Miskolci Egyetem

Gépészmérnöki és informatikai Kar

Általános Informatikai Intézeti Tanszék

Biztonsági rések a pajzson

Szakdolgozat

Készítette:

Név: Kovács Gergely

Neptunkód: MKTJS0

Szak: Mérnökinformatikus BSc, levelező

Korszerű web technológiák szakirány

Tartalom

1. Bevezetés

Az informatikai szempontú biztonság fontossága egyre növekszik napjainkban. Az orosz-ukrán háború árnyékában felfokozódtak a hackertevékenységek, egyre több (nem hadviselő) ország esetében merül fel a gyanú, hogy valamilyen visszaélés, befolyásolás történt (esetleg az országgyűlési képviselők választásába, vagy az élet egyéb területein, a közösségi média felhasználásával).

Ha ez nem lenne elég ok komolyabban foglalkozni a témával, ott vannak a békés országokban elkövetett online térben, telefonon elkövetett visszaélések. Az ilyen bűnözői tevékenység által okozott károk egyre nőnek, az FBI (amerikai szövetségi nyomozóiroda) 2000-ben alakult Internet Crime Compliant Center nevű osztálya minden évben elkészíti az Internet Crime Reportot [3], amely a kiberbűncselekmények típusairól és az általuk okozott károk mértékéről ad információt. A 2025-ös jelentés április 23-án készült, eszerint 2024-ben a kiberbűnözők 16,6 milliárd dollárt loptak el, ez 33%-os növekedés az előző évhez képest. A tendencia a korábbi évekhez képest is folyamatosan növekvő, a Mastercard [4] becslése szerint 2028-ra a károk elérik a 14000 milliárd dollárt.

Adatokról van szó, amit hajlamosak vagyunk komolytalan problémaként kezelni. Csak legyintünk, hogy „persze mit kezdenének a mi adatainkkal?!”. „Mitől lenne olyan értékes az én születési dátumom?!” Ezeket a kérdéseket érdemes újra átgondolni. Bár az előbb említett, online térben elkövetett csalások kárértéke bár óriási szám, távolinak tűnhet, érdemes elgondolkodni Marsi Tamás NBSZ NKI Igazgatóhelyettes szavain: „a kiberbűnözők végcélja a pénzünk megszerzése. Az adataink, amiket ellopnak tőlünk, eszközök ebben a folyamatban, hiszen előbb, vagy utóbb ezeket is monetizálni fogják” [4 6. dia].

Az adataink tehát tényleg értékesek. Amellett, hogy tudatosítani kell a felhasználókban is, hogy mennyire fontos vigyázni az adatokra, más szempontból tudatosítani kell ezt a fejlesztőkben is, mivel ők mások adataiért is felelősek. Az ő esetükben ugyanis olyan adatokról beszélünk, amelyeket szoftverek, alkalmazások tárolnak. Az adat tehát

érték, ha jobban belegondolunk valójában a bankkártyaszámunk is csak egy adat, amit nem szeretnénk, ha mások is megtudnának.

Ezzel el is érkeztünk a biztonságos programozás felelősségének kérdéséhez. A cél az, hogy a biztonsági kérdések és témakörök tisztázásával ráirányítani a figyelmet a biztonság kérdésének fontosságára, amely fontos a felhasználónak és fontos az adatokat valamilyen formában kezelőknek.

Ha ugyanis egy szoftvert készítünk, számolnunk kell azzal, hogy értékes adatokat tárolunk, ezekért a programozó bizonyos mértékben felelős. A biztonságos fejlesztés érdekében tudatosítani kell, hol lehet sérülékeny a programunk, tudatosítani kell, hogy milyen sebezhető pontjai vannak a készülő alkalmazásnak, és hogyan lehet megerősíteni ezeket a pontokat. Érdemes olyan módon gondolkodni, hogy nem elég beérni azzal, hogy a kód működjön, biztonságossá is kell tenni azt. Mindig indokolt feltenni a kérdést: Hogyan támadnának itt meg? Hogyan lehet ezt rosszindulatúan kihasználni? Hasznosabb ilyen szemlélettel készíteni egy szoftvert, ezzel hosszú távon pénzt lehet megtakarítani, mert egy utólagos javítás összköltsége biztosan több lesz, arról nem is beszélve, mekkora presztízsveszteség az, ha tőlünk lopják el mások értékes adatait.

A most következő biztonsági áttekintés tehát fontos lehet a felhasználók szemszögéből, akik így tudatosíthatják magukban az online világ veszélyeit. Fontos továbbá az adatokat valamilyen szempontból felhasználók és kezelők szemszögéből is, akik alkalmazásokat készítenek. Az tehát a cél, hogy mindenkiben megfogalmazódjon a szándék, hogy biztonságos alkalmazást akarok használni vagy készíteni.

A következőkben ebben segít a problémák nagyrészt elméleti áttekintése, a biztonsági rések példákon keresztül történő bemutatása, majd egy kis webalkalmazás segítségével biztonsági megoldások szemléltetése, közben pedig a *sosem lehetünk biztosak abban, hogy hibamentes az alkalmazásunk* elv szellemében a webalkalmazás kritikája is megtörténik, további javítási javaslatokkal.

2. A biztonság alapjai [28][30][31][32]

Ahhoz, hogy biztonságos alkalmazásokat tudjunk készíteni, fontos, hogy figyeljünk azok biztonságára már a tervezés során is. Ahhoz pedig, hogy biztonságosan tudjunk tervezni, tisztázni kell az online biztonság alapfogalmait, illetve azokat a feltételeket, amelyeknek meg kell felelnie a készülő alkalmazásunknak. A biztonság alapjai mellett pedig tisztázni kell, a lehetséges veszély- és hibaforrásokat is.

A szoftverek biztonsága meginoghat magán a terméken (*Product*). Fontos tudatosítani, hogy a gyakori hibák általában rontják a biztonságot is. Az alkatrészek és a felhasznált platformok biztonsága is veszélyeket rejt magában, illetve, ha a szoftver és a gazdarendszer konfigurációja nem biztonságos. Ezek a kérdések előkerülnek később, illetve lesz szó a biztonsági irányelvek kidolgozásáról, és az ezeknek való megfelelés fontosságáról.

A másik biztonságot aláásó tényező: az ember (*Person*). Az emberi hibák, a helytelen feltételezések a szoftver működésével kapcsolatban, a helytelen szoftverhasználat, a nem szándékos bizonytalan viselkedés a szoftverrel kapcsolatban (a felhasználók tudják legjobban elrontani a szoftvert), mind veszélyt jelentenek, amelyre fel kell készülni. És még nem is beszéltünk arról, hogy a támadásokat is emberek követik el, a rosszindulatú, másokat megkárosító tevékenységük eredményeként. Ennek legfőbb tanulsága a programozók számára az, hogy az emberben (legyen az felhasználó, vagy támadó) nem lehet megbízni, minden esetben feltételezni kell a rosszindulatú, vagy véletlen kihasználást.

A harmadik veszélyforrás a fejlesztés folyamata, nevezhetjük *Process*nek. Itt veszély az, ha a biztonság kérdése nem integrálódik a teljes fejlesztési folyamatba. Ide sorolható még, ha a szoftverek és a telepített eszközök karbantartása nem megfelelő, és ha a fejlesztési eszközök nem támogatják a biztonságot. Ezekről a kérdésekről is lesz még szó.

A fentieket nevezhetjük a szoftverbiztonság 3P-jének [32, 1.6. videó], az angol elnevezésük alapján, így a legkönnyebben megjegyezhetők.

A veszélyforrások után pedig fontos tisztázni a biztonság alapfogalmait és alapelveit, azért, hogy mindenki ugyanazt értse a fogalmak alatt, és egyúttal ráirányítják a figyelmet a problémákra is. Erről szól a következő fejezet.

2.1. Alapfogalmak, alapelvek

Az információ biztonsága (*Information Security*) az adatok és az információk védelmét jelenti a jogosulatlan hozzáféréstől, módosítástól, megzavarástól, nyilvánosságra hozataltól vagy megsemmisítéstől. [30 4. old] Ettől elkülönül az információs rendszerek biztonsága (*Information Systems Security*), amely a számítógépek, szerverek, hálózati eszközök, vagyis a kritikus adatokat tároló és feldolgozó rendszerek védelmét jelenti. Ha valamilyen szoftvert készítünk, ez nem a mi feladatunk, nem is témája a dolgozatnak, de tudatosítandó veszélyforrás.

Ahhoz, hogy biztonságos programokat tudjunk készíteni, meg kell felelni a *Confidentiality*, vagyis titoktartás elvének is, amely azt jelenti, hogy az általunk tárolt információkhoz csak az arra jogosult személyek férhessenek hozzá. Ez a jogosulatlan hozzáférés elleni védelmet és a nyilvánosságra hozatal elleni védelmet jelenti, hogy magánjellegű vagy érzékeny információk ne legyenek elérhetők jogosulatlanok számára. Védeni kell a személyes adatokat, az üzleti előny fenntartása érdekében és a meglévő szabályozásoknak való megfelelés céljából. A védelem biztosításához a titkosítás (*encryption***),** a hozzáférés-vezérlés (*Access Control*), az adatmaszkolás (*Data Masking*), a fizikai biztonsági intézkedések (*Physical Security Measures*), és a képzés és tudatosság (*Training and Awareness*) eszközeit használhatjuk fel.

Másik alapelv az *Integrity***,** vagyis az integritás, amely biztosítja, hogy az adatok pontosak és változatlanok maradjanak, jogosulatlanul ne módosíthassa azokat senki. Ezért ellenőrizni kell az adatok pontosságát és megbízhatóságát a teljes életciklusuk során. A cél az adatok pontosságának biztosítása, a bizalom fenntartása, és a rendszer működőképességének biztosítása. A programozás során ezt hash-eléssel (*hashing),* a digitális aláírások (*Digital Signatures)*, az ellenőrző összegek (*checksums***),** a hozzáférés-vezérlés (*Access Control)* használatával tudjuk biztosítani és rendszeres ellenőrzések (*Regular Audits)* segítségével fenntartani.

Az *Availability,* vagyis az elérhetőség biztosítása azt jelenti, hogy az arra jogosult személyeknek az információk és erőforrások mindig hozzáférhetők és működőképesek legyenek. Fontos ez az üzletmenet folytonosságának biztosítása miatt, az ügyfelek bizalmának fenntartása miatt, és a szervezet jó hírének fenntartása miatt. Ezt redundancia, vagyis többszörözés, a rendszer kritikus összetevőinek megkettőzése segítségével tudjuk elérni (*Redundancy*). Létezik *Server Redundancy* (több szerver használata), *Data Redundancy* (adatok több helyen tárolása, biztonsági mentések), *Network Redundancy* (hálózati utak többszörözése) és *Power Redundancy* (tartalék áramforrások használata), a biztonságos programozás szempontjából a Data Redundancy érdekes ezek közül.

A 3 korábbi biztonsági alapelv, a CIA háromszög mellé csatlakozott még újabban egy szintén nagyon fontos elv, a *Non-Repudiation,* amely letagadhatatlanságot jelent, és biztosítja, hogy egy bizonyos eseményt vagy cselekményt az érintett felek ne tagadhassanak le, bizonyítékot nyújt a digitális tranzakciókra, ezzel elszámoltathatóságot biztosít a digitális folyamatokban. Ennek biztosítására használhatunk digitális aláírást (*Digital Signatures*), melyről később még lesz szó.

A biztonság további összetevői, a 3A a biztonságban (*Triple A's of Security)* az *Authentication* vagyis Hitelesítés, az *Authorization* vagyis Engedélyezés, és az *Accounting* vagyis könyvelés**,** naplózás is segít az adatok védelmében.

Az *Authentication* a felhasználó személyazonosságának ellenőrzését jelenti, amely biztosítja, hogy a digitális folyamatokban jelen lévő személyek valóban azok legyenek, akiknek vallják magukat. Ez lehet valami, amit a felhasználó ismer (tudásalapú faktor, például jelszó), valami, ami a felhasználóé (birtoklásalapú faktor – fizikai elem, például pendrive, belépőkártya), valami, ami a felhasználó maga (biológiai faktor – ujjlenyomat, arckép), valami, amit csinál (akció faktor – pl. mozgásminta), valami, ahol a felhasználó van (lokációs faktor – földrajzi helymeghatározás). Előbbiek közül több alkalmazása a hitelesítés során a két- vagy többfaktoros hitelesítés (*Multi-Factor Authentication System – MFA)*. Ennek használatával megelőzhetjük a jogosulatlan hozzáférést, védhetjük a magánéletet és a felhasználói adatokat, és biztosíthatjuk, hogy a rendszerhez csak az arra jogosult felhasználók férhessenek hozzá.

Az Authenticationt követő művelet az *Authorization*, amely meghatározza, hogy a hitelesített felhasználó milyen jellegű információkhoz férhet hozzá, és engedélyek és

jogosultságok segítségével biztosítja, hogy csak azokhoz férjen hozzá, amikhez hozzá kell férnie. Ezzel védjük az érzékeny adatokat, megőrizzük a rendszer integritását.

Ha előbbi két művelet megtörtént, még mindig fontos az *Accounting,* mely szerint mindent naplózunk és rögzítünk, beleértve a felhasználói aktivitást és erőforrás-felhasználást is. Időrendben kell rögzíteni az összes felhasználói tevékenységet, ez segíthet majd, ha szükség lesz a változások, az illetéktelen hozzáférés, vagy hibák forrásáig és időpontjáig való visszakövetésre. A részletes eseménynaplók segíthetnek a biztonsági szakértőknek is az események kivizsgálásában és megértésében, ha már megtörtént valamilyen incidens, és a következő megelőzésében, hogy ne történhessen meg újra. Az erőforrás-felhasználás naplózása segíthet hatékonyabb alkalmazások készítésében, a fejlesztésben is. A felhasznált technológiák ehhez: *Syslog Servers* (naplók a hálózati eszközökről, rendszerekről), *Network Analysis Tools* (hálózati forgalom rögzítése és elemzése), *Security Information and Event Management (SIEM) Systems* (valós idejű elemzés biztonsági riasztásokról), amelyek bár alacsonyabb szintű eszközöknek számítanak a programozáshoz képest, érdemes ismerni ezeket és felhasználni a szoftver üzemeltetése során.

Fontos biztonsági szemléletet hordoz a *Zero Trust* modell, amely alapértelmezetten mindenkit és mindent megbízhatatlannak tekint, beleértve a leendő felhasználókat is. Alapelve: „soha ne bízz, mindig ellenőrizz” (“never trust; always verify” - *Jacqueline Pitter: Zero Trust Software Architecture*) [28 123. oldal]. Jogosnak tekinthető ez az elv, mivel a felhasználóink véletlenül elronthatják a hibásan implementált folyamatokat. Például ha egy étterem webes alkalmazásának online rendelési felületén új funkcióként utólag bevezetünk rendszerhasználati díjat, és azt egyszerűen egy ugyanolyan (törölhető) új tételként jelenítjük meg a kosárban, mint mondjuk egy pizza, akkor ha a felhasználó meggondolja magát, és mindent kitöröl a kosárból (beleértve a rendszerhasználati díjat is) és egy másik feltéttel rendelkező pizzát választ helyette, nem biztos, hogy a rendszerhasználati díj újra hozzáadódik a kosárhoz (ha mégis, ugyanúgy kitörölhető, a rendelhető tételektől eltérő módon, egy új megoldásra van szükség). Ha pedig a felhasználó rosszindulatú, és valamilyen haszonszerzés céljából „piszkálja” az alkalmazásunkat, azonnal ki fogja használni a biztonsági réseket, tehát teljesen jogosan úgy kell kezelnünk a felhasználót, és minden eszközt is amit a készülő alkalmazás felhasznál, hogy az megbízhatatlan, potenciális veszélyforrást jelent (például harmadik féltől származó komponensek, amelyek a Node.js esetén az npm

csomagjainak függőségei, azoknak a függőségei és a függőségek függőségei, amelyekkel sérülékenységeket örökölhetünk). Különösen veszélyes az, ha egy belső, nagyobb jogosultsággal rendelkező felhasználónk akar valamilyen kárt okozni, de ez is létező jelenség, ezzel is számolnunk kell. Nem szabad megbízni a felhasználókban. A Zero Trust a hálózaton belül minden eszköz, felhasználó és tranzakció ellenőrzését követeli, függetlenül annak forrásától.

Egyértelmű, hogy az előbbi alapelvek egy alkalmazás készítése során mind megvalósítandó problémát jelentenek. Természetesen befolyásolja ezt, hogy milyen alkalmazást készítünk, és milyen adatokat tárolunk. Meg kell tehát fogalmaznunk már a tervezés fázisában a szoftverkövetelmények mellett biztonsági követelményeket is, a felhasználói és platformkövetelmények mellett. Meg kell még felelni üzleti követelményeknek, és a vonatkozó szabványoknak és jogi előírásoknak is, attól függően milyen alkalmazás készül. A követelmények tisztázása és az azoknak való megfelelés szintén felvet biztonsági kérdéseket, amelyekről később még lesz szó.

A következő alapfogalmak csoportja segít megérteni az online tér veszélyeit. Ilyen a *Threat* vagyis fenyegetés, és a *Vulnerability*, vagyis sebezhetőség. A Threat bármilyen olyan kárt, veszteséget vagy romboló hatást jelent, amely kárt okozhat az információtechnológiai rendszerekben, ilyen lehet például egy természeti katasztrófa, vagy esetleg kibertámadások, adatintegritást sértő események, vagy ha bizalmas információkat nyilvánosságra hoznak. Ezek nagyrészt kívül esnek a dolgozat témáján, ezért inkább a sebezhetőségeket (*vulnerabilities)* részletezem, amelyek jelenthetnek bármilyen gyengeséget a szoftver tervezésében és implementálásában, és olyan belső faktorokból származnak, mint szoftver hibák (*software bugs*), rosszul konfigurált szoftverek, nem megfelelően védett hálózati eszközök, hiányzó biztonsági javítások (*security patch),* vagy a fizikai biztonság hiánya. A fenyegetések fontossága a szoftverfejlesztés szempontjából ott mutatkozik meg, hogy ahol a fenyegetések és sebezhetőségek „keresztezik egymást”, ott van a vállalati rendszerek és hálózatok kockázata. Ha van fenyegetés, de nincs hozzá tartozó sebezhetőség, akkor nincs kockázat (ennek elérése a biztonságos programozás célja, a sebezhetőségek elkerülése, minimalizálása). Igaz viszont az is, hogy ha van egy sebezhetőség, de nincs ellene fenyegetés, szintén nincs kockázat. Ez az alapja a kockázatelemzésnek.

Fontos még tisztázni azt is, hogy a kockázatok minimalizálhatók, de sosem küszöbölhetők ki teljesen. 100%-os biztonság nem létezik, ha létezne is, nagyon

kényelmetlen, nehézkes és körülményes lenne a felhasználók számára (de nem létezik!). A biztonság megfelelő szintjének megtalálása kockázatelemzést igényel, amely során figyelembe veszik a lehetséges problémákat, azok költségét (ha bekövetkeznek), és a megelőzésük költségét. Felesleges például a múlt heti lottószámokat titkosítani és gondosan őrizni, mert az már egy nyilvános adat. Viszont nagyon fontos egészségügyi adatokat őrizni, bankkártya-adatokat és egyéb személyes adatokat. A tervezés során megfelelő egyensúlyt kell találni a biztonság szempontjából, mivel a nagyobb biztonság mindig kényelmetlen, a felhasználók hátráltató tényezőként élik meg, igyekeznek elkerülni. Ez is oka annak, hogy mindig lesznek biztonsági incidensek. A lényeg, hogy minden tőlünk telhetőt megtegyünk a biztonság érdekében az alkalmazás készítésének minden egyes fázisában.

A továbbiakban szó lesz azokról a veszélyforrásokról, sebezhetőségekről, amelyek a szoftverfejlesztés során egyes összetevők nem megfelelő implementálása miatt alááshatják az alkalmazások biztonságát, veszélyeztethetik a korábban tárgyalt alapelvek érvényesülését.

2.2. Fenyegetésvektorok és támadási felületek

A fenyegetésvektor (*Threat Vector*), a támadó által a támadáshoz használt program vagy út, tulajdonképpen a támadás elkövetésének módszere. Eszköz vagy útvonal, amellyel a támadó jogosulatlanul hozzáférhet egy számítógéphez vagy hálózathoz, hogy azon valamilyen nem kívánt műveletet hajtson végre, vagy rosszindulatú ’rakományt’ szállítson (malware-t).

A támadási felület (*Attack Surface*) az összes olyan pont, ahol egy jogosulatlan felhasználó megpróbálhat a rendszerbe bejutni, adatokat bevinni vagy adatokat kinyerni onnan. Ez minimalizálható a hozzáférés korlátozásával, felesleges szoftverek, szoftverelemek kerülésével és eltávolításával, és a nem használt protokollok tiltásával. A jól átgondolt, biztonságos tervezés is elősegíti a támadási felületek megelőzését, melynek során pontosan meghatározzák milyen funkciókra van szükség, és csak a valóban szükségeseket valósítják meg. A legbiztonságosabb kód igazából az, amit meg sem írunk (persze a felhasználói igények változnak, amelyekre lehet számítani,

de a felesleges funkciók megvalósítása kerülendő, mert hibákat, sérülékenységeket rejthetnek teljesen feleslegesen).

A fenyegetés vektor tekinthető a támadás hogyanjának, a támadási felület pedig a támadási pontnak (tehát ahol a támadás megtörténik).

A fenyegetés vektorok tehát olyan biztonsági rések, ahol az alkalmazás, illetve a rendszerünk támadható. Ilyen fenyegetés vektor az üzenetek (*Messages*), az e-mailben, egyszerű üzentküldési szolgáltatásban SMS-ben küldött üzenetek, vagy az azonnali üzenetküldési egyéb formáiban küldött fenyegetések. Többnyire adathalász törekvések eszközei, amikor a támadó megbízhatónak adja ki magát, hogy áldozataitól érzékeny adatokat szerezzen meg.

A képek szintén fenyegetés vektorok lehetnek, a szteganográfia segítségével a képfájlba kártékony kód ágyazható be a kép érzékelhető minőségromlása nélkül. Szabad szemmel teljesen észrevehetetlen.

A fájlok, legitimnek tűnő dokumentumok, szoftverek telepítőfájljai, e-mail mellékletek sem biztosan veszélytelenek. Ezek fájlmegosztó szolgáltatásokon keresztül is továbbíthatók, mint az illegális kalóz szoftverek, és rosszindulatú webhelyeken tárolhatók.

A nem biztonságos hálózatok is veszélyeket rejtenek, amelyek olyan vezetékes, vezeték nélküli vagy Bluetooth hálózatok, amelyek nem rendelkeznek a megfelelő biztonsági intézkedésekkel. Ebben az esetben a hálózati forgalom illetéktelenek számára hozzáférhető, esetleg lehallgatható. A vezeték nélküli hálózatok könnyen hozzáférhetők így, a vezetékesek kissé nehezebben, a közegbeli eltérések miatt, de a hálózati infrastruktúrához való fizikai hozzáférés minden esetben különféle támadásokhoz vezethet. Az adatokat tehát titkosítottan kell továbbítani, és ellenőrző összegek (*checksum*) segítségével ellenőrizni kell sértetlenségüket (erről az adatok védelménél még lesz szó). A Bluetooth protokoll sérülékenységeinek kihasználása pedig *Blueborne*, vagy *BlueSmack* támadásokat eredményezhet. A *BluBornhoz* a Bluetooth technológia sebezhetőségeinek halmaza vezet, amelyek lehetővé teszik a támadók számára, hogy átvegyék az eszközök feletti irányítást, rosszindulatú programokat terjesszenek, vagy támadást indítsanak a kommunikáció elfogására. A *BlueSmack* a *Denial of Service*, vagyis szolgáltatásmegtagadási támadás típusa,

amely a Bluetooth kompatibilis eszközöket célozza. Ha Bluetooth technológiát akarunk használni a szoftverünkben, számolni kell annak sebezhetőségével a fejlesztés során.

A teljesség kedvéért – bár programozás szempontjából nem lényeges – a *Vhising* is a fenyegetések közé tartozik. Ez a *Phishing* vagyis adathalászat mintájára elkövetett hanghívást (*Voice Call*) jelenti, melynek célja szintén érzékeny információk, akár bankkártyaadatok megszerzése.

Szintén veszélyesek az eltávolítható eszközök (*Removable Device*), az ezekkel elkövetett technika egyébként a *Baiting* (csalizás). A támadó egy rosszindulatú szoftverrel fertőzött USB eszközt helyez el nyilvános helyen, ahol célpontja megtalálhatja és megfertőzheti vele a számítógépét.

A programozás szempontjából ez a kérdés felhívja a figyelmet azokra a területekre, azokra a résekre, ahol az alkalmazásunk sebezhető lesz. Érdemes ezt is átgondolni, és minimalizálni a kockázatokat. Már a tervezés során érdemes összegyűjteni a sérülékeny technológiákat, programösszetevőket, akár már az UML diagramokon is jelölhetők a sérülékeny komponensek, pontok.

2.3. Támadási minták

A támadási minták megvizsgálásával a támadók módszereit igyekszünk megérteni, azzal a céllal, hogy megértsük mi ellen kell védekezni egy jól működő alkalmazásnak. Ezek különféle súlyosságú információszerzési és beavatkozási módok, amelyek segítségével elérhetők a támadók céljai.

Érdemes röviden átgondolni, hogy milyen módon gyűjthetnek információkat a támadók a weboldalakról, webalkalmazásokról, amely információgyűjtés előkészítése lehet különféle támadásoknak.

Az információgyűjtés első fázisa a felderítés, felmérés, feltárás (*Reconnaissance*). Ennek során még semmi illegális nem történik, a felhasznált komponensek verziószámait, a webalkalmazás készítéséhez felhasznált eszközökkel, az alkalmazás felépítésével kapcsolatos információkat próbálnak gyűjteni. Megvizsgálják az URL-eket, azok felépítését, próbálnak beírni az URL-ekbe, továbbá nyitott portokat

keresnek, amelyeken később hálózati forgalom bonyolítható. Ezeket az információkat el kell rejtenünk, nem szabad olyanokról tájékoztatni a felhasználót (a támadót meg főleg), amiről nem muszáj (ez az információgyűjtési tevékenység is mutatja egyébként a naplózás és hozzáférési minták nyomon követésének fontosságát).

Az *Excavation* (kiásás) magában foglalja az URL-ek és lekérdezési láncok vizsgálatát és módosítását, a konfigurációk és beállítások módosítását, a weboldalak forrásainak megtekintését, rendszernaplók felfedezését is. Itt már visszaélési taktikák felhasználása történik (például érvénytelen vagy nem szokványos bemeneti értékek megadása hiba kikényszerítése céljából, vagy esetleg kísérlet a kezeletlen kivételek felfedésére, hibaüzenetek kikényszerítése információszerzés céljából, konfigurációs adatok, útvonalak és egyebek felderítésére). Ez a tevékenység már egy fokkal súlyosabb a felderítésnél, itt már történhet bizonyos fokú beavatkozás.

A *Footprinting* (lábnyomok keresése) során konfigurációs információkat keresnek, amelyek hasznosak lehetnek egy támadásnál (nyitott portok, verziószámok, hálózat topológia információk stb), a *Fingerprinting* (ujjlenyomat keresés) során pedig a rendszer kimenetét ismert „ujjlenyomatokkal” hasonlítják össze, amelyek egyedileg azonosítják a rendszer részleteit.

A *Reverse Engineering* alkalmazásával egy objektum, erőforrás vagy rendszer szerkezetét, funkcióját és összetételét elemzik a visszafejtés céljából. A cél itt is minél több információ szerzése a célpontról.

A *Functionality Misuse* (funkcionalitás visszaélés) alkalmazásával a támadók az alkalmazás funkcióival próbálnak visszaélni negatív műszaki hatás elérése érdekében. Ekkor nem módosítanak a funkciókon, azok hibáit akarják kihasználni más módon, mint amire a funkciókat használják és tervezték (ezért kell jól átgondoltan tervezni, és a lehető legkevesebb funkciót implementálni).

A támadások következő lépcsője a *Gain Access Privileges*, vagyis a hozzáférési jogosultságok szerzése. Ez már aktív támadásnak számít. Megvalósítható *Brute force attacks,* vagyis nyers erővel végrehajtott támadásokkal, amely során gyorsan és ismételten visznek be, próbálgatnak különböző nem ismert értékeket. Ha például jelszó megszerzésére irányul a támadás, a helyes érték megfejtése céljából különböző lehetőségeket próbálnak ki egymás után, hogy hozzáférést szerezzenek. Ezzel a módszerrel próbálják kitalálni a jelszavakat, titkosítási kulcsokat, adatbázis-kezelési

kulcsokat, vagy más hasonlókat. Ha például a jelszó kisebb karakterkészletet használ (például csak betűk és számok), vagy rövidebb, az kevesebb variációs lehetőséget jelent a nyers erővel való végrehajtott feltörés során. A szótári szavak használata is megkönnyíti a jelszótörés folyamatát, például a *John (John the ripper*) alkalmazás szótárakat is képes használni (amely különböző szótári szavak hashelt értékeit hasonlítja össze a jelszóhash-el, a jelszavak elleni támadásokról szóló fejezetben lesz még szó erről, és példa).

Az *Authentication abuse* jogosulatlan hozzáférést jelent a hitelesítés gyengeségei miatt, az *Authentication bypass* pedig jogosulatlan kiváltságos hozzáférést a normál hitelesítési útvonalon kívül (például a felhasználók bejelentkezését követően egy rosszul megtervezett webhely betölt egy érzékeny információkat tartalmazó URL-t, amit ha a támadók a címsorba másolnak hitelesítés nélkül közvetlenül a védett tartalomhoz juthatnak).

A *Memory manipulation* alkalmazásával a támadók jogosulatlanul hozzáférnek memóriahelyekhez, ahova írni is képesek. A *Buffer manipulation*, az adatok olvasását vagy írását jelenti nem hagyományos módon, a normál folyamatokat megkerülve. Ezzel a támadó a megfelelő memórián kívül is tud írni vagy olvasni adatokat. A támadás során a memóriába bevitt tartalom többnyire nem számít, a lényeg, hogy kiszoruljon az eredeti puffertartalom, túlcsorduljon, ezzel olvashatóvá vagy írhatóvá váljon egy másik memóriahely.

Az arra érzékeny programozási nyelvekkel készített alkalmazások esetében (ahol a programozók kezelhetik a pointereket, például a C nyelvben) előfordulhat, hogy a *Pointer manipulation* alkalmazásával is támadhatók lesznek. Ez a pointerváltozók módosítását jelenti, amivel nem kívánt memóriahelyek érhetők el, így olyan adatokhoz vagy funkciókhoz férhetnek hozzá a támadók, amelyek normál esetben nem lennének elérhetők.

A *Parameter Injection* támadások alkalmazásával a kérés paraméterek módosítása történik meg. Az *Input Data Manipulation* a webes beviteli adatok módosítását jelenti.

Az *Action Spoofing* műveletek egyes műveletek elrejtését és másik műveletnek álcázását jelentik, céljuk ezzel befolyásolni a felhasználót, hogy tudtán és akaratán kívüli műveleteket is végrehajtson.

A *Software Integrity Attack* típusú támadások rábírják a felhasználót, alkalmazást, kiszolgálót vagy eszközt olyan műveletekre, amelyekkel a szoftverkód, eszköz, vagy különféle adatszerkezetek integritásának, vagyis sértetlenségének sérülése történik meg. Az integritás alapelvének sérülése miatt a célpont nem biztonságos állapotba kerül, ez egy újabb támadás előkészítésének kockázatát hordozza magában.

Az *Infect the Application with Malicious Code* típusú támadásokkal saját, rosszindulatú kódot juttatnak be az alkalmazásba különféle technikák felhasználásával (*Code Inclusion, Code Injection, Command Injection, Content Spoofing, Resource location spoofing).*

A *Denial of Service* típusú támadásokkal bizonyos műveletek elvégzésével a jogos felhasználók hozzáférését akadályozzák meg a szoftverhez. A szolgáltatás megtagadása többféle módon is megvalósítható, mindegyiknek az a célja, hogy letiltsák a szolgáltatásokat vagy annyira leblokkolják, túlterheljék a rendszert, hogy az már ne tudja támogatni a szolgáltatást (formái az *Excessive allocation,* vagyis túlzott elosztás, a *Flooding* vagyis elárasztás, a *Resource leak exposure,* vagyis Erőforrás-szivárgás expozíció, és a *Sustained client engagement,* vagyis tartós ügyfélelköteleződés).

A *Repudiation*, vagyis megtagadás, elutasítás elérésével a támadó úgy bonyolít le tranzakciókat, hogy a rendszer ne tudja bizonyítani, hogy az valójában megtörtént. A támadó manipulálhatja a rendszert, hogy hibás adatokat naplózzon (valamely folyamat nem történt meg, vagy egy másik folyamat naplózása az eredeti helyett). Ez akkor fordul elő, ha a rendszer nem követi megfelelően a felhasználók műveleteit, vagy valami oknál fogva nem védik megfelelően a naplókat. *Repudiation* történik, ha a támadó egy hitelkártyás vásárlás során lehetetlenné teszi a rendszer számára a vásárlás bizonyítását.

A tömör áttekintés után, amelynek célja csak a rosszindulatú támadók elkövetési módszereinek felsorolása volt, később, a *Vulnerabilities and attacks* című fejezetben részletesebben is lesz még szó a sebezhetőségekről és a támadásokról.

2.4. Third-party Vendor Risks – Harmadik féltől származó összetevők kockázatai

A *Third-party Vendor Risks* alapvetően azokat a potenciális biztonsági és működési kihívásokat foglalja magában, amelyeket a velünk együttműködő külső szervezetek jelentenek, lehetnek azok eladók, beszállítók vagy szolgáltatók. A külső partnerek valamilyen módon történő beintegrálásával, potenciális fenyegetéseknek és sebezhetőségeknek tesszük ki magunkat, amelyek természetesen hatással lehetnek a korábban tárgyalt integritásra, és az adatbiztonságra.

A veszélyt jelentheti egy hardverszolgáltató sebezhető alkatrészekkel, egy szoftverszállító *backdoor*t („hátsó ajtó”) tartalmazó szoftverekkel, vagy akár egy olyan szolgáltató, aki hozzáférhet érzékeny adatokhoz, de nem rendelkezik megfelelően szigorú kiberbiztonsági protokollal.

Az alkalmazás fejlesztése során gyakran szükség van harmadik fél által fejlesztett szoftverkomponens felhasználására is. Fontos, hogy ezek mentesek legyenek sebezhetőségektől és hibáktól, mert az ilyen összetevők felhasználásával már sérülékeny lehet az alkalmazásunk, az első sor kód megírása előtt. Érdemes egy jó vírusirtó vagy rosszindulatú szoftverek elleni eszközzel átvizsgálni, hogy mentes legyen minden rosszindulatú kódtól (a nyílt forráskódú szoftver esetében a legkönnyebb ennek ellenőrzése, a kód áttekinthetősége miatt).

Ha egy szolgáltató valamilyen szoftverszolgáltatását használjuk (például egy felhőszolgáltatás esetén), meg kell vizsgálni, hogy hogyan tudja biztosítani azt, hogy a hozzá kerülő információk valóban megőrizzék a titkosságot és az integritást. Meg kell nézni azt is, hogy a szolgáltató biztonsági protokolljai elég erősek-e ahhoz, hogy megvédjék az adatainkat, és azt is, hogy ha bekövetkezik egy biztonsági incidens, a szolgáltató megfelelő felszereltséggel rendelkezik-e ahhoz, hogy biztosítsa a megfelelő támogatást (rendelkezik-e megfelelő naplófájlokkal, mindenféle szükséges bizonyítékkal, amik a nyomozást segíthetik, ha szükség lesz erre).

A felhasznált komponensek, szoftverek esetében lehetnek különböző platformok, alkalmazásokká összeállított modulok, alkalmazáson kívüli modulok, helyi külső API-k, amelyeket az alkalmazás hív meg, és web-és felhőszolgáltatások, amelyeket az alkalmazás hív meg. Ezek sebezhetőségi állapotával kapcsolatban többféle

információforrás is van. Az egyik ilyen az *Exploit Database* [6] *(exploit-db)*, ahol sebezhetőségi típus, sebezhető platform, port és egyéb szűrési paraméterek beállításával kereshetünk (ha például *NodeJS* sebezhetőségeket keresünk a platform szűrési paramétereknél ezt kell kiválasztani, és a találatok időrendi sorrendben, dátummal, leírással – benne verziószámmal – és típusinformációkkal meg is jelennek). Az *exploit-db* egyébként ennél is többet tud, előfordulhat a sérülékenység kihasználásának leírása is, azt illusztráló kóddal (*POC – Proof of Concept* – a sebezhetőség kihasználásának részletes leírása).

További sebezhetőségi adatbázisok is léteznek még, ezeket általában különféle kormányzati és kereskedelmi szervezetek adnak ki. Átfogók, tartalmaznak keresési lehetőségeket, és automatikusan is kereshetők automatizálási scriptek segítségével, és szintén előfordulhat bennük *POC* is. Ilyen az USA-ban a *National Vulnerability Database*, [7] ez az *SCAP (Security Content Automation Protocol)* protokoll alapján, géppel olvasható formátumban van tárolva. Tartalmazza a biztonsági ellenőrzőlisták adatbázisait, a biztonsággal kapcsolatos szoftverhibák hibás konfigurációit, a termékneveket és a hatásmérőket.

További információforrás a *Command Attack Pattern Enumeration and Classification (CAPEC)* [8] oldala is, amely gyakori támadási mintákat tartalmaz, karbantartója a *MITRE Corporation*.

Az *OWASP Top 10* [9] a gyakori fenyegetések listáját és leküzdésük stratégiáit mutatja be, és az *Open Web Application Security Project* a karbantartója, és neve alapján is látható, hogy webes sérülékenységekre specializálódott.

Ha más alkalmazások sérülékenységeit kutatjuk, vannak általánosabb, másfajta alkalmazásokat érintő adatbázisok is, ilyen a CWE/SANS Top 25 [10] oldala, azon belül is a CWE Top 25 Most Dangerous Software Weaknesses, [11] amely a legelterjedtebb és legkritikusabb szoftverhibák listáját tartalmazza, amelyek súlyos szoftver sérülékenységekhez vezethetnek. Míg az OWASP a webes és mobilalkalmazásokra összpontosít, a CWE/SANS mindenféle szoftvert lefed, beleértve az asztali alkalmazásokat is. A biztonsági közlemények és tanácsok is segítséget jelenthetnek, van ilyen a Microsoftnak [12], az Apple-nek [13], az Androidnak [14], az Ubuntunak [15], a Google Chrome-nak [16], a JQuerynek [17] is többek között, természetesen az adott technológiára vonatkozóan.

Léteznek még nyílt forráskódú szoftverprojektek hibakövetői is, mint a NodeJS [18][19], a Python Bug Tracker [20], a Docker [21] és MySQL [22] hibáit soroló oldalak, a teljesség igénye nélkül.

Érdemes ezeken az oldalakon körülnézni, hogy a felhasználni kívánt komponens esetleg sérülékeny-e (vagy előfordulhat az is, hogy sérülékeny volt egy korábbi verzióban, de már javították).

3. Data Protection – Adatvédelem [25][29][30][31][32]

Az adatvédelem, az adatok védelmét jelenti az illegális hozzáféréstől, az illegális adatmódosítástól, továbbá magában foglalja az adatvesztés elkerülését is. A hatékony védelem érdekében az adatokat osztályozni kell, ezzel jelezve a szükséges biztonság és védelem szintjét. Egy lehetséges osztályozási mód szerint lehetnek nyilvános, érzékeny, privát, bizalmas és kritikus adataink. A lényeg, hogy elkülönítsük és rangsoroljuk az adatainkat fontossági, így adatvédelmi szempontból, és kidolgozzuk az adatvédelmi irányelveinket.

3.1. Adatcsoportok

Amikor adatokról beszélünk, lehetnek azok mindenfélék: üzleti titkok, szellemi tulajdon, jogi információk, pénzügyi információk, ember által olvasható és ember által nem olvasható adatok. Ezek megőrzésére vonatkozóan lehetnek jogi előírások, amelyeknek meg kell felelni (meddig kell tárolni az adatokat, esetleg meddig tárolhatók az adatok stb.). A „szabályozott adatok” (*Regulated Data*) kezelését törvények, rendeletek, vagy iparági szabványok szabályozzák, megfelelőségi követelmények vonatkoznak rájuk például a GDPR (*General Data Protection Regulation*).

Másik adatcsoport a személyes azonosító adatok (*PII – Personal Identification Information*), mint az azonosító okmányaink számai, TB számok, címek. Ezek bűnözők célpontjai lehetnek, mivel személyazonosításra alkalmas adatok, nagyon komoly visszaéléseket lehet elkövetni, ha a támadó ismeri ezeket. Egyértelmű tehát, hogy nagyon fontos adatokról van szó, amelyeket gondosan védeni kell.

A védett egészségügyi adatok (PHI – *Protected Health Information*) csoportjába egy adott személyhez kapcsolódó egészségügyi állapotra, egészségügyi ellátásra és ezzel kapcsolatos fizetésre vonatkozó adatok tartoznak. Magyarországon ezt az 1997. évi XLVII. az egészségügyi és a hozzájuk kapcsolódó személyes adatok kezeléséről és védelméről szóló törvény szabályozza, és a 2025/327. Európai Parlament és a Tanács rendelete (2025. 02.11) az európai egészségügyi adattérről szól. Kevésbé érzékenyek mint a személyazonosításra alkalmas adatok, de magánjellegűek, így szintén

védelemre szorulnak (bár alacsonyabb szinten elegendő, mint a személyazonosító adatokat).

Az üzleti titkok versenyelőnyt biztosító bizalmas információkat jelent, védelmük fontossága egyértelmű, természetesen jogi védelmet is élveznek. Magyarországon a 2018. évi LIV. az üzleti titok védelméről szóló törvény szabályozza ezt.

Hasonló ehhez a szellemi tulajdon adatcsoportja is, ide tartoznak mindenféle egyedi alkotások, találmányok, irodalmi művek stb. Szerzői jogok, szabadalmak védjegyek védik ezeket. Magyarországon ezt szabályozza a 1999. évi LXXVI. a szerzői jogról szóló törvény.

Újabb adatcsoportot alkotnak a jogi információk, amelyek jogi eljárásokkal, szerződésekkel jogszabályi megfeleléssel kapcsolatos adatok. Szintén magas szintű védelmet igényelnek, az ügyfélbizalom megőrzése érdekében is.

A pénzügyi információk pénzügyi tranzakciókkal kapcsolatos adatok, például bizonylatok, számlák, bankszámlakivonatok, adóbizonylatok, bankkártyaadatok. Ezek szintén bűnözők célpontjai lehetnek csalás és személyazonosságlopás elkövetése céljából. Ahol vásárlás történik, ott pénzügyi adatokat használunk, szóval gyakori adatcsoportról van szó, amelyet magas szinten biztosítani kell, és az előírt ideig őrizni.

Az emberek által olvasható és emberek által nem olvasható adatok csoportja abból a szempontból különíti el az adatokat, hogy ember által azonnal megérthető, vagy esetleg mondjuk binárisan kódolt adatokról van-e szó. Az emberek által olvasható adatok egyértelműen kritikusságuk szerinti védelemre szorulnak, mivel az emberek számára is azonnal értelmezhetők (elolvashatók). Kicsit nehezebben dekódolhatók az emberek által nem olvasható adatok, mert értelmezésükhöz gép vagy szoftver szükséges. Ennek ellenére, ha érzékeny információkat tartalmaz, ezeket is védeni kell.

3.2. Adatállapotok

Az adatok védelme magában foglalja a különböző állapotban lévő adatok védelmét is, az adatok ugyanis több állapotban is jelen vannak egy alkalmazásban. Lehetnek nyugvó adataink, ezek statikus állapotban vannak tárolva az adatbázisokban, fájlrendszerekben. Valószínűleg ez a leggyakoribb támadási felületük. Titkosítással

védhetjük ezeket, akár egész meghajtó titkosítással (*Full Disk Encryption – FDE,* például a Windowsban a BitLocker segítségével), az adott partíció titkosításával, esetleg csak az adott fájl titkosításával, fájl- vagy könyvtár (*directory*) titkosításával, adatbázis titkosításával (történhet ez oszlop, sor, vagy táblaszinten), vagy az adatbázis adott rekordjának titkosításával, attól függően, hogy milyen szintű biztonságra van szükségünk.

Miközben adatokat használ az alkalmazásunk, az adataink mozgásban vannak, akár úton valahol a hálózaton, akár a RAM-ban, úton a processzor felé. Ez is egy támadható állapotuk, így ebben az állapotban is védenünk kell őket. Szállítási titkosítási eszközeink az SSL (*Secure Socket Layer*) és a TLS (*Transport Layer Security*). Ezek biztonságos kommunikációt biztosítanak a hálózaton belül a webböngészéshez és email-ezéshez. A mozgó adatokba való belehallgatás megelőzésének másik eszköze a VPN (*Virtual Private Network*) használata, amely biztonságos kapcsolatot hoz létre a kevésbé biztonságos hálózatokon, így az interneten. Az IPSec (*Internet Protocol Security*) pedig az IP csomagok hitelesítésével és titkosításával biztosítja az IP kommunikációt.

Ezen túl az éppen használt, tehát éppen módosítás, feldolgozás alatt álló adatok állapotáról sem szabad megfeledkeznünk (CRUD műveletek, *create* - létrehozás, *read* - olvasás, *update* - frissítés, *delete* – törlés), amelyeken a processzor éppen módosítást végez. Ezeket védhetjük úgy is, ha titkosítjuk az alkalmazás szintjén, tehát feldolgozás közben titkosítjuk az adatokat. További biztonsági intézkedés a hozzáférés-vezérlés (*Access Controls*) implementálása is, amely korlátozza a feldolgozás során hozzáférhető adatok körét, vagy létrehozhatunk elszigetelt környezetet az érzékeny adatok feldolgozásához (*Secure Enclaves*). A teljesség kedvéért léteznek továbbá olyan mechanizmusok, mint például az INTEL Software Guard megoldása, amelyek a memóriában lévő adatokat titkosítják, az illetéktelen hozzáférés elkerülésére, tehát hardveres megoldások is segítenek.

3.3. Adatvédelmi módszerek

Az adatbiztonság elérésének módszerei a földrajzi korlátozások (*Geofencing*), a titkosítás (*Encryption*), a hash-elés (*Hashing),* az adatmaszkolás (*Masking*), a

tokenizálás (*Tokenization),* az obfuszkálás *(Obfuscation),* a szegmentálás *(Segmentation),* és az engedély korlátozás *(Permission Restriction).*

A földrajzi korlátozásokkal (*Geofencing*) virtuális határokat hozhatunk létre, az adatok hozzáférésének hely szerinti korlátozására. Ezzel betartjuk a helyi törvényeket, ha eltérő jogi szabályozással rendelkező földrajzi helyeken is jelen vagyunk, és megakadályozhatjuk a hozzáférést a magas kockázatú helyek irányából. Az adatszuverenitás (*Data Sovereignty*) azt jelenti, hogy az adatokra, a digitális információkra annak az országnak a törvényei vonatkoznak, ahol az található. Ez a felhőszolgáltatók világában, ahol az adatok határokon átnyúlóan áramlanak, nem is olyan egyszerű kérdés.

A titkosítással (*Encryption*) a normál, olvasható szöveget (*plaintext*) alakítjuk át algoritmusok és kulcsok segítségével kódolt, titkosított szöveggé (*ciphertext*). A nyugalomban lévő és a mozgó adatok védelmének módszere, az adatok visszafejtéséhez szükséges a megfelelő kulcs ismerete.

A kivonatolás (*Hashing)* segítségével az adatokat rögzített méretű, numerikus vagy alfanumerikus kivonatkértékké (*hash value*) alakítjuk át, egy visszafordíthatatlan, egyirányú megoldás segítségével, amelyet nem lehet visszafejteni (ez különbözteti meg a titkosítástól). Gyakran jelszavak titkosítására és tárolására használjuk, illetve fájlintegritás ellenőrzésre ellenőrző összegek számolásával (checksum).

Az adatmaszkolás (*Masking*) alkalmazásával adatok egészét vagy egy részét helyőrzőkkel helyettesítik (pl. * karakterekkel), hogy elrejtsék az eredeti tartalmat. Ha a bankkártyás fizetés alkalmával a bizonylaton csak a bankkártya utolsó 4 számjegyét látjuk, akkor ott adatmaszkolással védik az érzékeny bankkártyaszámunkat. Ez szintén egyirányú, visszafordíthatatlan folyamat.

A tokenizálás (*Tokenization),* az érzékeny adatokat nem érzékeny helyettesítő adatokkal, tokenekkel helyettesíti. Az eredeti adatokat biztonságosan tárolják egy adatbázisban, a tokeneket használják fel hivatkozásként. Általában fizetésfeldolgozó rendszerekben alkalmazzák a bankkártya-adatok védelmére.

A „ködösítés, homályosítás” *(Obfuscation)* az adatokat érthetetlenné teszi, ezzel nehezíti az illetéktelen felhasználók általi megértést. Különböző technikákat jelent, mint például a titkosítás, adatmaszkolás és pszeudonevek használata.

A szegmentálással *(Segmentation)* a hálózatot több különálló részre bontjuk, amelyek külön-külön saját biztonsági ellenőrzéssel rendelkeznek. Ha egy szegmensbe be is jut illetéktelen felhasználó, a többihez nem fér hozzá továbbra sem, ezzel korlátozható az általa okozott kár mértéke.

Az engedély korlátozás *(Permission Restriction)* azt határozza meg, hogy ki férhet hozzá az adatokhoz és mit tehet velük. Ezt hozzáférési lista (*Access Control List*) és szerepkör alapú hozzáférés megvalósításával (*role-based access control* RBAC) érik el.

A *Data Loss Prevention (DLP)* az adatvesztés megakadályozását jelenti, amely egy újabb stratégia a szervezet érzékeny adatai kiszivárgásának megakadályozása. A DLP rendszerek (*Data Loss Prevention Systems*) figyelik az adatokat minden állapotukban, azzal a céllal, hogy felfedjék az adatlopási törekvéseket. Lehetnek szoftveres vagy hardveres megoldások. A végponti rendszer egy munkaállomásra vagy laptopra telepített szoftver, és figyelemmel kíséri az eszközön használt adatokat. Ha valaki fájlátvitelt akar kezdeményezni, vagy leállítja azt, vagy megfelelő szabályok és irányelvek alapján értesíti a rendszergazdát az eseményről. Hasonló az IDS-hez (*Intrusion Detection System* – Behatolásjelző rendszerek) és IPS-hez (*Intrusion Prevention System* – Behatolás megelőző rendszerek), csak adatokra koncentrálva, és beállítható észlelési vagy megelőzési módra. A DLP-nek van hálózatra vonatkozó megoldása is, ami a rendszer peremén elhelyezett szoftvert vagy hardvert jelenti, továbbá a tárolási DLP pedig adatközpontokban lévő szerverekre telepített megoldás, a felhőalapú DLP pedig felhőszolgáltatók adatvesztésének megakadályozására szolgál például a Google Drive tárhelyszolgáltatás részeként (Microsoft 365-nél is van).

A szervezetünknek, az alkalmazásunknak rendelkeznie kell egy adatkezeléssel, adatvédelemmel kapcsolatos irányelvvel, amely meghatározza az adatok osztályozási, megőrzési és selejtezési követelményeit az aktuális helyi jogszabályoknak megfelelően. Ha az adatokat rendszereztük meg kell tervezni milyen módon, milyen eszközök és lehetőségek felhasználásával védjük meg azokat. Részletesen végig kell gondolni az adatokkal kapcsolatos problémákat, és gondosan megvalósítani a saját megoldásunkat.

3.4. Identity and Access Management (IAM) – Azonosság- és hozzáférés kezelés

Az azonosság- és hozzáférés-kezelés (IAM) az információbiztonság alapvető eleme, biztosítja, hogy a jogosult személyek a megfelelő időben és megfelelő okból a számukra engedélyezett erőforrásokhoz férjenek hozzá. Az ennek megvalósításához használt technológiák olyan eszközöket biztosítanak az üzleti folyamatokhoz, amelyek megvalósítják és megkönnyítik az elektronikus személyazonosságok kezelését, tehát a jelszókezelést, a hálózati hozzáférés-ellenőrzést és a digitális személyazonosság kezelését.

A hatékony IAM biztonsági ellenőrzéseket végez, azonosítási technikákat alkalmaz, megvalósítja a hozzáférés-ellenőrzést és a fiókkezelést, hogy csak a jogosult felhasználók férjenek hozzá, a számukra elérhetőnek szánt tartalomhoz. A folyamat összetevői az *Identification* vagyis azonosítás, mely során a felhasználó megkülönböztetve magát a többiektől, felhasználónév vagy e-mail cím formájában azt állítja magáról, hogy ő a tulajdonosa a fióknak, melybe be szeretne lépni. Az *Authentication*, vagyis hitelesítés során ennek a személyazonosságnak az ellenőrzése történik meg. Az *Authorization*, vagyis engedélyezés a sikeres hitelesítést követő folyamat, melynek során a felhasználó hozzáfér a jogosultságának megfelelő tartalomhoz, a megfelelő hozzáférési szinten. Az *Accounting*, vagyis elszámolás, könyvelés, auditálás tulajdonképpen naplózás, amely fontos a felhasználói tevékenységek követéséhez, rögzíti az eseményeket, nyilvántartást készít a megfelelő biztonsági felügyelet érdekében. Az IAM-nek tehát meg kell valósítania a felhasználói fiókok rendelkezésre bocsátását és megszüntetését, a személyazonosság igazolásának folyamatát, majd a tanúsítást is.

A hitelesítés folyamata történhet a felhasználónév és jelszó felhasználásával. Ebben az esetben egyetlen faktor biztosítja a felhasználói fiókunkat (a jelszó), ezért nagyon fontos, hogy ez biztonságos legyen. Ennek biztosításához jelszóbiztonsági irányelveket kell kidolgozni és kikényszeríteni a regisztrációs és jelszóváltoztató modulban. Például a védendő információ érzékenységétől függően kikényszeríthetjük a jelszavak előírt időszakonként történő megváltoztatását, annak ellenőrzését, hogy új jelszót adnak-e meg a felhasználók (korábbi jelszavakkal való egyezés ellenőrzése,

és azok kis mértékben módosítása, például a régi jelszóhoz hozzáírnak egy számot, amely nem ajánlott gyakorlat, érdemes megelőzni). Sajnos a biztonságos jelszavak nehezen megjegyezhetők és többszöri felhasználásuk sem ajánlott, ezért felhasználóinknak ajánlhatunk biztonságos jelszókezelőket segítségképpen. Érdemes felhívni a figyelmet egy rövid tájékoztatóban arra is, hogy nem ajánlott szótári szavakat használni jelszóként, mert ezek könnyebben feltörhetők. A véletlenszerű karaktersorozatból álló jelszavakat a hosszabb, minél szélesebb karakterkészletből kialakított megoldásuk teszi még biztonságosabbá (számok, kis- és nagybetűk, írásjelek, speciális karakterek, viszont ez megteremtheti az Injection típusú támadások lehetőségét, ezért a jelszó bevitelére szolgáló mezők esetében mindig ügyelni kell a Sanitization-re, erről később még lesz szó). A jelszókezelők általában biztonságos jelszógenerátorokat is tartalmaznak (amelyek megfelelnek az előbbieknek, sőt kiválaszthatjuk a jelszó hosszát is) és tárolják is a generált jelszavakat.

Magasabb szintű biztonságot érhetünk el a bejelentkezés megvalósításakor, ha több faktort használunk a hitelesítés során (*Multifactor Authentication - MFA*). A több faktor azt jelenti, hogy a felhasználónak van valamije, amit tud (ezt valósítja meg a jelszó, biztonsági kérdések stb.), valami, aminek birtokosa (beléptető kártyák, biztonsági kulcsok, belépési kulcsok), valami, ami a felhasználó része (biológiai értelemben, ujjlenyomat, arckép, stb), valami amit csinál (mozgásminta felhasználása az azonosításhoz), és a helyszín, ahol van (a tartózkodási hely felhasználása, például a világ másik végén a felhasználó nevében kezdeményezett tranzakció, illetve bejelentkezési kísérlet mindenképpen gyanús). A felsorolt faktorok együttes használatával valósítható meg a Multifactor Authentication, amely faktorai bármelyek lehetnek az előbbiek közül, akár jelszó használata nélkül is megvalósíthatók.

A hitelesítés megvalósítható egyébként *Sinle Sign-On (SSO)*, vagyis egyszeri bejelentkezés használatával is, amely olyan felhasználói hitelesítési szolgáltatást jelent, amellyel a felhasználó egyetlen bejelentkezési azonosítóval több alkalmazáshoz is hozzáférhet különböző technológiák segítségével, mint az *LDAP*, az *OAuth* és *SAML*. Sok helyre bejelentkezhetünk például a Google, vagy a Facebook fiók segítségével.

A *Federation* vagyis szövetségek alkalmazásával – amely lehetővé teszi az identitások megosztását és használatát több információs rendszerben vagy szervezetben – a felhasználók egyetlen hitelesítő adatpárral férhetnek hozzá különböző rendszerekhez.

Az *Authorization*, vagyis engedélykezelés a *Privileged Access Management (PAM)* felhasználásával valósítható meg. Kiváltság szerinti hozzáférési szinteket kell elkülöníteni és kezelni, a magas jogosultságú (rendszergazdai) fiókokat just-in-time engedélyekkel, jelszó páncéltermekkel és ideiglenes fiókok használatával kell védeni. A PAM megvalósítása hozzáférés-szabályozási modellek segítségével történik (*Access Control Models*), amelyek lehetnek a kötelező hozzáférés-szabályozás (*Mandatory Access Control*), a diszkrecionális hozzáférés-szabályozás (*Discretionaly Access Control*), a szerepkör-alapú hozzáférés-szabályozás (*Role-based Access Control*), a szabályalapú hozzáférés szabályozás (*Rule-based Access Control*), és az attribútumalapú hozzáférés-szabályozás (*Attribute-based Access Control*). Megtehetjük továbbá azt is, hogy a hozzáférést napszak szerint korlátozzuk, olyan módon, hogy a legkisebb jogosultság megvalósításának koncepciójának megfeleljen, ami szerint mindenkinek a lehető legkisebb jogosultságot adjuk meg, ami a munkája elvégzéséhez szükséges. Előbbi szempontok alapos mérlegelése után, azok alapján osztjuk ki a jogosultságokat.

3.4.1. A mesterséges intelligencia felhasználása a hozzáférés kezelés területén

Az IAM rendszerek megerősítésében a mesterséges intelligencia kiemelkedően fontos szerepet játszik a *Viselkedési Analitikában (UEBA)*. A gépi tanulási algoritmusok meghatározzák a felhasználók és entitások normál viselkedésének módjait, majd azonnal azonosítják az attól eltérő anomáliákat, amelyek belső fenyegetésekre vagy feltört hitelesítő adatokra utalhatnak.

A hagyományos jelszóalapú módszereken túlmutatóan az MI erősíti a hitelesítési mechanizmusokat is. Az MI-alapú többfaktoros hitelesítés (MFA) viselkedési biometriát alkalmaz, mint például a billentyűzetdinamika, lehetővé téve a folyamatos hitelesítést a munkamenet során. Folyamatosan felméri a felhasználó kockázati profilját, és dinamikus kockázati pontszámot (trust score) rendel a hozzáférési kérelemhez, ami alapján adaptív MFA döntések születhetnek.

Végül az MI lehetővé teszi a *Dinamikus Hozzáférés-szabályozást (DAC)*, amely a statikus jogosultságokkal szemben valós időben, a környezet (context) alapján adaptálja az engedélyeket. Ez a rugalmas megközelítés kulcsfontosságú a *Zero Trust*

modell megvalósításában, valamint növeli az automatizált fiókéletciklus-kezelés (*provisioning és deprovisioning*) hatékonyságát és biztonságát.

3.5. Cryptographic Solutions - Kriptográfiai Megoldások

A kriptográfia (*Cryptography*) a kódok írásának és megfejtésének gyakorlatát jelenti, annak érdekében, hogy elrejtsük az információ valódi jelentését [31 68. videó, 0:06]. A titkosítás egyik leggyakoribb formája. A titkosítás (*Encryption*) az a folyamat, amelynek során a közönséges információt (a *plaintextet*) értelmezhetetlen formátumú (*chipertext*) adatokba alakítjuk át, vagyis egy kulcs segítségével kódoljuk. Az eredeti információ visszanyerése a kulcs segítségével történik, ez a kulcs a titkosítás legfontosabb összetevője, ezt kell védeni. A titkosítás az adatok minden, korábban tárgyalt állapotában (*Data at rest, Data in transit és Data in Use*) fontos, adatainkat mindenhol védeni kell. A titkosítás során, a kódolás folyamata egy algoritmus segítségével történik, amellyel átalakítjuk *chipertext* formába a normál szöveget, és a dekódolás során vissza *plaintext formába* a kódolt szöveget. Az algoritmus többnyire nyilvános matematikai függvényt jelent, amely szakértők által létrehozott és folyamatosan tesztelt titkosítási folyamatnak a leírása (mivel nyilvános, ebből következően a kulcsot kell szigorúan védeni, és gyakran cserélni).

Mint a jelszavak esetében, itt is, a kulcs annál erősebb, minél hosszabb, emiatt lesz ellenálló a *Brute Force* támadásokkal szemben a titkosított szöveg (a 128 bites kulcs már biztonságosnak tekinthető, de a 256 bites sokkal biztonságosabb, és létezik 512 bites is, ezeknél a biztonság szintje exponenciális mértékben növekszik). A kulcs védelme érdekében ajánlott viszonylag gyakran cserélni, mert akármilyen erős is ez, egy idő után a fejlődő technológia miatt sérülékennyé válhat (a magasabb számítási kapacitású gépek sokkal hamarabb oldják meg az összetett matematikai problémákat). Továbbá biztonságos hardvermodulokban kell tárolni a kulcsokat, nyugalmi állapotban titkosítani kell, és biztonságosan kell továbbítani, amikor használják ezeket, valamint folyamatosan ellenőrzi, hogy csak a megfelelő jogosultsággal rendelkezők férjenek hozzá. A kriptográfia értelmezéséhez tehát az algoritmusokat és a kulcsokat kell részletesebben megvizsgálni.

A titkosítási algoritmusok lehetnek szimmetrikusak vagy aszimmetrikusak. A szimmetrikus titkosítási algoritmusok a titkosításhoz és a visszafejtéshez is ugyanazt a kulcsot használják, az aszimmetrikus titkosítási algoritmusok pedig egy nyilvános és egy titkos kulcsot tartalmazó kulcspárt használnak az adatok titkosításához és visszafejtéséhez.

A szimmetrikus algoritmusokat nevezik *Private Key Encryption* (magánkulcsos) algoritmusoknak is mivel a küldőnek és a fogadónak is ugyanazt a közös titkos kulcsot kell ismernie, ez a *private key*. Ez olyan, mint a fizikai zárak a házunk bejárati ajtaján, ugyanis minden családtagunknak ugyanolyan kulcsra van szüksége, hogy ki tudja ezt nyitni, ez a megosztott titkos kulcsnak a megfelelője (*shared secret key*). Ezzel a megoldással viszont nem teljesül a Non-repudiation, vagyis a letagadhatatlanság alapelve, mivel mindenki, aki rendelkezik a megosztott kulccsal, hozzáférhet a védett tartalomhoz, vagy a példánkban a házban lévő tárgyakhoz. Ebből következik a megosztott kulcs másik problémája is, hogy ha azt akarjuk, hogy minél több felhasználó hozzáférjen a titkos kulcshoz és azzal a tartalomhoz, azt egyre több emberrel kell megosztanunk, Olyan ez, mintha a Wifi jelszavunkat osztanánk meg minden egyes vendégünkkel, aki járt nálunk. Egy idő után túl sokan fogják ismerni azt, még azok is, akiknek már nem kellene hozzáférnie, és minél többen ismernek egy titkot, az már annál kevésbé titok, a védelme egyre nehezebb.

Az aszimmetrikus algoritmusok nem igényelnek megosztott titkos kulcsot, ezért *Public Key*, vagyis nyilvános kulcsú algoritmusoknak is nevezik. Két külön kulcsot alkalmaznak, az egyiket az adatok titkosítására, a másikat pedig a visszafejtésre. Ilyen algoritmusok a *Diffie-Hellman, az RSA (Ron Rivest, Adi Shamir, and Leonard Adleman*), és *az ECC (Elliptic Curve Cryptography* – elliptikus görbén alapuló kriptográfia).

A szimmetrikus és aszimmetrikus algoritmusokat eltérő céllal tervezték. A szimmetrikus algoritmusok népszerűek, mert sokkal gyorsabbak a hasonló biztonságot garantáló aszimmetrikusnál. Az Aszimmetrikus algoritmusok viszont megoldják a megosztott kulcs korábban tárgyalt problémáját, tehát mindkettőnek van helye gyakorlati megvalósításokban, sőt kombinálhatók is hibrid megoldásokban. Ilyen lehet, ha a kulcsmegosztás problémájának megoldására aszimmetrikus (nyilvános kulcsú) titkosítást használunk, majd az így kódolt megosztott titkos kulcs segítségével aztán

szimmetrikusan titkosítjuk az adatokat, amely megoldással gyorsabb adatátvitelt tudunk elérni.

A titkosítási algoritmusok csoportosíthatók matematikai algoritmusuk szerint is, ekkor beszélhetünk *Stream Cipher* (folyamrejtjel) és *Block Cipher* (blokk rejtjrel) algoritmusokról. A Stream Cipher algoritmusok esetén egyszerre csak egyetlen biten vagy bájton történik meg a titkosítási művelet, és egy kulcsfolyam-generátort használnak egy XOR-függvénnyel összekevert bitfolyam létrehozásához. Ez a megoldás nagyon jó a valós idejű tartalmak, hang vagy videó streamelésére. A Stream Cipher algoritmusok általában szimmetrikus algoritmusok, tehát ugyanazt a megosztott kulcsot használják titkosításhoz és visszafejtéshez. Többnyire hardveres megoldásokban használják.

A Block Cipher algoritmusok a bemenetet fix hosszúságú adatblokkokra bontják, mielőtt a titkosítási funkciókat végrehajtják. Az adott blokk mérete fix (általában 64, 128, vagy 256 bites), ha a titkosítandó adatunk kisebb méretű ennél, kitöltést alkalmaz a titkosítás végrehajtása előtt. Ezek a megoldások könnyebben beállíthatók és megvalósíthatók, és kevésbé érzékenyek a biztonsági problémákra, és szoftveres megoldásokkal is könnyebben végrehajthatók, mint a Stream Cipher alkalmazásai.

Szimmetrikus algoritmusok a *DES* (*Data Encryption Standard* – 64 bites kulcs, ebből 8 bit paritás, tehát hatékony kulcshossza 56 bit – 64 bites blokkok, majd 16 fordulóban transposition és substitution), a *3DES* (DES gyengeségei miatt módosított változata 3 különböző 64 bites - 56 bit hasznos – kulcsot használ, a DES 3-szor), az *IDEA* (*International Data Encryption Algorithm*, 64 bites szimmetrikus blokkos titkosítás, 128 bites kulcsmérettel, nem olyan elterjedt), *AES* (*Advanced Encryption Standard*, 128 bites, 192 bites vagy 256 bites kulccsal, *Rijndael-algoritmus*nak is hívják, tulajdonképpen szabvánnyá vált), *Blowfish* (64 bites blokkok, 32 bites és 448 bites kulccsal, nem túl gyakori, nincs szabadalmaztatva, nyílt forráskódú), *Twofish* (128 bites blokkok, 128 bites, 192 bites vagy 256 bites kulcsokkal, nincs szabadalmaztatva, nyílt forráskódú) *és Rivest Ciphers* azon belül *RC4, RC5, RC6* (*RC Cipher Suite,* Ron Rivest hozta létre, 6 algoritmus RC néven – RC4 folyamkódolású 40 bittől 2048 bitig változó kulcsmérettel, SSL és WEP használja – RC5 blokkos, akár 2048 bites kulccsal, RC6 az RC5 erősebb és fejlettebb változata). Az RC4 (szimmetrikus folyam-alapú algoritmus) kivételével mindegyik szimmetrikus blokkos kódolású algoritmus, és a legerősebb és leggyakrabban használt közülük az AES.

Az aszimmetrikus algoritmusok *Public Key*, tehát nyilvános kulcsú algoritmusok, kulcsuk szabadon és nyíltan hozzáférhető. A titkosításkor használt másik kulcs a *Private Key*, a saját kulcs. Az adatok titkosságának biztosítása érdekében azokat a címzett egyedi Public Key, vagyis nyilvános kulcsával kell titkosítani. Mivel a hozzá tartozó Private Key, vagyis az egyedi titkos kulcs csak a címzettnek van meg, csak egyedül ő lesz képes visszafejteni az információt (így lényegében egyirányú titkosítás). A *Non-repudiation*, vagyis letagadhatatlanság érdekében a feladó titkosítja a levelet a titkos kulcsával (ettől ez még nyilvános marad, mert a nyilvános kulcs párja bárkinek elérhető), de letagadhatatlanul bizonyítja, hogy a feladótól származik az adat, mivel az egyedi titkos kulcsával van titkosítva, amit csak ő ismer. Ahhoz, hogy a sértetlenséget is biztosíthassuk, a küldött üzenet alapján létrehozunk egy *hash digest* értéket, amelyet a feladó privát kulcsával kell titkosítani. Ez a digitális aláírás, és bizonyítja, hogy a levél a feladótól származik, mivel csak ő ismeri a titkos kulcsát, amivel titkosítani tudta azt. Ezután az üzenetet titkosítjuk a címzett nyilvános kulcsával, mivel a titkos kulcspár csak a címzettnek van meg, amivel visszafejtheti a levelet.

Az aszimmetrikus algoritmusok közül a Diffie-Helmannt kulcscserére használják (IPSec protokoll részeként VPN alagút létrehozásakor, vagy bármilyen kulcscsere alkalmával). Az RSA-t széles körben használják kulcscsere-titkosításra és digitális aláírásra, amely 1024 és 4096 bit közötti kulcsméreteket támogat. Az ECC-t mobileszközökben használják, igazából 6-szor hatékonyabb, mint az RSA azonos kulcsméreten, ezért alkalmazzák kisebb teljesítményű eszközökön (fajtái az *ECDH – Elliptic Curve Diffie-Hellman, ECDHE – Elliptic Curve Diffie-Helman Ephemeral, ECDSA – Elliptic Curve Digital Signature Algorithm*).

A *Hashing*, vagyis hashelés egy egyirányú kriptográfiai függvény, amely egy bemenetből egy egyedi „kivonatot” állít elő. Egyirányú, mert a kapott hash értékből nem fejthető vissza az eredeti érték, ez az eredeti fájl digitális ujjlenyomataként érthető. A hash érték minden esetben ugyanolyan hosszú lesz, függetlenül a bemeneti szöveg hosszúságától. Az ezt elvégző algoritmusok az *MD5*, az *SHA* család, a *RIPEMD*, és a *HMAC*.

Az MD5 algoritmus az egyik leggyakoribb, 128 bites hash-értéket állít elő. Mivel csak 128 bit, csak korlátozott számú hash értéket tud létrehozni, ami hash-ütközéshez vezethet (két különböző bevitt értéknek, fájlnak, előfordulhat, hogy azonos hash-értéke lesz).

Az *SHA* (*Secure Hash Algoritm*) családba beletartozik a *SHA-1* család (160 bites kivonat), az *SHA-2* család (hosszabb hash kivonat, *SHA-224, SHA-256, SHA-384, SHA-512*, nevük mutatja a hash bit értékét, 64-80 közötti számítási körrel), és az *SHA-3* (ugyanaz, mint a SHA-2, csak 120 számítási kört alkalmaz a hash-érték előállításához).

A *RIPEMD* (*RACE Integrity Primitive Evaluation Message Digest*) 160, 256, és 320 bites hash-értékeket állít elő, és nyílt forráskódú kódolási algoritmus, de nem vált olyan népszerűvé, mint a SHA család.

Az *HMAC* (*Hash-based Message Authentication Code*) a hash- alapú üzenethitelesítési kódot jelent, az üzenetek integritásának ellenőrzésére használják. Más algoritmusokkal párosítják a használat során (*HMAC-MD5, HMAC-SHA1, HMAC-SHA256* attól függően, milyen alapértelmezett hasht használnak az *HMAC*-hez).

A hash függvények használatának egyik leggyakoribb módja tehát a digitális aláírás létrehozása (vagyis a titkosítandó szövegből készítünk egy hash-értéket, és a saját privát kulccsal titkosítjuk, ennek értékét az üzenethez csatoljuk). A digitális aláírások gyakorlati alkalmazásához pedig egy algoritmusra van szükség, ami lehet *DSA* (*Digital Security Algorithm*), *RSA*, vagy *ECC*. Másik felhasználási terület a kódaláírás, amelynél a fejlesztők hozzáadják digitális aláírásukat a programhoz vagy fájlhoz (telepítőfájlhoz), például a Google Play Áruházban való feltöltéskor is, mobilalkalmazás fejlesztésekor, ez biztosítja, hogy a telepítőfájlt nem módosították.

A hash érték előállítása általánosan használt módszer a jelszavak titkosításakor is.

4. Vulnerabilities and Attacks – Sebezhetőségek és támadások [23][28][29][30][31][32]

A sebezhetőségek (*Vulnerabilities*) az alkalmazás konfigurációjának vagy folyamatainak (vagy a számítógépes rendszer, hálózat hardverének, szoftverének) olyan gyengeségeit vagy hibáit jelentik, amelyeket rosszindulatú szervezetek kihasználhatnak. A kihasználás módja lehet illetéktelen hozzáférés (*Unauthorized Access*), az adatok megsértése *(Data Breaches*), a rendszer megzavarása (*System Disruptions*), és a kompromittálódás más formái, amik veszélyeztetik a biztonságot és hozzáférhetőséget.

A sebezhetőségek eredhetnek szoftverhibákból, amelyeknek több forrása is lehet. Az *Error* olyan hiba, amelyet valaki a szoftver gyártása közben követ el a tervezés, programozás, telepítés vagy konfigurálás során. A *Fault* a kódon belüli hiba, amely biztonsági problémához *bughoz* vezet. A *Defect* a követelményektől való eltérés a *Fault* miatt, tehát valami nem úgy működik, ahogy kellene, vagy nem felel meg a minőségi követelményeknek. A *Failure* pedig az a meghibásodás, ami akkor jelentkezik, miután a szoftvert kiadják az ügyfeleknek. A hibák elkülönítése az eredetük alapján – vagyis, hogy a fejlesztés melyik fázisában fordulnak elő – sokat segíthet a hatékony tervezésben.

A támadások (*Attacks*) olyan szándékos cselekmények, amelyekkel a támadók a számítógépes rendszer, hálózat vagy alkalmazás sebezhetőségeit használják ki. Ha nincs sebezhetőség, nem lehetséges támadás sem, ezért a sebezhetőségeket és az azokat kihasználó támadási módokat indokolt egy fejezetben tárgyalni. A támadási módok az illetéktelen hozzáférés (*Unauthorized Access*), az adatlopás (*Data Theft*), a rosszindulatú szoftverekkel fertőzés (*Malware Infections*), a szolgáltatásmegtagadási támadások (*Denial of Service Attacks* - DOS), a közösségi befolyásolás (*Social engineering*) lehetnek, és minden egyéb tevékenység, melynek célja a digitális eszközeink bizalmas jellegének, integritásának vagy rendelkezésre állásának veszélyeztetése.

4.1. Sebezhetőségek csoportosítása

A sebezhetőségek különböző szinteken jelenhetnek meg. A hardveres sebezhetőségek (*Hardware Vulnerabilities*) közé tartoznak a system firmware sebezhetőségek, melyek forrása lehet régi, életciklus végi, illetve örökölt (*legacy*) hardverek. Problémát okoznak még a már nem támogatott rendszerek (unsupported system), a javításokra szoruló rendszerekről hiányzó javítások (*missing patches*), és a félrekonfigurált eszközök. A szoftverfejlesztés során ezek adottak, de számolni kell velük. A javításuk a rendszer keményítésével (*Hardening the System*), foltozással (*Patching*), alapkonfigurációk érvényesítésével, és a régi, nem biztonságos eszközök leszerelésével, illetve az eszközök elszigetelésének vagy szegmentálásának létrehozásával valósítható meg. Ide tartoznak a szerverek, workstationök, laptopok, switchek, routerek, hálózati eszközök, mobileszközök és IoT eszközök sebezhetőségei.

Másik kategória a *Bluetooth* sebezhetőségei, amelyek gyűjtőneveiről már volt szó. Ide tartozik a *Bluesnarfing, Bluejacking, Bluebugging*, *Bluesmark*, és a *Blueborne* sebezhetőségek.

A *mobil sebezhetőségek* (*Mobile Vulnerabilities*) és támadások az oldalletöltés (*Side loading*), a *Jailbraking*, és az ismeretlen forráshoz való nem biztonságos csatlakozási módszerek (*Insecure Connection Methods*). A megelőzés és a veszély csökkentésének módszerei a javításkezelés (*Patch Management*), a mobileszköz-kezelési megoldások (*Mobile Device Management Solutions*) és az eszközök oldalletöltésének és rootolásának megakadályozása (*Preventing Sideloading and Rooting of Devices*).

A *0. napi sebezhetőségek* (*Zero-day Vulnerabilities*) olyan típusú szoftver vagy hardver sebezhetőségek, amelyet rosszindulatú aktorok fedeznek fel és használnak ki, mielőtt a hiba nyilvánosságra kerülne és a rendszer gyártójának vagy fejlesztőjének lehetősége lenne kijavítani azokat.

Az *operációs rendszerek sebezhetőségei* (*Operating System Vulnerabilities*) a foltozatlan rendszerek (*Unpatched system*), a *Zero-Day,* vagyis 0. napi sebezhetőségek, a félrekonfigurálások (*Misconfiguration*), az adatszivárgás (*Data Exfiltration*) és a rosszindulatú frissítések (*Malicious Updates*). Ezek ellen a foltozással

(*Patching*), a konfigurációkezeléssel (*Configuration management*), a nyugalmi adatok titkosításával (*Encryption of Data*), végpontvédelem telepítésével (*Installing Endpoint Protection*), host-alapú tűzfalak használatával (*Utilizing Host-based Firewalls*), host-alapú IPS-ek bevezetésével (*Implementing Host-Based IPS*), hozzáférés szabályozás és engedélyek konfigurálásával (*Configuring Access Controls and Permissions*) és alkalmazások engedélyezési listájának használati előírásával (Requiring the Use of Application Allow Lists) lehet védekezni.

Bár a fentiek közül nem mindegyik kapcsolódik szorosan a szoftverfejlesztéshez, megemlítésük indokolt.

4.2. Cryptographic Attacks – Kriptográfiai támadások

A kriptográfiai támadások többnyire jelszavak megszerzésére irányulnak. Abban különböznek a jelszavak elleni támadásoktól, hogy nem akarják megszerezni az eredeti cipher text jelszót, hanem a hash érték segítségével követnek el támadásokat.

A *Pass the Hash Attack* elkövetése során a támadó megszerzi a felhasználó jelszavának hash értékét, és azzal hitelesíti magát a felhasználó nevében. Ebből következik, hogy a megszerzett hash-érték egyenértékű a plain text jelszóval. Tovább rontja a helyzetet, hogy a hash-ek begyűjtése automatizálható is, például a Mimikatz segítségével. Ez ellen úgy lehet védekezni, hogy megfelelően konfigurált és patchelt eszközöket használunk, folyamatosan frissítve az operációs rendszert.

A *Birthday Attack* hash-érték ütközés előállításával valósul meg (két különböző jelszónak ugyanaz a hash értéke, azért Birthday Attack, mert a dátumból csak hónapot és napot tekintve elég nagy az esélye, hogy két embernek azonos születésnapja legyen, mivel csak 365 napból áll egy év). Ebből következik, hogy egy másik jelszó is megfelelő a belépéshez, ha ugyanaz a hash-értéke mint az eredetinek.

Érdemesebb tehát nagyobb hash-értékű algoritmusokat használni mert így kisebb az ütközés veszélye. Emellett alkalmazható még védekezésre a *Key Stretching* technika is, amellyel a rövidebb kulcsot egy újabb algoritmus segítségével hosszabbra nyújtják. A másik technika a *Salting*, vagyis sózás, melynek során a jelszóhoz véletlenszerű karaktereket adunk hozzá, mielőtt hashelnénk, így két azonos jelszó hash-értéke is

eltérő lehet, mivel eltér a hozzáadott „sójuk”. A harmadik eszköz a *Nonce*, ami egyszer használatos számot jelent, egy egyedi, gyakran véletlenszerű számot, amelyet a jelszó alapú hitelesítési folyamathoz adnak hozzá, ezzel minden egyes alkalommal megváltoztatva a hash-értéket a bejelentkezés során. Továbbá még fontos a belépési lehetőségek korlátozása is, ami meghatározott számú sikertelen belépési kísérlet után ideiglenesen zárolja a belépés lehetőségét.

4.3. Password attacks – jelszavak elleni támadások

A *Password attacks*, vagyis a jelszavak feltörése céljából elkövetett támadások többféle elkövetési módot jelentenek. A *Brute Force Attacks,* vagyis a nyers erővel végrehajtott támadások során a karakterek minden lehetséges kombinációját kipróbálják, ameddig meg nem találják a helyes jelszót. Nagyon alapos, de időigényes és számításigényes támadási mód. Az ilyen támadásokat a jelszó összetettségének növelésével (nagyobb számú karakterkészlet), a jelszó hosszának növelésével, a bejelentkezési kísérletek korlátozásával, multifaktoros hitelesítéssel és *Captha* használatával lehet megnehezíteni, az online jelszófeltörési kísérletek megakadályozására.

A *Dictionary Attacks*, vagyis szótárak alkalmazásával elkövetett támadások egy listát vagy szótárat alkalmaznak, amely a leggyakrabban használt jelszavakat és azok hash-értékét tartalmazza. Ezeket behelyettesítéssel sorban kipróbálják, és ha egyezést találnak, meg is van a hozzáférés. Gyorsabb támadási mód a Brute Force Attacksnál, mivel a szótár jóval kevesebb szót tartalmaz, mint amennyi lehetséges kombinációja van a karakterkészletnek. A Dictionary Attacks szótára nagyon hasznos lehet a gyakori és nem túl összetett szavakat használó felhasználók esetén, jelentősen lerövidíti a támadás elkövetésének idejét, viszont haszontalan az egyedi jelszavak és a nem szótári szavakat tartalmazó, véletlenszerű karakterek egymásutánjából felépülő jelszavak esetén. Egyébként az elkerülés érdekében ugyanazokat a módszereket kell alkalmazni, mint a Brute Force Attacks esetében.

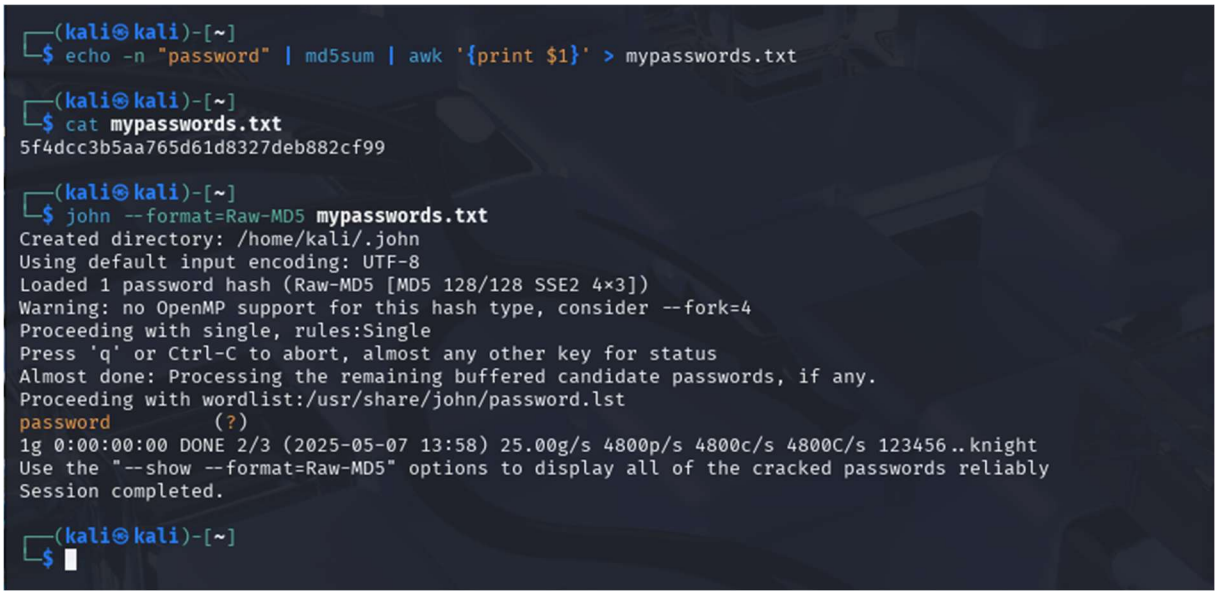
A *Password Spraying,* vagyis jelszószóró támadások a Brute Force Attacks egyik formája, amely során kis számú, nagyon gyakran használt jelszót próbálnak ki nagyszámú felhasználó vagy fiók ellen, például egy sok felhasználói fiókot tartalmazó

cég esetén. Megpróbálja a gyenge jelszavakat behelyettesíteni, és elegendő, ha csupán egy fiókhoz sikerül hozzáférést szerezni. Mivel kis számú próbálkozásra van szükség egy fiók esetén, így kivédhető a sikertelen próbálkozásokat követő fiókzárolás is, ha van ilyen a támadott felhasználói fiók esetén. És mivel sok fiók érintett, elég nagy az esélye annak, hogy valamelyik felhasználó gyenge és könnyen kitalálható jelszót használ (ha az alkalmazás, vagy szervezet biztonsági irányelve nem követeli meg az erős jelszót és annak időközönkénti cseréjét). A védekezés itt is megegyezik a Brute Force Attacks esetében tárgyaltakkal.

Végül a hibrid támadások (*Hybrid Attacks*) egyesítik a nyers erővel és szótárral végrehajtott támadások elemeit, mivel gyakran használt jelszavakból álló szótárból indulnak ki, de speciális karaktereket tartalmazó variációkat is tartalmaznak. Ez szerepelhet a szótárban, ekkor szótár alapú támadásról van szó, vagy a jelszófeltörő szoftver dinamikusan létrehozhatja ezeket. Például gyakran adnak a felhasználók ugyanolyan jelszavakat, azzal a különbséggel, hogy egy számjegyet vagy egyéb megkülönböztető karaktert írnak a jelszó végére. A dinamikus jelszófeltörő szoftver ezeket próbálgatják (például növekvő számokat illesztenek be gyakran használt jelszavakhoz).

A következő példa a John The Ripper vagyis John nevű szoftver segítségével végrehajtott jelszófeltörést mutatja be, egy gyenge titkosítással (MD5) ellátott, gyenge jelszó esetében (az MD5 nem ajánlott jelszavak hash értékeinek előállítására, sebezhetősége miatt ellenőrző összegek kiszámítására javallott, arra viszont nagyon jó, mivel akár egyetlen karakter megváltoztatása is az eredeti szövegben, teljesen más hash-értéket eredményez).

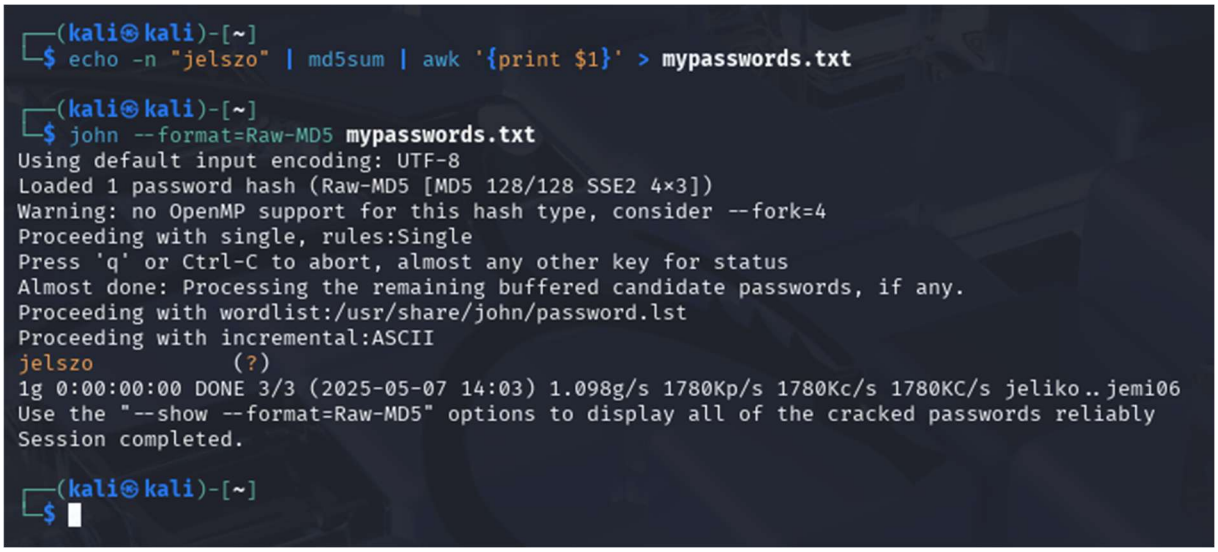
Az ábrán a „password” jelszót titkosítjuk md5 algoritmussal, majd a hash-értéket fájlba írjuk (mypasswords.txt). Ezt követően a John segítségével megszerezzük az jelszó plaintext értékét (egyéb jelszó-helyreállító alkalmazás még a *Hashcat*, amelyet offline jelszó-alapú támadások végrehajtására használnak, és képes felhasználni a CPU mellett a GPU teljesítményét is, a *Metasploit Framework*nek is vannak segédmoduljai a jelszótöréshez – *Auxilary Modules*, a *Hydra* is többszálas online jelszófeltörő eszköz, az *Ncrack* és a *Medusa* is hasonlók).



1. ábra - saját Kali Linux John teszt [31]

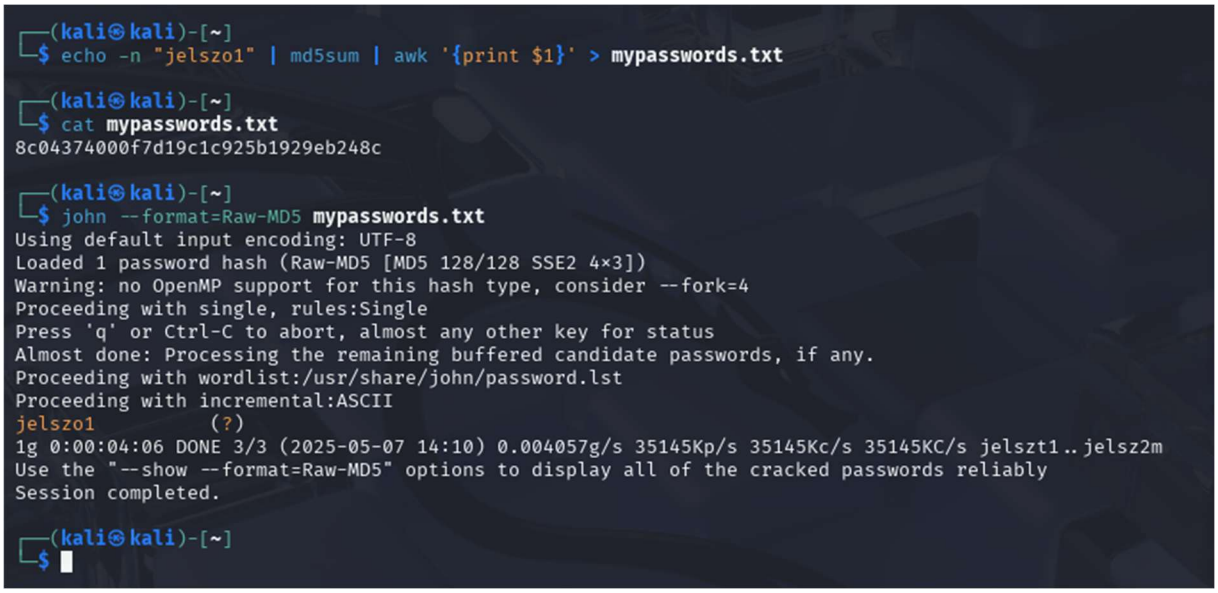
Amint a képen látható, azonnal megtalálta a jelszót a „password” jelszó esetében, a szükséges idő 00:00, tehát 0 másodperc. Az eredeti szöveg, a megtalált jelszó a kép alján narancssárga színnel van jelezve. Valószínűleg ez a jelszó szerepelhetett a John által használt szótárban, mivel nem írja az alábbi ábrán látható helyettesítési eszközkészletet (Proceeding with Incremental : ASCII).

A következő kísérlet a „jelszo” kitalálására irányul:



2. ábra - saját Kali Linux John teszt [31]

Amint látható a „jelszo” megtalálása sem volt nehezebb feladat, szintén 0 másodpercet vett igénybe. Minimálisan bonyolítva, a „jelszo1” esetében már kicsit más a helyzet:



3. ábra – saját Kali Linux John teszt [31]

A nagyon kicsit bonyolultabb, „jelszo1” megadott jelszóval a Brute Force támadás végrehajtásához 4 perc és 6 másodpercre volt szükség egy virtuális Kali Linuxban (2 processzormaggal és 4 GB RAM felhasználásával egy AMD Ryzen 7 4800H processzorral működő laptopban, érdemes belegondolni mennyi időre lehet szükség egy teljesértékű, a számítógépen egyedül bootolt Kali Linux esetén). A korábban tárgyalt szempontok alapján történő jelszóválasztás láthatóan nagyon fontos, mert exponenciálisan növeli a jelszótörés idejét.

4.4. Code Injection

Többféle kódinjektálási támadás létezik, közös bennük, hogy a támadó mindegyikkel további információt vagy kódot illeszt be egy bemeneti csatornán keresztül egy webalkalmazásba. A *Malicious Code Injection Attacks* általános típusa, gyűjtőneve a kódinjektálási támadásoknak.

4.4.1. SQL Injection

Az SQL Injection támadás *a Structured Query Language*, vagyis *SQL*, magyarul a strukturált lekérdezési nyelv utasításai segítségével történik. Négy alapvető utasításcsoport a SELECT, amellyel adott adat beolvasása történik az adatbázisból, az INSERT, amellyel beszúrás történik az adatbázisba, a DELETE, amely adott adatot eltávolít az adatbázisból, és UPDATE, amely az adott adatot felülírja az adatbázisban. A támadás végrehajtása is ezek segítségével történik.

Az SQL Injection olyan típusú kibertámadás, amely a webes alkalmazások vagy adatbázisok sebezhetőségeit használja ki. A támadó rosszindulatú inputot illeszt be a felhasználói bemenetekbe vagy lekérdezésekbe, és ezeken keresztül manipulálja az adatbázist, illetve futtatja a nem kívánt, elrejtett SQL parancsokat. Ezeket az adatbáziskezelő feldolgozza és végrehajtja, melynek természetesen a jogos felhasználó szempontjából nem kívánt következményei lesznek.

Ha szeretnénk bejelentkezni egy weboldalra, webalkalmazásba, bejelentkezéskor az általunk beírt felhasználónevet és jelszót elküldi a rendszer (remélhetőleg titkosítva) az adatbázisba, és lekérdezi, hogy a két érték megegyezik-e az ott tároltakkal. Ez történhet SQL utasítással például:

> SELECT * FROM Users WHERE user_id = ‘KGergely’ AND password = ‘Pass123’; [30 258. oldal] [31]

Ha ezek egyeznek, megkapjuk a hozzáférést. Ez a normál működési mód.

Az SQL injection ennek egy speciális típusa, egy olyan támadási forma, amely SQL-lekérdezés befecskendezéséből áll a beviteli űrlapon keresztül, amelyet a támadó arra használ, hogy adatokat küldjön egy webes alkalmazásnak (ebből következően webes környezetben fordul elő). A támadó megpróbál paramétereket vagy kódot beilleszteni az SQL utasításba, amelyet az adatbázis lekérdezésére használnak, ezt pedig URL paraméterként teszi, egy űrlapba való beírással, a cookie-k módosításával, POST adatok megváltoztatásával, vagy http-fejléc segítségével. Ez a tevékenység automatizálható is, léteznek eszközök, amelyek az SQL-Injection-re és kihasználására összpontosítanak.

Az SQL Injection-re való sérülékenység kihasználása történhet tehát űrlapmezőben, ahol a felhasználónév mezőbe beírjuk annak a létező felhasználónak a nevét, akivel be akarunk lépni, például: KGergely. Támadóként nem ismerem ennek a felhasználónak a jelszavát, ezért a jelszó mezőbe beírom:

```
‘OR 1=1;
```

Ennek hatására az űrlap adatai bekerülnek a backend SQL utasításában, majd lekérdezik az adatbázist. A teljes lekérdező parancs ez lesz:

> SELECT * FROM Users WHERE userID = 'KGergely' AND password = ''OR 1=1; [30 258. oldal] [31]

Az SQL Injection támadásra sérülékeny weboldalak esetén a hozzáférés engedélyezett lesz. Az injektált utasításban az ’ escape karakter, ezután további utasításokat adhatunk SQL nyelven az adatbázisnak. Amit ezt követően beírtunk, kiértékeli, hogy egyezik-e a megtámadott felhasználó jelszavával. Nem tudjuk a felhasználó jelszavát, amit beírtunk, nem egyezik azzal. Viszont az SQL-ben a vagy (OR) beírásával egy másik kiértékelési ágat adunk hozzá. Vagy a jelszónak kell tehát egyeznie, vagy a másik ág feltételének kell igaznak lennie. A másik ág az 1=1-gyel. 1 minden esetben egyenlő 1-gyel, tehát ha nem tudjuk a jelszót, a feltétel második része akkor is igaz, így igaz értékű lesz a lekérdezés, hozzáférünk az adatbázishoz.

Az SQL Injection megakadályozására nagy felelőssége van a programozónak: bemeneti érvényesítést kell alkalmaznia (*Input Validation*), amellyel kizár bizonyos karaktereket a jelszó mezőből. Meghatározhatja a jelszó minimális és maximális hosszát is, korlátozhatja az engedélyezett adattípusokat (decimális számok, pénznem), szűkítheti az adattartományt tehát különféle korlátozások bevezetésére van szükség, hogy ezzel gátolja meg az Injection végrehajtását. Nem elég a frontend oldalon megtenni, szükséges a backend oldali bevitel érvényesítés is. Fontos, hogy a felhasználótól származó adatokat „fertőtlenítsük” (*Sanitize data*), ez a veszélyes karakterek eltávolítását jelenti. Célszerű ezt egy middleware-ben elvégezni, vagy használhatunk webes tűzfalakat az ügyfél és webkiszolgáló közé helyezve (*web*

application firewall). (A National Institute of Standards and Technology 800-88 néven létrehozott egy dokumentumot, amelyre a média fertőtlenítésére vonatkozó iránymutatásként hivatkoznak). A *Sanitization* az adatok különböző technikákkal történő hozzáférhetetlenné és visszaállíthatatlanná tételét jelenti, esetünkben az injektálandó veszélyes kódrészlet eltávolítását jelenti.

4.4.2. NoSQL Injection

Míg a relációs rendszerek táblázatos adatokat használnak, a *NoSQL* adatbázisok, mint például a *MongoDB*, rugalmas, *JSON-szerű* dokumentummodellt alkalmaznak, ami azt jelenti, hogy a támadás végrehajtásának technikái eltérnek, de a cél, a jogosulatlan hozzáférés és adatszivárgás, ugyanaz marad.

Sikeres NoSQL Injection támadás eredménye lehet érzékeny adatok, például jelszavak és személyes információk jogosulatlan kiszivárgása, vagy akár a szerver feletti irányítás átvétele. Egy ilyen támadás adatvesztéshez, adatsérüléshez vagy szolgáltatásmegtagadási (DoS) állapothoz vezethet. A védekezés elsődleges célja az, hogy a programozó elválassza a megbízhatatlan adatokat a parancsoktól és lekérdezésektől. Ezt úgynevezett paraméterezett lekérdezésekkel (vagy azok NoSQL megfelelőivel) lehet elérni, amelyek a felhasználói bemenetet adatokként kezelik, soha nem kódként, ezzel lehetetlenné téve a legtöbb injektálási támadást.

4.4.3. XML injection:

Az *Extensible Markup Language (XML)*, vagyis bővíthető jelölőnyelv, egy emberek által is olvasható, széles körben felhasználható jelölőnyelv, amely a *HTML* (pontosabban az *SGML*) alapjaira épül. A HTML-hez hasonló címkéket (tageket) használ az adatok elkülönítésére, a különbség az, hogy míg a HTML-nél kötött címkék vannak, az XML esetében bármilyen előfordulhat. Lényeges kötöttség, hogy minden egyes nyitó címkét le kell zárni egy zárócímkével, amelyben a címkenévnek teljesen meg kell egyezni (kis- és nagybetű érzékeny), kivéve, hogy a zárócímke elé / jelet kell írni.

```
<?xml version="1.0" encoding="UTF-8"?>
<konyvek>
    <konyv kID="K0001" nyID="NY0001">
      <szerzo>Neil Bradley</szerzo>
      <cim>Az XML-kézikönyv</cim>
      <kiado>Szak kiadó</kiado>
      <megjelenEv>2000</megjelenEv>
      <kategoria>Adatkezelés</kategoria>
      <kategoria>Web</kategoria>
      <kategoria>XML</kategoria>
    </konyv>
</konyvek>
```

Az XML-t a webalkalmazások hitelesítésre, engedélyezésre vagy más típusú adatcserére használják. Az XML-ben tárolt adatok az ügyféltől a kiszolgálóhoz, vagy egyik kiszolgálótól a másikhoz kerülnek. Az XML-adatok szállítás közbeni védelme érdekében azokat mindig titkosított alagúton belül kell küldeni pl. *TLS Tunnel* használatával.

Az *XML Injection* olyan biztonsági sebezhetőség, amely az XML-adatokat vagy bővíthető jelölőnyelvi adatokat feldolgozó webes alkalmazásokat célozza. *XPath injection*nek is nevezik, mert az XML útbejárásra, csomópontjelölésre vonatkozó XPath szabványát használja fel. A támadó a támadás során manipulálhatja az XML-bemeneteket, mellyel kihasználja az alkalmazás XML-elemző vagy feldolgozó mechanizmusainak sebezhetőségeit. Ha ez sikerül, jogosulatlan hozzáféréshez vezet, így a támadó megismeri a tárolt adatokat, vagy más rosszindulatú tevékenységet végezhet a megtámadott webes alkalmazáson belül.

Az elkerülésére használt módszerek, a titkosítás mellett, a kiszolgáló védelme érdekében az *Input Validation*, és az *Input Sanitization*. Ez a leghatékonyabb eszköz mindenféle kódinjektálási támadás elkerülésére. Ha titkosítás vagy bemeneti érvényesítés nélkül küldünk XML-adatokat, akkor azok sebezhetőek lesznek a *Snooping* (szimatolás), a *Spoofing* (hamisítás), a Request forgery (kéréshamisítás), és az *Injection of arbitary code* (tetszőleges kódbevitel) támadásokra.

Az XML sérülékenységeinek kihasználásai (*XML exploitok*) az *XML Bomb* és *XML External Entity (XXE)* támadások. Az *XML Bomb (Billion Laughs Attack)* az XML

körkörös hivatkozási lehetőségeit használja ki, entitásokat kódol és azokat exponenciális méretűvé bővíti, ezzel nagy mennyiségű memóriát köt le az eszközben, melynek hatására az akár össze is omolhat. Célja szolgáltatásmegtagadási támadás elkövetése, ezért is hívják bombának. Ha egy támadó hozzáfér a szerverhez, és fel tudja dolgozni az alább látható XML fájlt, azzal elkezdi lekötni, fogyasztani az szerverünk erőforrásait. A támadás másik neve, a *Billion Laughs Attack*, arra utal, hogy az entitás hivatkozások faktoriális jellege miatt egymilliárd „lol” entitást hoz létre, amellyel akár 3 GB-nyi memóriát is foglalhat.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```
[31 172. videó 10:09]

A másik XML sérülékenységet kihasználó támadási mód az *XML External Entity (XXE)* támadás, amely egy helyi erőforrás iránti kérés beágyazására tesz kísérletet. Az alább látható példakódban az XML fájl az XML fájlhivatkozási megoldását használja ki, ami egyébként egy DTD fájlhivatkozásra van kitalálva. Normál esetben a DTD-ben megkötéseket adhatunk az XML fájl tartalmára és adattípusaira vonatkozóan, itt viszont egy Linux-gép shadow fájlját próbálja beolvasni, amely a rendszer fiókjainak password hash-eit tartalmazza.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM "file:///etc/shadow">
]>
<foo>&xxe;</foo>
```
[31 172. videó 11:10]

Bármikor, amikor a felhasználó bármilyen adatot visz be az alkalmazásunkba, legyen az URL, fájl, mező bevitele egy weboldalon, vagy valamilyen XML-adat átküldése, sok biztonsági problémát megelőzhetünk már azzal, ha csak bemeneti érvényesítést használunk. Ez az egyik legfontosabb védekezési mód az SQL és XML Injection típusú támadások elkerülésére, emellett pedig titkosítás és ellenőrző összegek használata tovább javíthat az alkalmazás biztonságán.

4.5. Cross-site scripting (XSS)

A *Cross-site scripting* olyan webes biztonsági sebezhetőséget jelent, mely során a támadó egy rosszindulatú scriptet juttat be egy olyan weboldalra, amelyet más felhasználók néznek. A rosszindulatú script egy támadó webhelyen található, vagy egy olyan linkbe van kódolva, amelyet egy megbízható webhelyre juttatnak be, ezzel veszélyeztetik a weboldalt böngésző felhasználót. A támadó így megkerülheti a böngésző biztonsági megoldásait, vagy a megbízható zónákat, és ezzel ráveheti a felhasználót, hogy a rosszindulatú scriptet futtassa, amelyet a háttérben a böngésző végrehajt. Az alábbi link mutatja a Cross-site scripting linkbe ágyazott scriptjét.

```
https://www.kovacsgergely.hu/search?q=<script%20type='application/javascript'>alert('xss')</script>
```

A példa linkben először az URL-címet alakítjuk ki úgy, hogy a szkriptet hozzáadjuk egy biztonságos weboldalhoz, azon belül a ’search?q=’ részhez, amelyet általában egy weboldalon keresésre, lekérdezésre használnak. Ezt követően a lekérdezést kicseréljük a beilleszteni kívánt scriptre.

Az előbbi példában bemutatott károkozási kísérlet a reflektált, vagy *Non-Persistent XSS*, ahol a támadás (és a script futtatása) akkor következik be, ha a felhasználó rákattint a linkre, vagy valamilyen módon a link tartalma beíródik a webböngészőbe és a felhasználó Entert nyom, jóváhagyja. Ha a script futtatás megtörténik, egyszer történik meg, aztán leáll, tehát nem tartós támadásról van szó. (A Cross-site scripting végrehajtása egyébként legálisan és szabadon kipróbálható az xss-game.appspot.com oldalon.

Az ilyen jellegű támadások fő megelőzési módja itt is a bemeneti érvényesítés (*Input Validation)*. Minden esetben nagyon oda kell figyelnünk arra, hogy milyen bemeneti információt fogadunk el a felhasználóinktól, sok biztonsági probléma forrása lehet, ha erre nem fordítunk megfelelő figyelmet.

A Cross-site Scripting támadásnak létezik egy másik változata is, a *Persistent XSS*, amely az előzőtől eltérően egy tartós Cross-site scripting támadás. Itt a támadó célja, hogy kódot illesszen be az adott megbízható webhely által használt backend adatbázisba. Ha ez sikerül, akkor a támadónak már nem kell megvárnia, hogy valaki rákattintson az általa elhelyezett linkre, mivel a rosszindulatú kódja már be van ágyazva a weboldal adatbázisába, ki tudja használni azt. Ennél a támadási módnál bármikor, amikor a felhasználó betölti az oldalt vagy a tartalmat az adatbázisból, a tartós cross-site scriptet is betölti, tehát minden alkalommal lefut.

A Persistent és a Non-Persistent XSS is a kiszolgáló oldalát célozza, tehát backend oldali támadásnak tekinthető, mivel a szerver az, amely végrehajtja a bejutatott scripteket.

Létezik viszont egy kliens-oldali XSS támadás is, amelyet *DOM cross-site scripting* támadásnak nevezünk. Ebben az esetben a *DOM (Document Object Model*) az ügyfél webböngészőjét használja ki a kliensoldali scriptek segítségével a weboldal tartalmának és elrendezésének módosítására. A DOM egy platform- és nyelvfüggetlen standard programozói interfész, amely a HTML, XHTML, XML, valamint rokon formátumaiknak a szerkezetét és az objektumaikkal történő interakciókat modellezi [24].

A DOM dokumentumfát épít a weboldal alkotóelemeiből, amelyek egymással szülő-gyermek kapcsolatban álló objektumok rendszerét alkotják. A dokumentum tartalmát, illetve a dokumentum valamennyi összetevőjét magában foglalja és a módosítás

eredménye mindig visszahat a böngészők esetén a megjelenített oldalra. Másképpen fogalmazva a DOM az, ahogyan a dolgok megjelennek a kliens böngészőjében, így a scriptek DOM-ba történő beillesztésével ez ténylegesen megváltoztatható.

Az alábbi példában a linkbe ágyazott script hatására az ’alert’ felugró ablak a dokumentum cookie-jait mutatja meg. Ezzel a megoldással a webböngésző DOM-ján belül próbálunk meg hozzáférni a cookie-tárolóhoz, ezért ezt a böngészőben próbáljuk végrehajtani, és megjeleníteni az ügyfél cookie-jainak tartalmát. Ha a webböngésző sebezhető erre, akkor a jelen példában a felugró ablak dokumentum cookie-kat fog megjeleníteni (*document.cookie*).

```
https://kovacsgergely.hu/index.html#default<script>alert(document.cookie)</script>
```

Az előbbi példa alapján lehetséges document.write alkalmazása is, amely módosíthatunk, a document.location, amely helyadatokat láthatunk, és egyéb olyan dolgok, amelyek hozzáférhetnek a DOM környezethez és megváltoztathatják azt. Amikor DOM XSS-t hajtunk végre, akkor az adott helyi rendszer bejelentkezett felhasználói engedélyeivel hajtjuk végre a támadás, tehát ha a bejelentkezett felhasználó rendszergazda jogosultságokkal futtatja a cross-site scripting támadást, máris rendszergazda hozzáférése lesz a támadónak a rendszerhez.

4.6. Session Hijacking – Munkamenet eltérítés

A *Session/Cookie/Session Key Hijacking* fogalmak a munkamenet eltérítés nevű támadásra utalnak, melynek során a támadó átveszi a felhasználói munkamenetet, ezzel egy érvényes számítógépes munkamenetet kihasználva jogosulatlan hozzáférést szerezhet a számítógépes rendszerben lévő információkhoz vagy szolgáltatásokhoz.

A *Session Management*, vagyis a munkamenetkezelés, alapvető biztonsági összetevője a webes alkalmazásoknak. Lehetővé teszi a webes alkalmazások számára, hogy egy felhasználót egyedileg azonosítsanak több különböző művelet és kérés során, közben megőrizve az adott felhasználó által generált adatok állapotát, és azt továbbra is az adott felhasználóhoz rendelve.

A *Cookie* tulajdonképpen egy szöveges fájl, amely a felhasználóról szóló információkat tárolja, amikor a felhasználó meglátogatja a webhelyet. Akkor jön létre, amikor a kérésre a kiszolgáló először küldi el a HTTP-válasz fejlécet, ezzel együtt küldi a *cookie*-t is. Ezt követően az ügyfél minden egyes további kérése tartalmazza ezt a cookie-t, a legfrisebb információkkal kiegészítve. Azért van erre szükség, mert a HTTP állapotmentes (stateless) protokoll, ami azt jelenti, hogy a szerver alapértelmezés szerint semmilyen információt nem őriz meg a kliensről. Az információk tárolásához szükség van cookie-ra, amelyet a kliens gépe fog tárolni (vagy másik megoldás az információk tárolására az adatbázisba való mentésük is).

A cookiek lehetnek tartósak (*Persistent*), ezek nevükből adódóan megmaradnak a böngésző gyorsítótárában, ameddig a felhasználó nem törli őket, vagy le nem járnak. Tartósságukból adódóan fontos a titkosításuk és védelmük, mert érzékeny információkat is tartalmazhatnak. A cookie-k másik fajtája a nem tartós (*Non-Persistent*) cookie-k, amelyek a memóriában találhatók, és nagyon rövid ideig használjuk őket. Adott munkamenetre vonatkoznak, amikor a böngésző befejezi ezt, a munkamenet cookie törlődik.

A Session Hijacking, vagyis munkamenet eltérítés nevű támadás, a *spoofing attack,* vagyis hamisítási támadás típusú támadások egy olyan fajtája, mely során a támadó leválasztja az állomás kapcsolatot, majd az eredeti állomás IP-jének meghamisításával vagy más átvételi mechanizmus használatával a saját gépével helyettesíti azt. Történhet még a cookie-k ellopásával és módosításával is, ha sikerül ellopni a munkamenet cookie-t, akkor a korábban már hitelesített felhasználó nevében átveheti a munkamenetet a támadó.

Ha egy weboldal a munkamenetet adatbázisban kezeli, amely adatbázisban minden felhasználóhoz hozzárendel egy véletlenszerű munkamenet-tokent, még mindig támadható lesz, ha a véletlenszerű séma nem igazán véletlenszerű. Ekkor olyan tokenek jöhetnek létre, amelyeket egy támadó könnyen kitalálhat. Ha ez sikerül, a támadó előre megjósolhatja a munkamenetet, ezzel szintén átveheti a már hitelesített munkamenetet.

A *Session Prediction* nevű munkamenet előrejelzési támadások szintén a hamisítási támadások fajtái, ahol a támadó megpróbálja megjósolni a munkamenet-tokent, hogy eltérítse a munkamenetet. Megelőzésük érdekében a munkamenet-tokeneket valóban

véletlenszerűvé kell tenni, nem kiszámítható algoritmussal kell generálni. Ezzel elérhető, hogy a munkamenet-tokeneket ne lehessen könnyen kitalálni. Fontos követelmény még a tokenekkel szemben, hogy ezek semmilyen információt ne fedjenek fel az ügyféllel kapcsolatban (pont erre lettek kitalálva), és egyszer használatos jegyek legyenek, amelyek csak az adott munkamenet időtartamára jönnek létre.

4.7. Cross-site request forgery (XSRF)

A Session hijacking támadást egy lépéssel tovább viszi a *Cross-Site Request Forgery (XSRF)* támadás elkövetője, aki arra összpontosít, hogy megpróbálja becsapni a felhasználót, hogy az tudtán kívül, beleegyezése nélkül hajtson végre egy műveletet egy másik weboldalon, ezzel nem szándékos, káros műveletet végrehajtását érje el a felhasználó nevében az adott weboldalon. Ennek megfelelően a *Cross-Site Request Forgery (XSRF)*, vagyis webhelyközi kéréshamisítás egy olyan rosszindulatú scrript, amelyet a támadó webhelyén tárolnak, és amely felhasználható egy másik webhelyen indított munkamenet kihasználására ugyanazon a böngészőn belül.

A végrehajtás érdekében a támadónak meg kell győznie áldozatát, hogy indítson munkamenetet a célzott weboldalon. Ha ez megtörtént, a támadó átadhat egy HTTP-kérést az áldozat böngészőjének, és ezt a céloldalon végzett műveletnek álcázhatja. Például, ha a felhasználó már bejelentkezett a fiókjába, a támadó megpróbálhatja egy cross-site request segítségével megváltoztatni a felhasználó jelszavát vagy e-mail címét.

A Cross-site Request Forgery sokféleképpen álcázható, a támadók használhatnak kódolási technikákat, mint a kép, címkék és más HTML-kódolási technikák, hogy elrejtsék magukat, és anélkül is megvalósítható, hogy az áldozatnak rá kellene kattintania egy linkre. Viszont ahhoz, hogy az XSRF megtörténhessen, a weboldalnak rendelkeznie kell egy olyan funkcióval, amely jogosulatlan hozzáféréshez vezethet (például egy „elfelejtettem a jelszavam” funkcióval). További feltétel még, hogy a webhelynek a felhasználók hitelesítéséhez cookiek-ra kell támaszkodnia, és kiszámítható, kitalálható mintákra kell támaszkodnia a mukamenet-kezeléshez. Ezek

elkerülése nagyon fontos az XSRF támadással kapcsolatos sérülékenység megakadályozásához.

Egy XSRF támadás például úgy történhet, hogy a felhasználó a böngészőjében már hitelesítette magát a céloldalon (mondjuk a banki felületén). Ezután a támadó megpróbálja ellopni az érvényes munkamenet-tokeneket az áldozat böngészőjéből (esetleg olyan módon, hogy a felhasználó webböngészőjében két lap van nyitva, az egyik a támadó weboldala, a másik a felhasználó bankja). Ha a felhasználó már bejelentkezett a bankjához, és egyúttal a támadó weboldalához is csatlakozott az új lapon (mert a támadó adathalász módszerekkel vagy más social engineering módszer felhasználásával rávette a felhasználót, hogy rákattintson egy linkre), a támadó már elérheti a célját. Egy XSRF támadással megpróbálhatja manipulálni a munkamenetet a felhasználó bankjával, mivel a böngészője már hitelesített az adott webhelyen, így megpróbálhatja átvenni a munkamenetet a másik lapon belül.

A cross-site request forgery megelőzéséhez biztosítani kell, hogy minden űrlapbeadásnál felhasználó-specifikus tokeneket használjon az alkalmazás. Véletlenszerűség hozzáadásával tovább növelhető a biztonság, illetve a jelszóváltoztató funkció megvalósításánál további, felhasználóval kapcsolatos információk kérése is indokolt. Például az alkalmazás megkövetelheti a felhasználótól, hogy jelszóváltoztatás alkalmával adja meg az aktuális jelszavát (ez megállítja az XSRF támadások nagy részét, mert kulcsfontosságú a felhasználó tudja nélkül megváltoztatni a felhasználó jelszavát a fiók átvétele céljából. További biztonsági intézkedés lehet kétfaktoros hitelesítés alkalmazása.

Összefoglalva tehát, ha valaki megpróbálja rávenni az áldozatot, hogy akaratlanul végezzen el egy műveletet egy weboldalon, akkor ez általában a cross-site request forgery egy formája. Ez leggyakrabban úgy történik, hogy az áldozatot valamilyen ismeretlen frissítésre próbálják rávenni az alapértelmezett e-mail címükön, vagy a felhasználó jelszavának megváltoztatásával.

4.8. Buffer Overflow – Puffer túlcsordulás

A *Buffer Overflow* akkor következik be, amikor egy programban egy processz a megfelelő tartományon kívül tárol adatokat, több adatot ír egy memóriapufferbe, mint amennyit az elbír.

Az adatlopások nagy részénél a Buffer Overflow-t kihasználó támadások a kezdeti támadási vektorok. Ha egy támadó megpróbál túl sok adatot bevinni a stackbe (verembe), vagy megváltoztatja a visszatérési mutató értékeit, akkor támadást hajthat végre. Ekkor megpróbálja felülírni a pointer visszatérési címét, hogy az egy másik helyre mutasson, ahol a támadó elhelyezheti a rosszindulatú kódját. Ehhez fel kell tölteni a puffert NOP (non-operation instruction), vagyis nem-műveleti utasítással, hogy eltolja a pointer értékét. Ez a NOP csúszás, és lehetővé teszi a rosszindulatú kód számára a szomszédos memória felülírását, majd tetszőleges parancsok végrehajtását vagy a program összeomlását.

A Buffer Overflow támadások ellen használható Address Space Layout Randomization (ASLR – a címtartomány elrendezésének véletlenszerűsége). Ez egy programozási technika, amely segít megakadályozni, hogy a nem rosszindulatú programkód mutatója hova mutat (egyébként ez is kijátszható). A Buffer Overflow támadást nehéz végrehajtani, komoly technikai tudás kell hozzá.

Sok programozási nyelvnél nem mi kezeljük a pointert, maga a nyelv megoldja helyettünk. Viszont, ha olyan programnyelvvel dolgozunk, ahol ez ránk van bízva, fontos figyelni a Buffer Overflow elkerülésére, helyesen kezelve a pointert.

4.9. Race Conditions - Versenyfeltételek

A *Race Condition (versenyhelyzet)* egy szoftveres sebezhetőség, amely párhuzamos rendszerekben jön létre, amikor több folyamat egyszerre próbál hozzáférni egy megosztott erőforráshoz. A végrehajtás időzítése miatt kiszámíthatatlan eredmények születnek, ami lehetővé teszi a támadó számára, hogy a jogos műveletek közé ékelje be a rosszindulatú tevékenységét. Ennek egyik kiaknázási módja volt a *Dirty COW* Linux sebezhetőség, amely jogosultsági szint emeléshez vezetett.

A támadások gyakran az *ellenőrzés (Time-of-Check)* és a *használat (Time-of-Use)* időpontja közötti rövid időablakot használják ki (pl. banki átutalásoknál), de előfordul a *kiértékelési idő (Time-of-Evaluation)* alatti manipuláció is. A sebezhetőség azért veszélyes, mert nehezen észlelhető, mivel gyakran a normál naplózáson kívül zajlik.

A védekezés a *mutexek (Mutual Exclusion, kölcsönös kizárás)* és az *erőforrás zárolások (use locks)* alkalmazásával történik, amelyek biztosítják, hogy egyszerre csak egy folyamat férhessen hozzá a kritikus erőforráshoz. A zárolások bevezetése ugyanakkor magában hordozza a deadlock (holtpont) kialakulásának kockázatát, ezért a többszálú rendszerek tervezése során kulcsfontosságú a gondos tesztelés és tervezés.

5. Applicaton security

Az alkalmazásbiztonság témakörében a legfontosabb kérdés a biztonságos alkalmazások építése, és hogy hogyan valósítható ez meg. Hogyan előzhetők meg a biztonsági sebezhetőségek, hogyan észlelhetők és javíthatók. Sok kérdés már korábban is előkerült a témával kapcsolatban, ez a fejezet újra összeszedi és kiegészíti ezeket a kérdéseket, szemléltetve a problémás kérdéseket a felhasználóknak, figyelemfelhívásként pedig a fejlesztőknek.

A biztonsági eszközök szemléltetéséhez készítettem [33, 34 alapján] egy kicsi webalkalmazást, amely elérhető a GitHub-on [5]. A futtatásához szükséges technikai részleteket a readme.md fájl tartalmazza a repository gyökerében. Egy egyszerű alkalmazásról van szó, amelyben a felhasználó bejelentkezhet, kijelentkezhet és regisztrálhat, ha még nincsen fiókja. A bejelentkezett felhasználók megjeleníthetik a már rögzített könyveiket, amelyeket csak ők láthatnak, felhasználókhoz kötöttek.

Az alkalmazás tárgyalásának menetét az application security legfontosabb területei adják, így következzenek ezek.

5.1. Az alkalmazásbiztonság kulcsfontosságú területei [23][25][26][27][28][29][30][31]

A modern digitális környezetben a szoftverek és szolgáltatások minőségének biztosítása érdekében az *alkalmazásbiztonság (Application Security, AppSec)* alapvető fontosságúvá vált. Célja az üzleti integritás és a felhasználói adatok védelme. Az alkalmazásbiztonság magában foglalja a biztonságos szoftverek tervezését, megépítését és karbantartását, kiterjesztve a webalkalmazásokra, a mobilalkalmazásokra, az API-kra, valamint az IoT eszközökre, függetlenül attól, hogy helyszíni (on-prem) vagy felhőalapú (in-cloud) környezetben futnak.

Az alkalmazásbiztonság nem pusztán technológiai kihívás, hanem elsősorban humán és folyamat alapú probléma. Caroline Wong [28 9. oldal] szerint az alkalmazásbiztonsági tevékenységek négy fő kategóriába sorolhatók: *irányítás (governance)*, *biztonsági problémák megtalálása*, *javítása* és *megelőzése*. Kritikus fontosságú, hogy a fejlesztőket partnerként kezeljék a biztonsági stratégia kialakításában.

A Biztonságos Szoftverfejlesztési Életciklus (SSDLC), a hatékony alkalmazásbiztonsági program megköveteli a "*Shift Everywhere*" (*Sean Poris: Bug Bounty – Shift Everywhere*) [28 161. oldal] elv érvényesítését. Ez a szoftvertesztelés területéről is ismert „*Shift Left*” elv, vagyis a fejlesztési folyamatban való előretolás kiterjesztése, ami azt jelenti, hogy a biztonsági intézkedéseket integrálni kell az SDLC, vagyis szoftverfejlesztés minden szakaszába, a tervezéstől a karbantartásig. Ez a folyamat a *Biztonságos Szoftverfejlesztési Életciklus (SSDLC)*, amelynek kulcselemei a *fenyegetésmodellezés (Threat Modeling)*, a *kódellenőrzés*, a *tesztelés* és a *sebezhetőségkezelés*. A továbbiakban megnézzük a biztonságos szoftver építésének összetevőit.

5.1.1. Bemeneti adatok érvényesítése (Input Validation)

A bemeneti adatok érvényesítéséről már volt szó. Ez az egyik legfontosabb, az alapvető védelmi vonal, amely biztosítja, hogy az alkalmazások kizárólag jól definiált és biztonságos adatokat dolgozzanak fel. Ez a folyamat egyfajta minőség-ellenőrzésként működik: minden beérkező információt ellenőriz, hogy az érvényes,

biztonságos és helyesen formázott-e. Az előre definiált érvényesítési szabályok határozzák meg az elfogadható és elfogadhatatlan bemeneteket.

A validációt jellemzően a folyamat korai szakaszában, például a felhasználói felületen (front-end validáció) végzik. Ez viszont megkerülhető, a *többrétegű védelem (defense-in-depth)* elve alapján mindenképp szükség van a szerveroldali (back-end) ellenőrzésre is. Ezzel hatékony védelmet érhetünk el az olyan gyakori támadások ellen, mint az SQL injection, a Cross-site scripting (XSS) vagy a puffertúlcsordulás (buffer overflow).

5.1.2. Sütik (Cookies) biztonságos kezelése

A sütik kis adatcsomagok, amelyeket a webböngészők tárolnak a felhasználó gépén, hogy *állapottartó (stateful)* információkat őrizzenek meg a szerver és a kliens közötti kommunikáció során. Mivel érzékeny adatokat, például munkamenet-azonosítókat tartalmazhatnak, biztonságos kezelésük kiemelt fontosságú. A *biztonságos sütiket (Secure Cookies)* kizárólag titkosított *HTTPS* kapcsolaton keresztül lehet továbbítani.

Az ajánlott gyakorlatok közé tartozik az *állandó sütik (persistent cookies)* használatának kerülése a *munkamenet-ellenőrzéshez*, valamint a *Secure*, *HttpOnly* és *SameSite* attribútumok megfelelő beállítása, amelyekkel megakadályozható a sütikhez való illetéktelen hozzáférés.

5.1.3. Technikai alapelvek és védekezési mechanizmusok

A *fenyegetésmodellezés* már a tervezési fázisban segít felmérni az alkalmazás által hordozott üzleti kockázatokat.

A *kódellenőrzés* biztosítja, hogy a változtatásokat még a telepítés előtt áttekintsék a lehetséges sebezhetőségek szempontjából. Az alkalmazásbiztonság teszteléséhez számos eszköz áll rendelkezésre.

Statikus kódelemzés (Static Code Analysis - SAST), egy olyan hibakeresési módszer, amely az alkalmazás forráskódját még a futtatás előtt vizsgálja át. Ez a "fehér dobozos" tesztelési technika automatizáltan képes azonosítani olyan potenciális sebezhetőségeket, mint a buffer overflow, az SQL injection és az XSS. A statikus

kódelemzés különösen hatékony a megfelelő bemeneti érvényesítés biztosításának ellenőrzésében mind a front-end, mind a back-end kódban.

A *Dinamikus kódelemezés (DAST)* az alkalmazásokat futás közben elemzi a külső támadó szemszögéből. Ez a "fekete dobozos" megközelítés nem igényel hozzáférést a forráskódhoz. Két gyakori DAST módszer a a *Fuzz Testing* és a *Stress Testing*. A *Fuzzing (Fuzz Testing)* során a rendszerbe nagy mennyiségű véletlenszerű vagy hibás adatot táplálnak azzal a céllal, hogy összeomlásokat vagy váratlan viselkedést provokáljanak. Ez a technika segít feltárni a rejtett biztonsági hibákat és gyengeségeket. A *Stress Testing (Terheléses tesztelés)* módszerrel a rendszer stabilitását és megbízhatóságát értékeljük extrém terhelési körülmények között. Célja a szűk keresztmetszetek azonosítása és a rendszer helyreállítási képességének felmérése egy esetleges túlterheléses támadás után.

A *szoftverösszetétel-elemzés (SCA)* kiemelten fontos a modern alkalmazásoknál, mivel figyeli a harmadik féltől származó függőségek biztonságát és frissítéseit, amelyek a teljes kód 20–40%-át tehetik ki. Ezeket kiegészítik a védelmi réteg eszközei, mint a *Web Application Firewall (WAF)* és a *Runtime Application Security Protection (RASP)*, amelyek segíthetnek a kódinjektálási támadások elkerülésében

5.1.4. Kódaláírás (Code Signing)

A kódaláírás egy digitális aláírási eljárás, amely megerősíti a szoftver készítőjének identitását és garantálja a kód integritását. Az aláírás biztosítja a felhasználót arról, hogy a szoftvert nem módosították vagy hamisították meg a kiadása óta. Fontos viszont, hogy a kódaláírás önmagában nem garantálja a sebezhetőségek hiányát, csupán a kód hitelességét igazolja.

5.1.5. Homokozó (Sandboxing) használata

A homokozó egy biztonsági mechanizmus, amely a futó programokat egy elkülönített (izolált) környezetben tartja, korlátozva azok hozzáférését a kritikus rendszererőforrásokhoz, például a fájlrendszerhez vagy a hálózathoz. Ezzel megakadályozza, hogy egy potenciálisan káros program kárt tegyen a gazdaeszközön

vagy a hálózaton. A sandboxing ideális megoldás nem megbízható vagy nem tesztelt programok biztonságos futtatására.

5.1.6. Az AppSec kiemelt kockázati területei

A *Kockázatszámítás (Risk Calculation)* nagyon fontos, nem elegendő az automatizált eszközök eredményeire hagyatkozni, figyelembe kell venni az alkalmazás jellegét, az általa kezelt adatok érzékenységét és az üzleti kontextust.

Az *Ellátási Lánc Biztonsága (Supply Chain)* szintén fontos kérdés, mivel a külső szállítók és az open source függőségek (például npm csomagok esetén) jelentős biztonsági kockázatot jelentenek, mivel backdoorokat vagy szándékos hibákat tartalmazhatnak. A CVE és CWE listák segítenek a gyengeségek azonosításában. A *CVE (Common Vulnerabilities and Exposures)* egy nyilvános lista, amely konkrét, felfedezett kiberbiztonsági sérülékenységek egyedi azonosítóit és leírását tartalmazza, lehetővé téve a beazonosítást és a javítást.

A *CWE (Common Weakness Enumeration)* ezzel szemben a szoftverek és hardverek tervezési és kódolási hibáinak általános típusaiból álló katalógus, amely a sérülékenységek gyökérokaira összpontosít, és a megelőzéshez nyújt útmutatást.

Röviden: CVE = Konkrét hiba (tünet), CWE = Hibatípus (gyökérok).

A *Titkos Adatok Kezelése (Secret Management)* is fontos, a jelszavakat, kapcsolati stringeket, hash-eket és API-kulcsokat szigorúan tilos közvetlenül a kódban tárolni.

5.1.7. Mesterséges intelligencia az alkalmazásbizonság területén

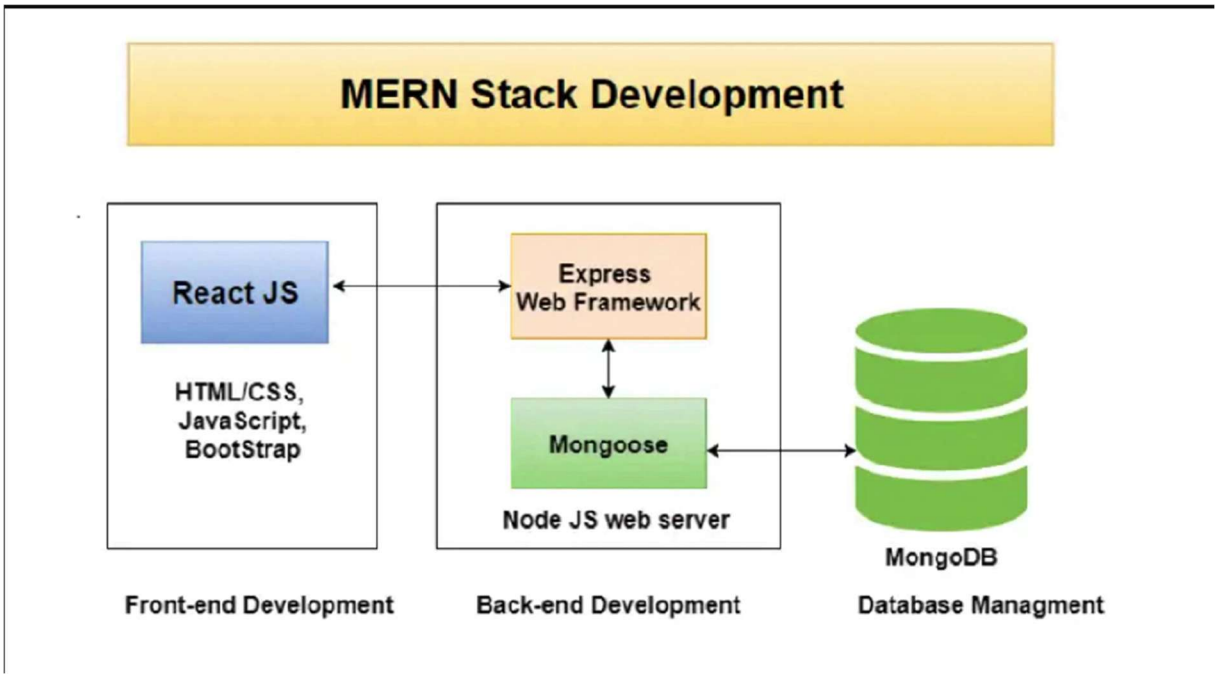
Az AppSec területén a mesterséges intelligencia lehetővé teszi a biztonsági tevékenységek korai fázisba helyezését, az úgynevezett "*Shift Left*" megközelítést, segítve ezzel a széttöredezett munkafolyamatok kezelését. Az *LLM-ek (Large Language Model)* az intelligens kód elemzés révén új paradigmát hoznak a *statikus biztonsági tesztelésbe (SAST)*. Képesek kontextuális szemantikát és programozási idiómákat felismerni, ezzel azonosítva a hagyományos, szabályalapú rendszerek számára észrevehetetlen bonyolult biztonsági réseket.

Az MI-vezérelt *dinamikus elemzés (DAST)* automatizálja a teszteset-generálást, kiterjesztve a lefedettséget. Továbbá az LLM-ek forradalmasítják a kódjavítást is azáltal, hogy *intelligens biztonsági javításokat (patching)* generálnak, mélyrehatóan megértve a kódstruktúrát és a sebezhetőségeket, figyelembe véve a kontextuális tényezőket. A GenAI eszközök növelik a kódolás sebességét, de óvatosságra van szükség a generált kódban esetlegesen rejlő biztonsági hibák miatt.

Az MI a tervezési fázisban is segít a dinamikus fenyegetésmodellezésben az alkalmazás architektúrájának és konfigurációjának multimodális elemzése alapján.

5.2. A Bookstore alkalmazás [5][33][34]

Ez a projekt a *MERN-stack*re épül, amely a *MongoDB* adatbázisból, az *Express backend* keretrendszerből, a *React* frontend keretrendszerből és a *Node.js* JavaScript futtatókörnyezetből áll.



4. ábra – [33]

Az *Express*en keresztül a MongoDB adatbázist a *Mongoose* segítségével érjük el. A Mongoose egy *Object Data Mapper (ODM)*, amely számos könnyen használható függvénnyel egyszerűsíti az adatbázishoz való csatlakozást és a különböző műveletek elvégzését, és néhány biztonsági megoldást is tartalmaz (később).

Az alkalmazás a GitHub-ról történő klónozás és a szükséges backend és frontend függőségek npm (*Node Package Manager*) csomagkezelővel történő telepítése után (`npm install` paranccsal a gyökérmappában és a frontend mappában) az alkalmazás az

```
npm run start
```

paranccsal indítható az alkalmazás gyökeréből. Ekkor a frontend és a backend is elindul együtt.

A konfigurációs beállításokat, amelyeket védenünk kell, egy külön fájlban, egy .env fájlban helyezetem el, szintén az alkalmazás gyökerében (ezt hozzá kell adni a

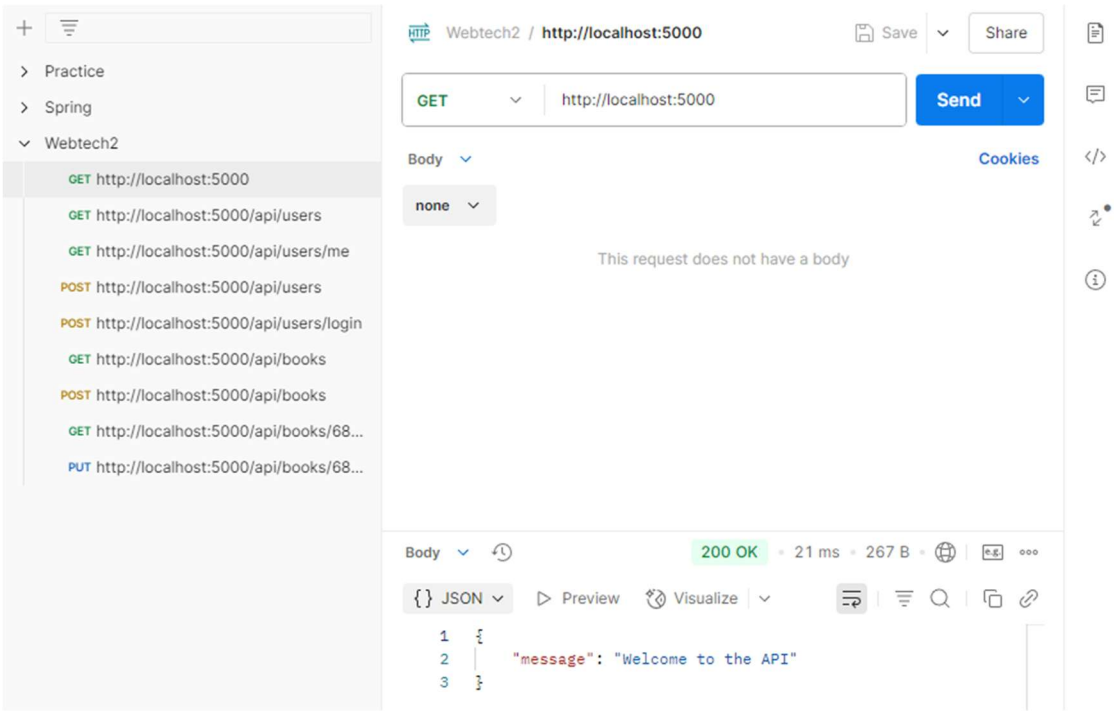
.gitignore fájl tartalmához, hogy a verziókövető rendszerbe ne kerüljenek be az érzékeny adatok).

5. ábra [5]

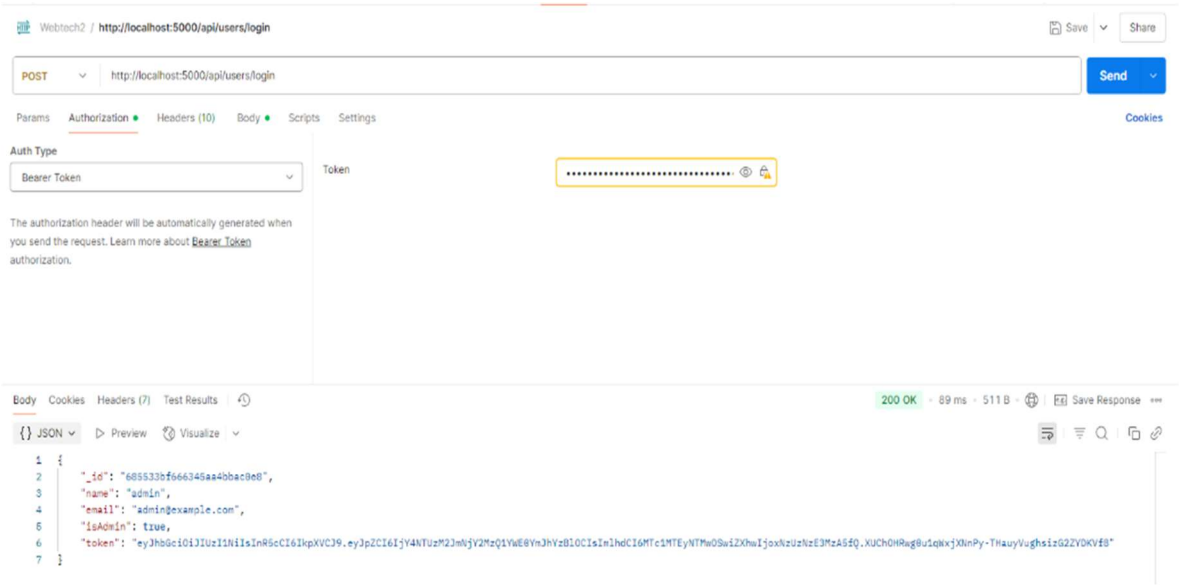
Látható, hogy ez tárolja a port számát, ahol a backend fut, a MongoDB adatbázishoz való csatlakozáshoz szükséges linket, és a Jason Webtoken munkamenetkezeléséhez szükséges titkot.

A MongoDB-t a Mongoose-on keresztül érjük el, ez egy Object Data Mapper sok könnyen használható függvénnyel, ami segít az adatbázishoz való csatlakozásban, és a műveletek elvégzésében, az Expressen keresztül.

A Postman segítségével történt a backend tesztelése.



6. ábra [5]

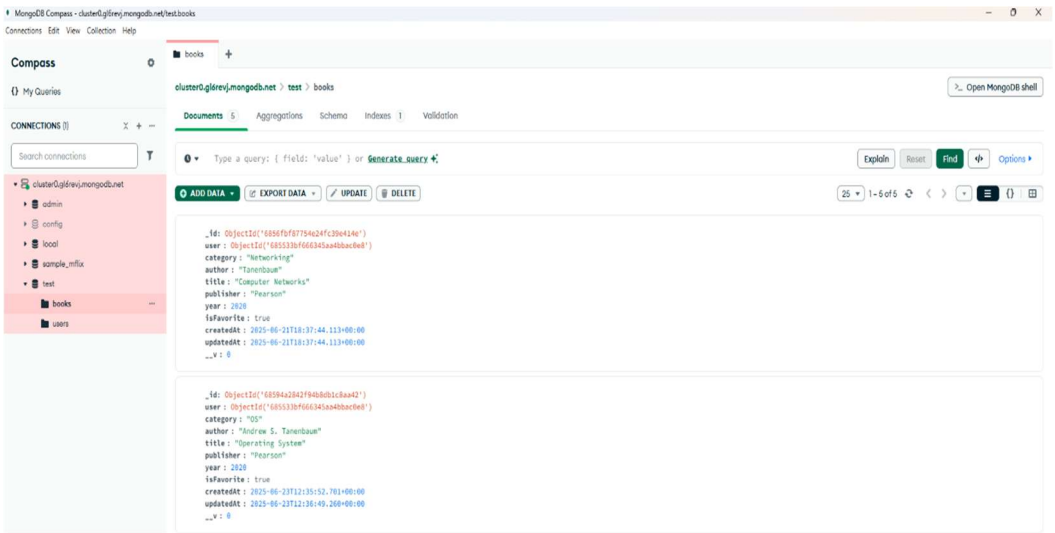


7. ábra [5]

5.2.1. Backend

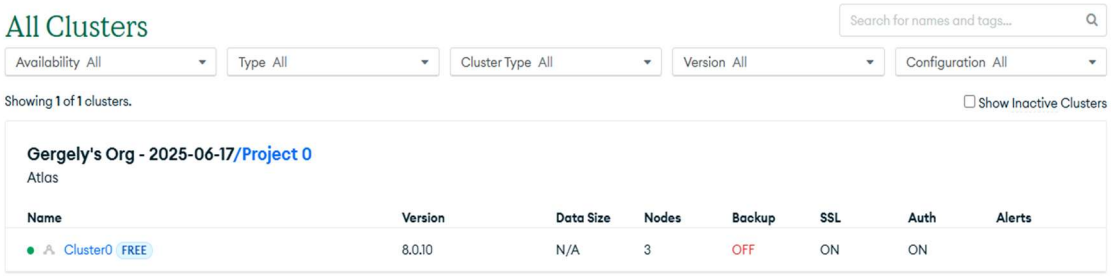
A MongoDB beállítása

A MongoDB Atlas a MongoDB felhőalapú verziója, amelyhez a Compass nevű letölthető asztali eszközzel lehet csatlakozni. A Compass bármely operációs rendszeren fut, és kapcsolatot biztosít az adatbázis eléréséhez. Például a mellékelt kép a projektben létrehozott books adatbázis tartalmát mutatja be a Compass felületén keresztül.

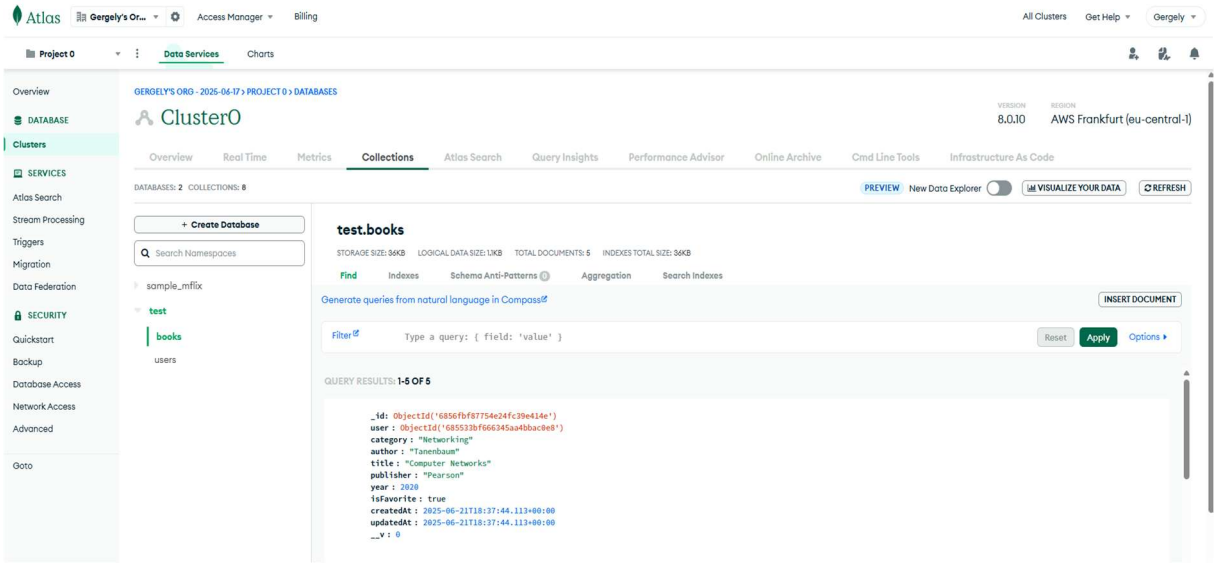


8. ábra [5]

A MongoDB-n létre kell hozni egy projektet és egy clustert, nekem az alapértelmezett maradt.



9. ábra [5]



10. ábra [5]

A létrehozáskor különböző biztonsági beállításokat végezhetünk el, mint például felhasználónév és jelszó, valamint IP-cím megadása; én az IP-címet 0.0.0.0-ra állítottam, hogy az adatbázis mindenhonnan elérhető legyen (Ez egyébként egy fontos védelmi pont, ha fix IP-címet használunk). Az Overview fülre kattintva a Clusteren belül csatlakoztathatjuk a backendhez (MongoDB shellen keresztül, az applikáción keresztül, és használhatunk MongoDB Compasst az itteni beállításoknál. Nekem a Compass sokkal szimpatikusabb volt a tesztelésnél, mert a mongodb.com oldalán viszonylag bonyolult elnavigálni az adatbázis megjelenítéséhez, a Compassnál ez

csak néhány kattintás, bár a Postman önmagában is elegendő, ott is látható egy GET kéréssel).

A szerverfájl-struktúra létrehozása

Létrehoztam egy backend mappát, azon belül a server.js-t.

Az npm init parancs segítségével létrehozom a package.json fájlt, amely a függőségeket és projektinformációkat, és konfigurációs beállításokat tartalmaz. Ilyen például a "scripts" részben, amely megmutatja az npm run paracsot milyen lehetőségekkel egészíthetjük ki, és mi történik. A kész backen package.json-ja a következő

11. ábra [5]

A következő lépés a .gitignore fájl létrehozása és kapcsolódás a githez. A .gitignore fájlhoz már ekkor hozzáadtam a node_modulest, kizárva a GitHubra való feltöltendő fájlok közül.

Npm package-ek hozzáadása: a npm install express dotenv mongoose colors bcryptjs parancs futtatásával a felsorolt eszközöket telepítettem (a dotenv készít egy

különálló konfigurációs fájlt, a colors a konzolüzenetek színezéséhez van, a bcrypts pedig a jelszavak hash értékének előállításához kell).

Felhasználtam még a nodemont is, amely a változtatásokat követően automatikusan újraindítja a szervert, így nem kell kézzel elvégezni.

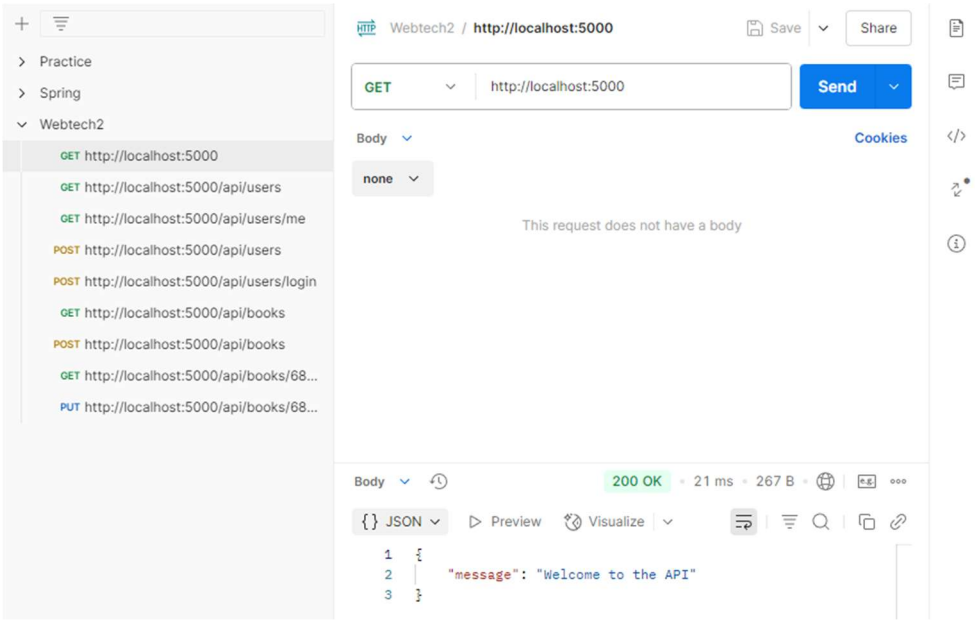
Alap Express szerver beállítások

Az Express szerver inicializálása: const express = require('express'); segítségével (ez a korábbi szintaxis, használható helyette az import segítségével is, de akkor a package.json-ben a "type" beállítását "module"-ra kell állítani).

A port beállítása: a .env fájlon keresztül éri el (5000-es port)

12. ábra [5]

A kezdeti beállítások elvégzése után teszt a Postammel: A válasz 200-as, tehát minden rendben ment, látható a beállított json formátumú üzenet.



13. ábra [5]

Útvonalak és vezérlő hozzáadása

A lépések a következők:

A backenden belül a routes mappa létrehozása, azon belül a userRoutes.js.

A user utak létrehozása a get és post requestek segítségével.

A controllers mappa létrehozása és a userController.js fájl benne. Itt szerepelnek a user utak function-jei, amelyeket a userRoutes.js felhasznál. Ehhez - mint minden hasonló esetben - a function-öket exportálni kell a kódjukat tartalmazó fájlból, és át kell adni a fájl elején (import vagy const változó segítségével), ahol azt felhasználjuk.

Hiba- és kivételkezelés

A lépések a következők:

Az első middleware, vagyis köztes réteg, amit használunk, az express.json. Ez teszi lehetővé, hogy json formátumban küldjük az információkat. A másik, a body parser middleware, amely az átadott url kód elemzésére való, kinyeri az adatokat a bodyból. Régebben ezt külön dependency-ként kellett használnunk, már tartalmazza az express. A userController.js-ben van megadva milyen adatokat vár a bodyból.

Ellenőrzések végrahajtása a bejelentkezéssel és a regisztrációval kapcsolatban. Mivel ezek hibát dobhatnak,

Hibakezelő middleware létrehozása a middleware mappában errorMiddleware néven. Ezt több helyen is alkalmazzuk később. ErrorHandler function létrehozása és exportálása, átadása a server.js-nek.

Express Async Handler telepítése, mert a regisztrációs és bejelentkezési függvényekben a Mongoose-t használjuk, egyik lehetőség ennek az alkalmazása. Az asyncHandler az aszinkron útvonalakon belüli kivételeket kezeli, és átadja azokat az Express kezelőnek. Ha nem használjuk az asyncHandlert,akkor a dokumentációban látható módon a .then függvényt kell használni, async await estében pedig try catch blokkot. A dokumentációja elérhető: https://www.npmjs.com/package/express-async-handler

Csatlakozás az adatbázishoz

A lépések a következők:

A MongoDB clusternél a connect gombra kattintva, majd a connect your application lehetőséget választva csatlakozhatunk (létrehoz egy karakterláncot, ami minden információt tartalmaz, ez a .env fájlban látható).

A backend mappán belül a config mappa létrehozása, abban a db.js fájl, ez lesz az adatbázis-kapcsolati fájl. Azon belül a connectDB function létrehozása és exportálása (az itt megjelenő konzolüzeneteket színezzük a korábban telepített colors segítségével).

Ezt követően a mongodb.com a clusterbeállításoknál a Network Access beállításainál megadhatjuk a szerver IP címét ("ADD CURRENT IP ADRESS" gomb segítségével), amelyről csatlakozni lehet az adatbázishoz (a 0.0.0.0 beállítással, vagy az "ALLOW ACCES FROM ANYWHERE" gombra kattintással bármely IP címről lehet csatlakozni).

A server.js-ben a connectDB functiont.

Felhasználó regisztrálása

A backend mappában létre kell hozni a models mappát, és a userModel.js fájlt. Ebben hozzuk létre az entitásunk sémáját (adattagjai felsorolása, ellenőrzés beállítása, alapértékek beállítása, unique elemek beállítása, és ellátjuk időbélyeggel timestamps: true,). A modell nevének beállítása és exportálás.

A userController.js-ben felhasználjuk a bcryptet, amellyel hashelni tudjuk a jelszót (plain text jelszót nem helyezhetünk el adatbázisban).

Felhasználjuk a modelt, jelen esetben a "User"-t. Ellenőrzések végrehajtása (például a regisztráció esetén, hogy létezik-e már az adott e-mail címmel felhasználó)

A hash password megjegyzéssel ellátott rész a userControlleren belül hajtja végre a jelszó hash-értékének előállítását a bcrypt segítségével, és salt, vagyis só használatával, amit a hash-elés előtt adunk a jelszóhoz. A genSalt metódus hajtja ezt végre, a hozzá tartozó szám a kívánt körök számát jelenti, a dokumentáció szerint 10 a javaslat. A Create User megjegyzéssel ellátott felhasználót létrehozó create functionben a jelszót hashedPassword, vagyis titkosított formában mentjük.

```
// Hash password
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
const isAdmin = req.body.isAdmin || false; // Allow setting isAdmin through request body
```

14. ábra [5][32]

A létrehozott felhasználóval vissza kell adnunk egy tokent, és egy id-t is. Fontos, hogy a MongoDB az id-t a következő módon tárolja: _id. Fontos az alsóvonás jel.

Bejelentkezés és JWT létrehozása

A loginUser function megvalósítása, a felhasználó létezésének és a tárolt jelszavával való egyezés ellenőrzése. Mivel a jelszó hash formában van tárolva, az összehasonlítást is így kell elvégezni.

Ha minden egyezik, hozzáadunk a felhasználóhoz egy tokent a generateToken felhasználásával, amivel telepíteni kell a json webtoken package-t az NPM-ből.

```
npm i jsonwebtoken
```

A generateToken megvalósítása:

```
// Generate JWT
const generateToken = (id) => {
    return jwt.sign({ id }, process.env.JWT_SECRET, {
        expiresIn: '30d',
    });
};
```

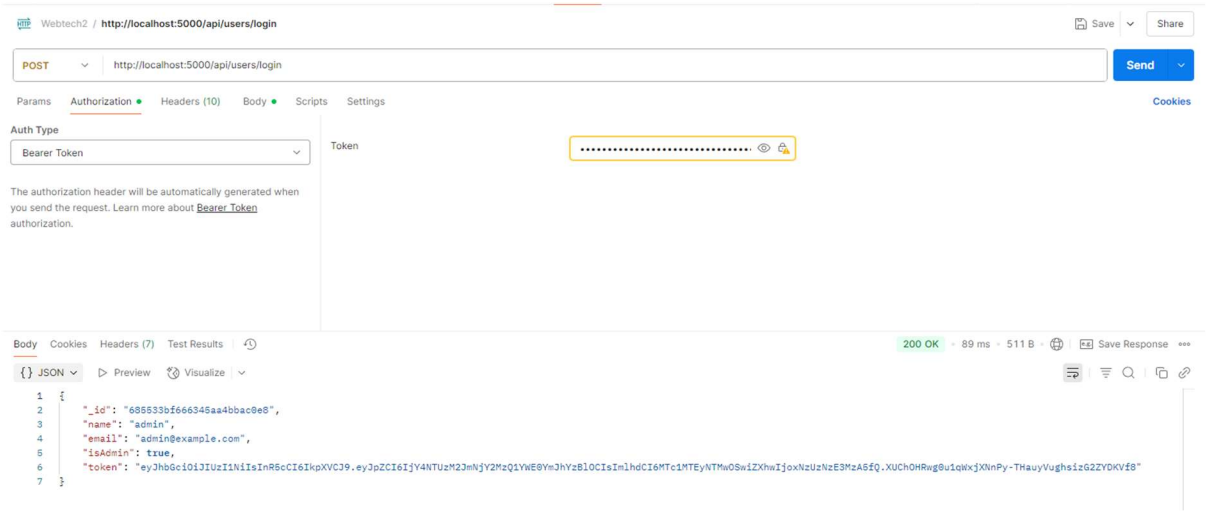
15. ábra [5][32]

A JSON Webtokenhez szükség van egy secret-re is, aminek az értéke bármi lehet. Ezt szintén a .env fájlból nyeri ki a function. Az expiresIn segítségével a token lejáríti idejét állíthatjuk be, itt 30 nap (30d).

A fentiek elvégzése után így néz ki egy tárolt felhasználó az adatbázisban:

16. ábra [5]

A Postmanben a helyes felhasználónév és jelszó megadásával a login oldalon (jelen esetben: http://localhost:5000/api/users/login) megjelenik a felhasználó tokenje is.



17. ábra [5]

Útvonalak és hitelesítés védelme

A JSON webtoken tartalmaz felhasználói azonosítót és néhány adatot, ennek segítségével védjük le az utakat. A userRoutesban a következő kód hajtja ezt végre: router.get('/me', protect, getMe);

A userControllerben szerepel a getMe function kódja.

```
// @desc    Get current user
// @route   POST /api/users/me
// @access  Private
const getMe = asyncHandler(async (req, res) => {
    const user = {
        _id: req.user._id,
        name: req.user.name,
        email: req.user.email,
        isAdmin: req.user.isAdmin
    }
    res.status(200).json({ user });
});
```

18. ábra [5][33]

Middleware létrehozása az útvonal védelmére a middleware mappába authMiddleware.js néven. Itt a protect function megvalósítása. Ez megszerzi a tokent a headerből, ellenőrzi és megszerzi a segítségével a felhasználói adatokat. (A Postmanben való tesztelés során is megadhatjuk a tokent, az Authorization fülre kattintva a legördülő menüből ki kell választani a Bearer Tokent, és beilleszteni).

```
You, last week | 1 author (You)
const jwt = require('jsonwebtoken');      You, last week • update backend
const asyncHandler = require('express-async-handler');
const User = require('../models/UserModel');

const protect = asyncHandler(async (req, res, next) => {
    let token;

    if(req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
        try {
            // Get token from header
            token = req.headers.authorization.split(' ')[1];
            // Verify token
            const decoded = jwt.verify(token, process.env.JWT_SECRET);

            // Get user from the token
            req.user = await User.findById(decoded.id).select('-password');
            next();
        } catch (error) {
            console.error(error);
            res.status(401);
            throw new Error('Not authorized');
        }
    }
    if (!token) {
        res.status(401);
        throw new Error('Not authorized, no token');
    }
});

module.exports = { protect };
```

19. ábra [5][33]

Egyéb entitások hozzáadása

A későbbiekben a Book entitás, a Book objektumtípust adtam még hozzá, előbbi lépések végrehajtásával.

5.2.2. Frontend

Create react app

A frontend elkészítéséhez a Vite-ot [34] használtam. Az

```
npx create react app@latest frontend --template redux [5]
```

parancs segítségével hozza létre Brad Traversy a React Front to Back [33] kurzusban a frontendet, amit felhasználva. Én ezzel hibákat tapasztaltam, előfordult, hogy nem jelenített meg semmit az oldal, az npm komoly sérülékenységekre hívta fel a figyelmet, és a redux esetében néhány fizetős package is előfordult. Ehelyett a Vite segítségével hoztam létre, ami sokkal stabilabb, és utólag adtam hozzá az npm-ből a redux-toolkitet.

```
npm create vite@latest my-react-app --template react [5]
```

A telepítéskor a React frameworköt és a JavaScript variant-ot választottam. Majd a frontend mappába lépve az npm install segítségével telepítettem a node modulokat.

20. ábra [5]

Emellett van még egy vite.config.js fájl is:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      // Minden /api kezdetű kérés továbbítása a backend szerverre
      '/api': {
        target: 'http://localhost:5000', // vagy ahol a backend fut
        changeOrigin: true,
        secure: false,
      },
    },
  }, Ctrl+I to continue with Copilot      You, last week • update wi
})
```

21. ábra [5]

Ha a frontendben is szerepel egy .git mappa, akkor azt törölni kell, nincs rá szükség, mivel a backend létrehozásakor már megtörtént.

Előbbiekkel létrejön egy alapvető mappaszerkezet. Az én verziómban nem jött létre az src mappán belül app mappa és store.js, ezt létrehoztam és exportáltam, mert szükség lesz rá.

A későbbiekben szereplő css fájlokat én készítettem a copilot segítségével.

Fejléc és kezdőlapok

A pages mappa létrehozása következett, a Home.jsx, Login.jsx és Register.jsx létrehozása az src mappán belül.

Az App.jsx-ben a

```
<Routes>
  <Route>
  </Route>
  <Route>
  </Route>
</Routes>
```

tagek között az utak létrehozása, tulajdonképpen az oldalak címeinek létrehozása.

A kész verzió a mellékletekben látható (1.melléklet).

A *Header.jsx* létrehozása az src mappán belül a components mappában, react ikonok használata. A kész verzió a következő:

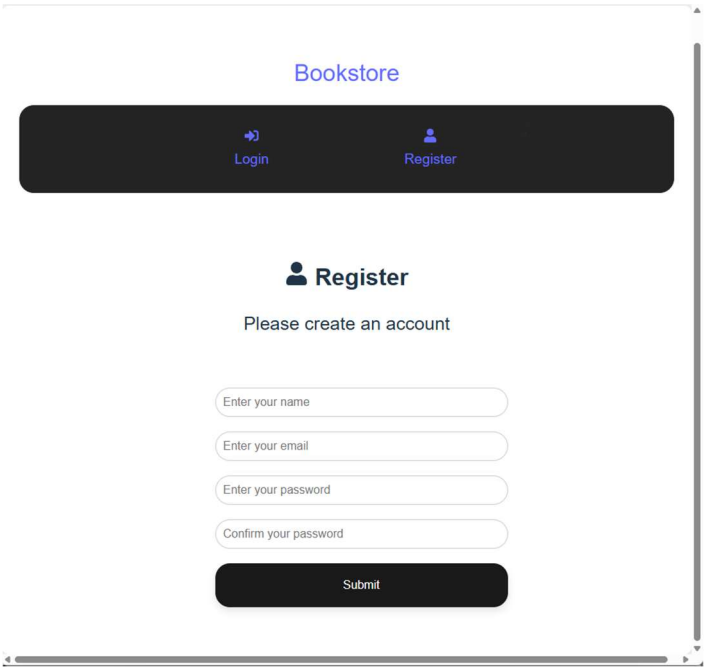
A következő importokra van szükség hozzá:

```
You, 4 days ago | 1 author (You)
import { FaSignInAlt, FaSignOutAlt, FaUser } from "react-icons/fa";
import { Link, useNavigate } from "react-router-dom";
import { useSelector, useDispatch } from "react-redux";
import { logout, reset } from "../features/auth/authSlice";
      You, last week • add and update frontend
```

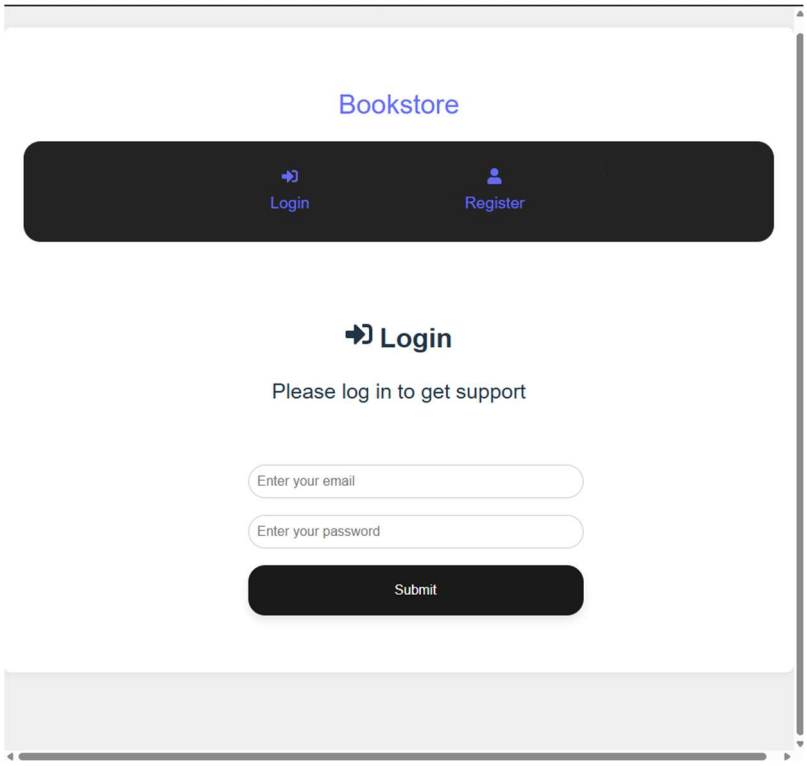
22. ábra [5][33]

Home, Login és Regisztrációs űrlapok létrehozása, felhasználói felület
A form létrehozása a Register.jsx-ben (a frontend/src/pages mappában található), a név, e-mail cím, jelszó, és jelszó megerősítése bekérésére. React ikonok és toast értesítések használata a react-toastify segítségével (a toastify működéséhez hozzá kell adni az App.jsx-hez a szükséges importokat és egy ToastContainert, lásd fentebb).

Az eredmény a következő:



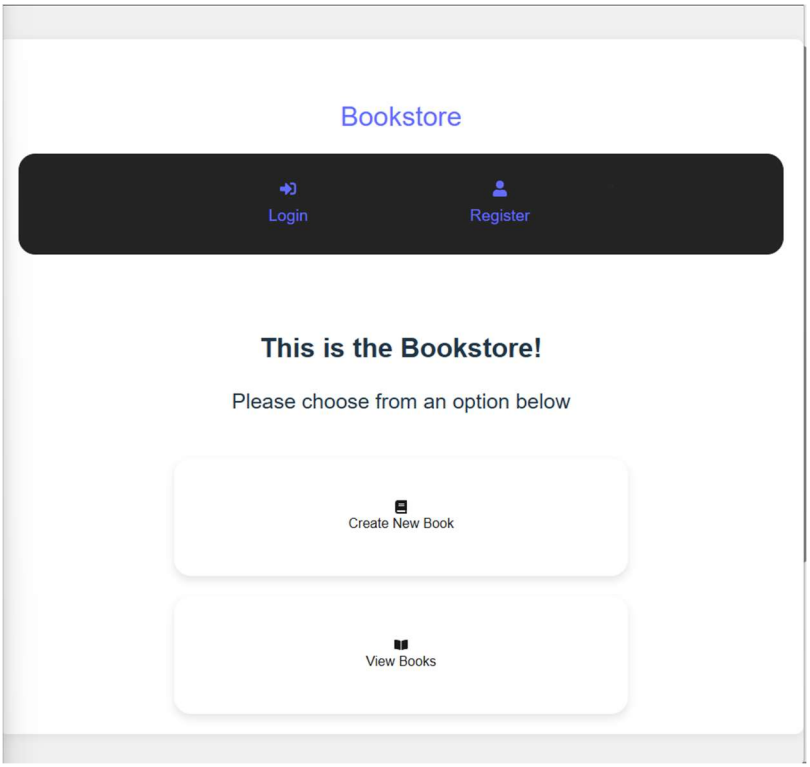
23. ábra [5]



24. ábra [5]

Következett a Home.jsx kódjának elkészítése, kiegészítése.

Az eredmény a következő:



25. ábra [5]

Redux beállítás és authSlice

Az src/feauters/auth mappában létrehozom az authSlice.js és az authService.js fájlokat.

Az authSlice.js-ben a @reduxjs/toolkit-ből a createAsyncThunk. Ezzel az állapotokat tudjuk kezelni. Tartalmaz initialState-et. A reducers üres objektum lesz, az extraReducers olyan függvények, amelyek a builder különböző állapotait valósítják meg.

Az authSlice.js-ben a register function megvalósítása, összekapcsolása a Register.jsx-szel. Ehhez utóbbinál useSelector (a globális állapot bármely részét be tudja vinni egy komponensbe) és useDispatch importálására van szükség a react-redux könyvtárból és a register functionre az authSlice.js-ből.

Felhasználó regisztrálása, kijelentkezése és bejelentkezése

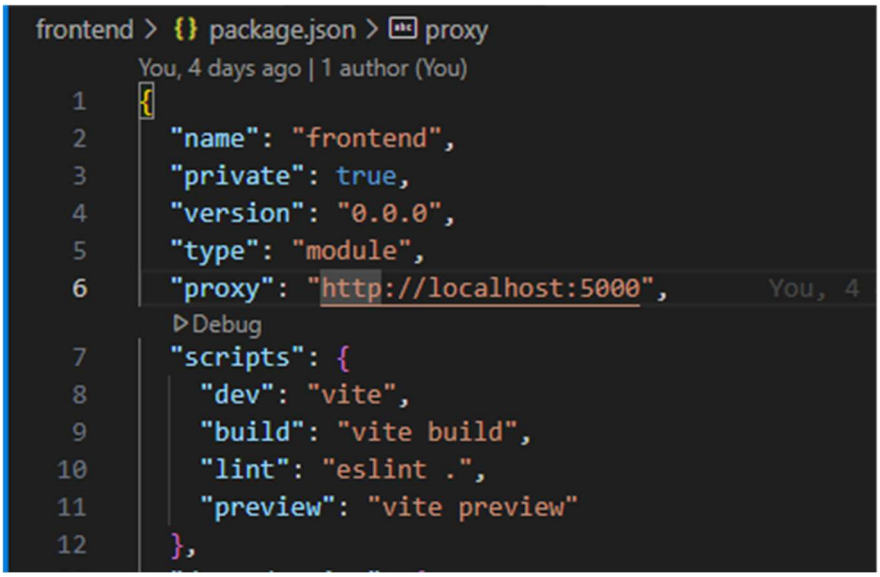
Az asyncThunk aszinkron módon működik, megkapja a user adatokat és a thunkAPI-t, és átadja azt authService.js-ben található register functionnek, amely továbbítja az axios importálávával az await axios.post segítségével a megfelelő url-re a userData néven mentett felhasználói adatokat.

Az állapotokat az authSlice-ban található extraReducerek kezelik.

A vite.config.js-ben és a frontendben található package.json-ban is be van állítva proxy, amely automatikusan a megadott url-re irányít anélkül, hogy mindenhova be kellene írni.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      // Minden /api kezdetű kérés továbbítása a backend szerverre
      '/api': {
        target: 'http://localhost:5000', // vagy ahol a backend fut
        changeOrigin: true,
        secure: false,
      },
    },
  }, Ctrl+I to continue with Copilot      You, last week • update wi
})
```

26. ábra [5]



27. ábra [5]

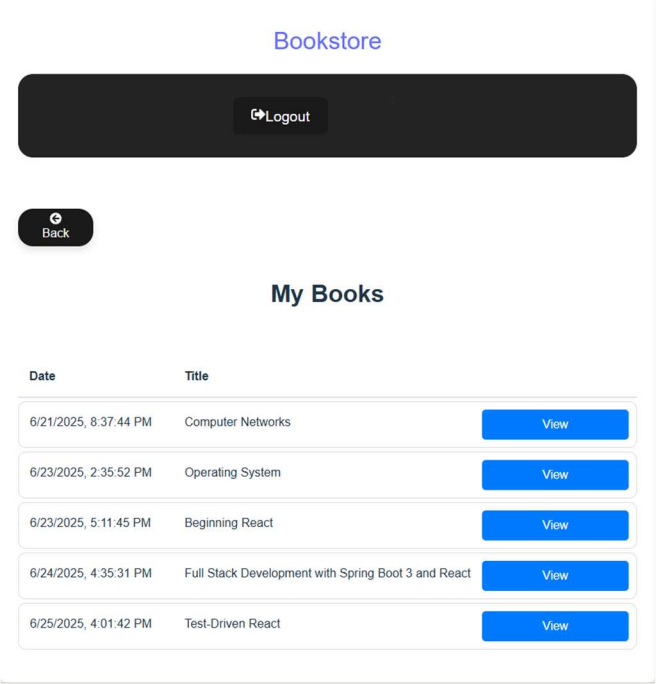
Látható, hogy a localhost:5000-es portra irányít, ahol a backend fut. 4. A bejelentkezés és kijelentkezés megvalósítása is az előbbi eszközökkel történik.

A Book-kal kapcsolatos műveletek létrehozása

A Book létrehozása a backendben, megjelenítése a frontendben az előbbiek felhasználásával történik. A backendben a korábban bemutatott lépések megismétlésével book modelt hozunk létre, összekapcsoljuk az adatbázissal, majd a felhasználóhoz kapcsoljuk.

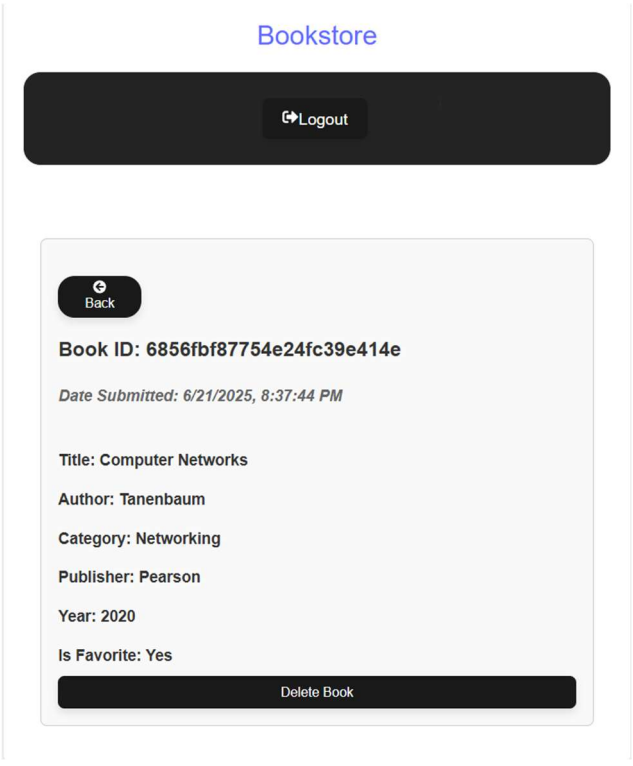
A bejelentkezést vagy regisztrációt követően elérhetők a felhasználóhoz kapcsolt könyvek, így csak a saját könyvei. Nem bejelentkezett felhasználót a kezdőlapról a könyvek megjelenítése helyett átirányít a bejelentkezési felületre. Bejelentkezett felhasználó új könyvet vihet fel a NewBook.jsx formjának segítségével, és megtekintheti a könyveit a View Books gomra kattintással. A gomb a Books.jsx oldalra visz, ami csak a cím alapján sorolja fel a könyveket, mind mellett egy View gombbal, amely megjeleníti a könyv további részleteit a Book.jsx oldalon keresztül. Itt található egy törlés gomb, ami a törlés funkciót valósítja meg egy delete requesten keresztül.

A *View Book* gombra kattintva tehát a felhasználó könyvei láthatók listázva.



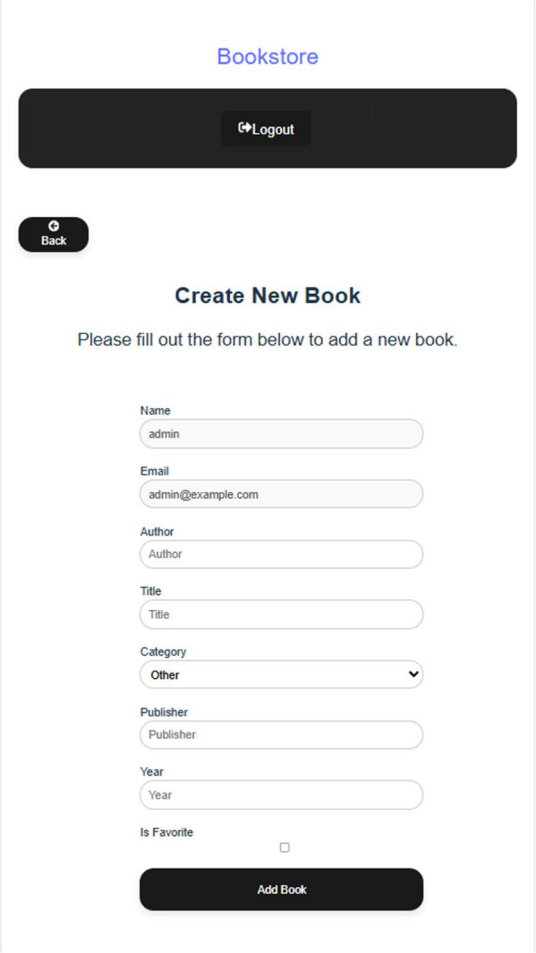
28. ábra [5]

A felhasználó egyes könyvei mellett található View gombbal, pedig eljutunk a könyvek adatlapjára, ahol látható a könyv azonosítója, adatai és egy Delete Book gomb alul, amivel a könyvet törölhetjük a nyilvántartásból.



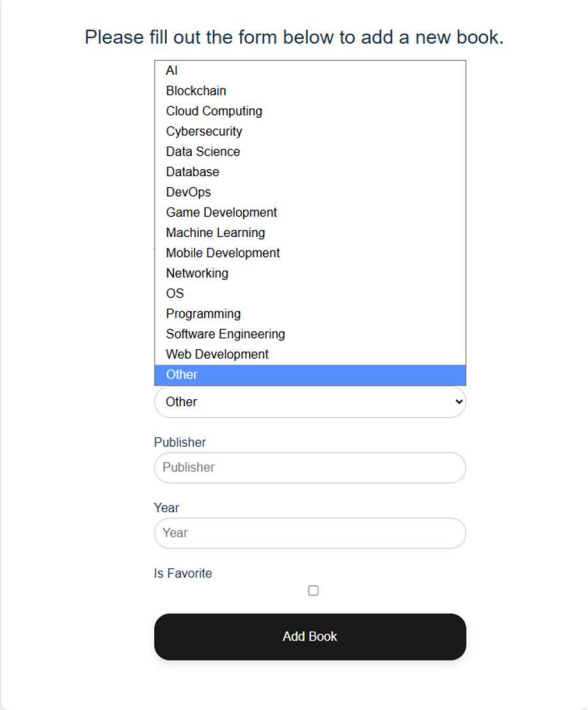
29. ábra [5]

És természetesen új könyvek felvitelére is van lehetőség, a backendben megvalósítottam egy működő put requestet is. A bejelentkezett felhasználó oldalán a New Book gombra kattintva az alábbi felületet érhetjük el, amely tartalmaz néhány bemeneti ellenőrzést, mint például az e-mail cím formátum, amit a frontend végez.



30. ábra [5]

Amint látható, a felhasználó nem módosítható, az szürke, csak jelzi a bejelentkezett felhasználó nevét. További megkötés, hogy a könyv kategóriája előre megadott témák közül választható.



31. ábra [5]

Az alkalmazás különféle biztonsági megoldásokat szemléltet. Mivel azonban sosem fogadhatjuk el a tesztelés során sem, hogy a kódunk hibamentes, ezért erről sem feltételezem ezt, a már bemutatott problémák mellett vizsgálom azt is, hogyan javítható még tovább az alkalmazás a biztonság szempontjából.

5.3. A Bookstore alkalmazás biztonsági elemzése [23][25][26][27][28][29]

Az ME-Bookstore egy MERN (MongoDB, Express, React, Node JS) alapú webalkalmazás, amely a gyakorlatban mutatja be a biztonsági követelmények alkalmazását. Az alábbi elemzés a korábban már tárgyalt alapelvek (CIA, AAA, Zero Trust) szempontjából vizsgálja meg a megvalósítást.

5.3.1. Hitelesítés (Authentication)

A hitelesítés biztosítja a felhasználók identitásának igazolását. A ME-Bookstore kritikus biztonsági intézkedéseket valósít meg ezen a téren.

A jelszókezelés *bcrypt*-et használ a jelszavak hasheléséhez és sózásához (10 körrel), mielőtt azokat a MongoDB-ben tárolná. A bcrypt egy visszafordíthatatlan (egyirányú) funkció, amely a sózással együtt védekezik a *Birthday Attack* és a *rainbow tables* típusú támadások ellen.

A *Munkamenet-kezelés* megoldása során, a sikeres bejelentkezéskor az alkalmazás *JSON Web Tokent (JWT)* generál, amelyet titkos kulccsal ír alá *(jwt.sign)* és lejárati időt állít be (30 nap). A JWT-t a kliens tárolja *(localStorage)*, és a szerveroldali *protect middleware* használja az *útvonalak (route protection)* védelmére és a felhasználó azonosítására.

Bár a hashelés erős, a jelenlegi implementáció egyfaktoros, ami a jelszókompromittálás (pl. adathalászat) kockázatát hordozza. Az MFA (például TOTP tokenek, amelyeket mobil applikációk generálnak, vagy a phishing-rezisztens WebAuthn/FIDO2 szabvány) bevezetése alapvető lépés a biztonság növelésére, mivel egy második, "valamid van" faktort ad hozzá az autentikációhoz.

5.3.2. Jogosultságkezelés (Authorization)

A jogosultságkezelés meghatározza, hogy a már hitelesített felhasználó milyen műveleteket végezhet. A ME-Bookstore *tulajdonosi alapú hozzáférés-szabályozást (Ownership-Based Access Control)* követ, kikényszerítve a *legkisebb jogosultság elvét (Principle of Least Privilege)*, mivel a felhasználó csak a saját könyveit láthatja és manipulálhatja.

Kiemelt kockázat az *Insecure Direct Object Reference (IDOR)*, így a kritikus pont az IDOR sebezhetőség megelőzése. Ez akkor fordul elő, ha a szerver nem ellenőrzi megfelelően a felhasználó tulajdonjogát az URL paraméterben vagy a kérés törzsében átadott objektumhoz (pl. könyv ID). A törlés (DELETE) művelet végrehajtása előtt a MongoDB lekérdezésében a *Mongoose* segítségével kötelezően ellenőrizni kell, hogy a törölni kívánt könyv azonosítója (_id) és a jelenleg bejelentkezett felhasználó ID-je is szerepeljen a keresési feltételek között. Ennek elmulasztása lehetővé tenné a támadónak, hogy más felhasználók könyveit törölje, ami az Integritás súlyos megsértését jelentené.

5.3.3. Bemeneti adatok érvényesítése és az adatintegritás (Input Validation and Data Integrity)

A bemeneti adatok érvényesítése az első és legkritikusabb védelmi vonal az olyan injektálási támadások ellen, mint az XSS (Cross-Site Scripting) és az SQL Injection, amelyek a vezető biztonsági fenyegetések közé tartoznak. Mivel a kliensoldali ellenőrzések könnyen manipulálhatóak és megkerülhetők, a szerveroldali validáció elengedhetetlen, és minden esetben ezen a rétegen kell megvalósítani.

A MERN stackben használt ME-Bookstore alkalmazás esetében a *Mongoose Object Data Mapper* kulcsszerepet tölt be a bemeneti adatok integritásának biztosításában. A Mongoose funkciói segítenek a NoSQL Injection megelőzésében is, ami egy speciálisan a nem-relációs adatbázis-kezelő rendszereket, például a MongoDB-t célzó támadástípus. Mivel a MongoDB minimális ellenőrzést végez a beillesztett adatokon, a NoSQL Injection elleni védelemért nagyrészt az alkalmazásréteg (a Mongoose/Express réteg) felelős a szigorú bemeneti érvényesítés (Input Validation) és tisztítás (Sanitization) alkalmazásával.

A Mongoose hatékony eszközöket biztosít ehhez. A *sémaalapú érvényesítés (Schema Validation)* a MongoDB *séma nélküli (schema-less)* természetére kényszerít ki egy struktúrát, beépített validációs képességeket kínálva. Ez lehetővé teszi az elvárt adattípusok (például String, Number) és a kötelező mezők (required: true) kikényszerítését. Emellett a sémák használatával korlátozásokat is be lehet állítani, mint például a stringek minimális és maximális hosszúsága (minlength, maxlength), valamint a számok minimális és maximális értéke (min, max). Ez jelentősen csökkenti a rosszindulatú adatok bekerülésének esélyét az adatbázisba.

A Mongoose támogatja az adattisztítást (Sanitization) is; például a trim: true beállítás automatikusan eltávolítja a kezdő és záró szóközöket a stringekből, ami egy általános védekezési mód a veszélyes karakterek eltávolításával az injektálási támadások ellen. Továbbá, mivel a MongoDB azonosítói ObjectId típusúak, kiemelten fontos, hogy az alkalmazásréteg ellenőrizze, hogy a kérésben kapott azonosítók (például a törléshez használt könyv ID-k) érvényes MongoDB ObjectId formátumúak legyenek, mielőtt az adatbázis-lekérdezés elindul. Bár a Mongoose képes az automatikus konverzióra, a manuális ellenőrzés (például egy isValid metódussal) megelőzi az esetleges CastError-okat.

Végül, a Mongoose ORM réteg (Object Data Mapper) használata magasabb szintre emeli a nyers adatbázis-lekérdezéseket, automatikusan paraméterezve azokat, ami általában biztonságosabb, mintha közvetlenül a MongoDB natív illesztőprogramjával dolgoznánk. Ez minimalizálja a manuális kódinjektálási hibák lehetőségét.

5.3.4. Munkamenet-kezelés és Támadások Megelőzése (XSS és XSRF)

A JWT tokenek munkamenet fenntartására való használata esetén a tárolás módja kritikus. Az ME-Bookstore, amely a JWT tokent a böngésző localStorage-ban tárolja, érzékeny a Cross-site Scripting (XSS) támadásokra. Sikeres XSS támadás esetén egy rosszindulatú JavaScript kód képes lehet kiolvasni a tokent a localStorage-ból, ami Session Hijacking-hez (munkamenet eltérítéshez) és fiókátvételhez vezet.

A védekezés alapja a bemeneti tisztítás (Input Sanitization) a szerveroldalon (mielőtt az adat elérné a Mongoose-t), valamint a *Content Security Policy (CSP) HTTP header bevezetése*, amely korlátozza a szkriptek betöltését és futtatását.

A *Cross-Site Request Forgery (CSRF)* támadás megelőzésére a leghatékonyabb módszer a token (vagy JWT) tárolása egy *HttpOnly* és *SameSite=Strict* attribútumokkal ellátott cookie-ban. Az HttpOnly megakadályozza, hogy a kliensoldali JavaScript hozzáférjen a cookie-hoz (enyhítve az XSS hatását), míg a SameSite=Strict megakadályozza a böngészőt, hogy a tokent kereszt-oldali (cross-origin) kérések során elküldje, meghiúsítva ezzel a CSRF támadást.

5.3.5. Naplózás és Monitorozás (Logging and Monitoring)

A naplózás és monitorozás biztosítja az elszámoltathatóságot (Accounting) és a biztonsági incidensek felderítését. Az ME-BookStore kiegészíthető lenne egy naplózást végrehajtó middleware-rel. Ha ez megvan, figyelni kell arra is, hogy a fejlesztés során használt konzolra írt naplók nem megfelelőek AppSec szempontból, mivel nem auditálhatók, nem központosítottak, és nem alkalmasak valós idejű anomália-észlelésre.

A legkritikusabb hiba, ha a naplók érzékeny adatokat (pl. jelszavakat, PII-t) tartalmaznak.

Szigorúan tilos a tiszta szöveges adatok naplózása. Bár a ME-Bookstore hasheli a jelszavakat, a hashek véletlen naplózásának elkerülése érdekében elengedhetetlen, hogy a Mongoose lekérdezésekben explicit módon zárjuk ki a hashelt jelszómezőt (select: false attribútummal) a válaszobjektumokból és a naplókból. Továbbá, a végfelhasználók felé generikus hibaüzeneteket kell küldeni, megakadályozva ezzel a Stack Trace-ek vagy adatbázis-hibaüzenetek kiszivárgását.

5.3.6. Titkos Adatok Kezelése (Secret Management)

Az ME-Bookstore a konfigurációs beállításokat és titkos kulcsokat (JWT_SECRET, MongoDB kapcsolati string) egy .env fájlban tárolja. Bár a .gitignore fájl védi a nyilvános repositoryba való feltöltéstől, a fájlrendszeri hozzáférés esetén a titkos adatok tiszta szövegben kiszivároghatnak.

A titkos adatok védelmének növelése érdekében egy dedikált *Titkos Adattároló (Secret Store)* vagy *szoftveres páncélterem* használata lenne a legjobb. Ez a megoldás titkosítja a kritikus adatokat (kapcsolati stringek, API kulcsok) és az alkalmazás programozott módon, API-n keresztül fér hozzájuk. Ez teljesen elvonja a szervertől a titkos adatok közvetlen, titkosítatlan tárolásának felelősségét, megakadályozva a véletlen kódolást és a kiszivárgást.

5.3.7. Harmadik Féltől Származó Összetevők Kockázata (Supply Chain Risk)

Mivel az ME-Bookstore Node.js-t és NPM csomagokat használ, az ellátási lánc biztonsága kiemelkedő fontosságú. A külső komponensekben rejlő sebezhetőségek elfogadásával az alkalmazás biztonsága gyengülhet.

A kockázatok azonosítására elengedhetetlen a *SCA (Software Composition Analysis)* eszközök használata, amelyek ellenőrzik a felhasznált NPM csomagokat az ismert sebezhetőségek (CVE) szempontjából. Ezenkívül a SAST és DAST eszközök folyamatos integrációja is szükséges.

Létre kell hozni a Szoftver Alkatrészjegyzéket (SBOM), és az SCA/SAST/DAST eszközöket automatikusan futtatni kell a CI/CD pipeline-ban (Shift Left). A folyamatos

frissítés és a szükségtelen függőségek eltávolítása szintén csökkenti a támadási felületet.

5.3.8. Hibakezelés és Biztonságos Kilépés (Error Handling and Secure Failing)

A helytelen hibakezelés súlyos biztonsági kockázatot jelent, különösen az Express/Node.js környezetben.

A rosszul kezelt *kivételek (Exceptions) Sensitive Data Exposure-t* okozhatnak, mivel a végfelhasználónak *Stack Trace-eket*, adatbázis-hibaüzeneteket vagy konfigurációs adatokat adnak át. Emellett a nem kezelt futásidejű hibák a szerver összeomlásához vezethetnek, lehetővé téve a DoS támadásokat.

A *Biztonságos Kudarc Elve (Fail Closed)* érvényesítése fontos, ha hiba történik, a rendszernek zárt, biztonságos állapotba kell kerülnie *(Fail Closed)*. A tranzakciókat vissza kell vonni, és a hibákat naplózni kell.

A hibakezelő *middleware* (köztes réteg) feladata, hogy a technikai hibát elkapja, de a felhasználónak csak generikus, minimális információt felfedő üzenetet küldjön. Ez megakadályozza a felhasználói számbavételi (Enumeration) támadásokat.

6. Felhasznált irodalom

Linkek:

- [1] https://www.hwsw.hu/hirek/69045/fbi-amerikaiszovetseginyomozoiroda-kar-kiberbunozo-aldozatok-veszteseg.html
- [2] https://www.ic3.gov/AnnualReport/Reports
- [3] https://www.fbi.gov/news/press-releases/fbi-releases-annual-internet-crime-report
- [4] A kiberbűnözés kora 2025. Mastercard tanulmány - https://www.mastercard.hu/content/dam/public/mastercardcom/eu/hu/pdfs/A%20kiberb%C5%B1n%C3%B6z%C3%A9s%20kora%202025.pdf
- [5] kovacsgergely22: *ME-Bookstore Repository (Readme.md)*, GitHub, (https://github.com/kovacsgergely22/ME-Bookstore)
- [6] https://www.exploit-db.com
- [7] https://nvd.nist.gov
- [8] https://capec.mitre.org
- [9] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- [10] https://cwe.mitre.org/index.html
- [11] https://cwe.mitre.org/top25/
- [12] https://learn.microsoft.com/en-us/security-updates/
- [13] https://support.apple.com/en-us/100100
- [14] https://source.android.com/docs/security/bulletin
- [15] https://ubuntu.com/security/notices
- [16] https://chromereleases.googleblog.com/
- [17] https://blog.jquery.com/
- [18] https://groups.google.com/g/nodejs-sec
- [19] https://nodejs.org/en/learn/getting-started/security-best-practices
- [20] https://bugs.python.org/
- [21] https://github.com/moby/moby/issues
- [22] https://bugs.mysql.com/

Könyvek:

- [23] Andrew Hoffman, *Web Application Security: Exploitation and Countermeasures for Modern Web Applications, Second Edition*, O'Reilly Media, Inc., 2024.
- [24] Dr. Adamkó Attila: *Fejlett Adatbázis Technológiák – Jegyzet*, 2013
- [25] Dr. R. Sarma Danturthi, *Database and Application Security: A Practitioner's Guide*, Pearson Education, 2024.
- [26] Loren Kohnfelder, *Designing Secure Software: A Guide for Developers*, No Starch Press, 2022.
- [27] Malcolm McDonald, *Web Security for Developers*, No Starch Press, 2020.
- [28] Reet Kaur, Yabing Wang (szerk.), *97 Things Every Application Security Professional Should Know*, O'Reilly Media, Inc., 2024.
- [29] Tanya Janca, *Alice & Bob Learn Application Security*, John Wiley & Sons, Inc., 2021.
- [30] *CompTIA Security+ (SY0-701) Bootcamp*. Dion Training 2025. – Study Guide

Videósorozatok, online kurzusok:

- [31] *CompTIA Security+ (SY0-701) Bootcamp*. Dion Training 2025. https://www.udemy.com/course/securityplus/
- [32] *Cyber Secure Coder (CSC-110)*. IT PRO TV https://www.udemy.com/course/cyber-secure-coder-csc-110/
- [33] *React Front to Back 2022*. Brad Traversy (https://subscription.packtpub.com/video/web-development/9781838645274/p1/video1_1/welcome-to-the-course)
- [34] *Become a Full-Stack Web Developer with just ONE course. HTML, CSS, Javascript, Node, React, PostgreSQL, Web3 and DApps*. Dr Angela Yu (https://www.udemy.com/course/the-complete-web-development-bootcamp/learn/lecture/12638830#overview)

7. Összefoglalás

A szoftverfejlesztés elengedhetetlen része a programozók biztonságos kódolás iránti folyamatos felelősségvállalása. A sebezhetőségek tudatosítása és megelőzése rendkívül fontos, alapvetően az adatok védelmére fókuszálva, amely a CIA-modell (Titoktartás, Integritás, Elérhetőség) és a Letagadhatatlanság alapelvei köré épül.

A felhasználói hozzáférés menedzsmentjében (IAM) emiatt elengedhetetlen a 3A + Accounting modell: Azonosítás, Hitelesítés, Engedélyezés, valamint a kulcsfontosságú Naplózás (Accounting), amely az elszámoltathatóságot és az incidensek kivizsgálását szolgálja.

A fenyegetések és sebezhetőségek minimalizálása már a tervezés fázisától kezdve alapvető, különösen a kódinjektálási támadások esetében, amelyeket a bemeneti adatok érvényesítésével és a rosszindulatú kód tisztításával kell megelőzni.

Az ME-Bookstore MERN webalkalmazás elemzése bemutatta, hogyan alkalmazhatók ezek a biztonsági alapelvek a gyakorlatban (bcrypt hashelés sózással a jelszavakhoz, a digitálisan aláírt JSON Web Token (JWT) használata a munkamenet fenntartására, stb.) miközben a rendszer tulajdonosi alapú hozzáférés-szabályozást és a legkisebb jogosultság elvét is érvényesíti. Tárgyaltuk a Multi-Faktoros Hitelesítést (MFA), és a titkos adatok kezelését.

Végül, a biztonságot a tervezés fázisától integrálni kell a Szoftverfejlesztési Életciklusba (SSDLC), létrehozva a Biztonságos Szoftverfejlesztési Életciklust. A valós idejű naplózás és monitorozás, valamint a hibák biztonságos kezelése, szintén kiemelt téma.

Az alkalmazások biztonsága tehát jól láthatóan kulcsfontosságú kérdés, a felhasználók és a fejlesztők szemszögéből is. Ne hagyjuk, hogy az online térben elkövetett csalások kárértéke 2028-ra elérje a becsült 14000 milliárd dollárt!

8. Secure Software Development: Summary

An essential part of software development is the continuous commitment of programmers to secure coding. Awareness and prevention of vulnerabilities are extremely important, fundamentally focusing on data protection, which is built around the principles of the CIA triad (Confidentiality, Integrity, Availability) and Non-repudiation.

Consequently, the 3A + Accounting model is essential in User Access Management (IAM): Identification, Authentication, Authorization, plus the crucial Accounting (Auditing/Logging), which serves accountability and incident investigation.

Minimizing threats and vulnerabilities is fundamental right from the design phase, particularly in the case of code injection attacks, which must be prevented by input data validation and sanitization of malicious code.

The analysis of the ME-Bookstore MERN web application demonstrated how these security principles can be applied in practice (bcrypt hashing with salting for passwords, the use of digitally signed JSON Web Token (JWT) for session maintenance, etc.) while the system also enforces owner-based access control and the principle of least privilege. We discussed Multi-Factor Authentication (MFA) and secret data management.

Finally, security must be integrated from the design phase into the Software Development Life Cycle (SDLC), creating the Secure Software Development Life Cycle (SSDLC). Real-time logging and monitoring, as well as secure error handling, are also highlighted topics.

Application security is thus clearly a key issue, from both the users' and developers' perspectives. Let's not allow the estimated damage value of online fraud to reach 14 trillion dollars by 2028!

9. Mellékletek

9.1. Mellékletek

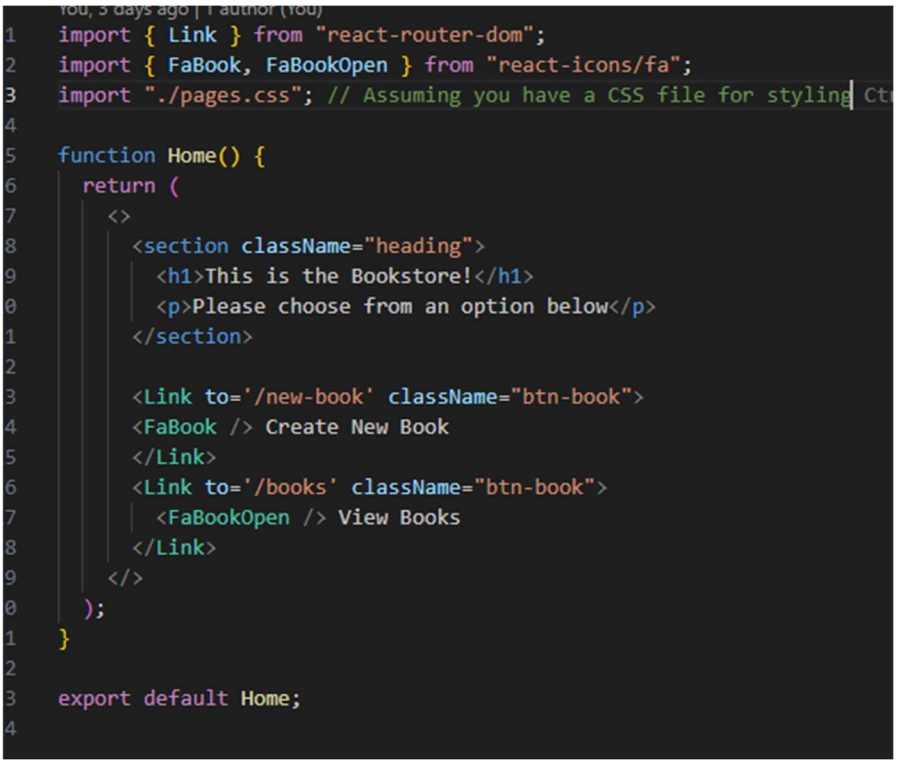
9.1.1. A function App kódja a React Router felhasználásával

```
You, 3 days ago | 1 author (You)
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";
import Header from "./components/Header";
import PrivateRoute from "./components/PrivateRoute";
import Home from "./pages/Home";
import Login from "./pages/Login";
import Register from "./pages/Register";
import NewBook from "./pages/NewBook";
import Books from "./pages/Books";
import Book from "./pages/Book";

import "./App.css"; // Assuming you have a CSS file for styles

function App() {
  return (
    <>
      <Router>
        <div className="container">
          <Header />
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/login" element={<Login />} />
            <Route path="/register" element={<Register />} />
            <Route element={<PrivateRoute />}>
              <Route path="/new-book" element={<NewBook />} />
            </Route>
            <Route element={<PrivateRoute />}>
              <Route path="/books" element={<Books />} />
            </Route>
            <Route element={<PrivateRoute />}>
              <Route path="/book/:bookId" element={<Book />} />        You, 3
            </Route>
          </Routes>
        </div>
      </Router>
      <ToastContainer />
    </>
  );
}

export default App;
```

9.1.2. A Header React komponens kódja

```
function Header() {
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const { user } = useSelector((state) => state.auth);

  const onLogout = () => {
    dispatch(logout());
    dispatch(reset());
    navigate("/");
  };

  return (
    <header className="header">
      <div className="logo">
        <Link to="/">Bookstore</Link>
      </div>
      <ul>
        {user ? (
          <li>
            <button className="btn" onClick={onLogout}>
              <FaSignOutAlt />
              Logout
            </button>
          </li>
        ) : (
          <>
            <li>
              <Link to="/login">
                <FaSignInAlt /> Login
              </Link>
            </li>

            <li>
              <Link to="/register">
                <FaUser /> Register
              </Link>
            </li>
          </>
        )}
        ;
      </ul>
    </header>
  );
}

export default Header;
```

9.1.3. A Home.jsx React komponens kódja



9.2. CD/DVD melléklet tartalma

9.2.1. Dolgozat mappa

- Biztonsági rések a pajzson.docx – a szakdolgozat szerkeszthető formátumban
- Biztonsági rések a pajzson.pdf – a szakdolgozat pdf formátumban
- Összefoglalás.docx
- Összefoglalás.pdf
- Summary.docx
- Summary.pdf

9.2.2. ME-Bookstore alkalmazás mappa

- ME-Bookstore-master.zip – a tartalmazó GitHub repository kódja zip fájlban
- ME-Bookstore-master mappa – a tartalmazó GitHub repository kódja kitömörítve