în cultură. În paralel, educați-vă utilizatorii prin aplicație – de exemplu, includeți mesaje care să îi sfătuiască să nu-și jailbreak-uiască telefonul, sau să-și actualizeze aplicația, sau explicați de ce cereți anumite permisiuni ("Pentru siguranța contului dvs, vă rugăm să activați verificarea în doi pași"). O bază de utilizatori conștienți va completa eforturile tehnice ale dezvoltatorilor.

Direcții viitoare în securitatea aplicațiilor mobile: Privind înainte, securitatea mobilă se va intersecta tot mai mult cu noi tehnologii și paradigme. IoT și 5G vor crește suprafața de atac – multe dispozitive vor fi controlate de pe mobil, deci compromiterea telefonului poate însemna compromiterea casei inteligente, mașinii etc. Aplicațiile mobile vor trebui să se conformeze și mai mult conceptului de **Zero Trust**, presupunând mereu că rețeaua e nesigură și verificând continuu identitatea și integritatea dispozitivului. **Inteligența artificială** va juca un dublu rol: pe de o parte atacatorii o pot folosi pentru a găsi vulnerabilități sau genera phishing adaptativ, pe de altă parte apărătorii o pot utiliza pentru a detecta anomalii de comportament (AI/ML în detectarea fraudelor în aplicații bancare, de exemplu, deja se folosește). **Criptografia post-cuantică** este un subiect emergent: deși pare de domeniul viitorului îndepărtat, serviciile în cloud și aplicațiile care le vor folosi se vor pregăti treptat pentru algoritmi rezistenți la calcul cuantic, asigurând că datele criptate astăzi rămân sigure și mâine. Pentru dezvoltatorii mobile, asta ar putea însemna adoptarea unor noi biblioteci criptografice în următorul deceniu.

În rezumat, viitorul securității aplicațiilor mobile va necesita adaptabilitate și vigilența continuă. Atacurile evoluează, la fel și contramăsurile. Arhitecții și dezvoltatorii trebuie să mențină securitatea ca pe un obiectiv de prim rang, nu ca pe o after-thought, și să fie dispuși să îmbrățișeze noi practici și tehnologii pe măsură ce acestea devin disponibile. Numai astfel vor putea proteja eficient utilizatorii și datele acestora în ecosistemul mobil, care a devenit poate cea mai importantă platformă computațională a erei noastre.