Hogyan fejlesszünk biztonságos szoftvert?

Gyakorlati lépések

Erdélyi Miklós
erdelyi@dcs.uni-pannon.hu

Miről is lesz szó?

- Alapvetően szoftverbiztonságról
- Kockázat- és fenyegetéselemzés
- Szoftverhibák taxonómiája
- Kódvizsgálati eljárások
- Tesztelés és verifikáció
- Szoftverbiztonság mérési kérdései

Biztonság (Oroszi Norbert diáiból)

- Informatikai alapfenyegetettségnek azon fenyegető tényezők hatásösszegét nevezzük, amelyek az információk
  - rendelkezésre állását (availability);
  - sértetlenségét (integrity);
  - bizalmasságát (confidentiality);
  - hitelességét (authenticity);
- illetve az informatikai rendszer
  - rendelkezésre állását;
  - funkcionalitását
- veszélyeztetik.

Példa

- Alice és Bob rézvezetéken keresztül kommunikál morze jelekkel
  - Eve lehallgató eszközt helyez el
  - Mi sérül?
- A szituáció ugyanaz
  - Eve elvágja a vezetéket és morze készüléket helyez mindkét végére
  - És ekkor?

Biztonságos szoftverfejlesztés lépésekben (Fortify Software Inc.)

- 1. A szoftverbiztonság jelenlegi állapotának felmérése és terv készítése kezelésére a szoftverfejlesztési életciklus során.
- 2. Kockázatok, fenyegetések felmérése és kiküszöbölése még a fejlesztés során.
- 3. Kódvizsgálati eljárások alkalmazása.
- 4. Sérülékenységek sikeres eltávolításának tesztelése és verifikációja.
- 5. Ellenőrző lépés megkövetelése, hogy a sérülékenységet tartalmazó alkalmazásokat ne lehessen bevezetni.
- 6. A folyamat eredményességének állandó mérése a visszacsatoláshoz.

1. lépés: Felmérés és tervezés

- Felmérés: egyszerű, pontozásos módszerrel a szoftverbiztonsággal kapcsolatos jelenlegi tevékenységekre és tervezett tevékenységekre vonatkozóan.
- Tervezés fontos szempontjai:
  - Szoftverfejlesztési projektek szoftverinfrastruktúrája (IDE, fordító, verziókövető-rendszer, etc.)
  - Milyen biztonsági lépéseket kell tennie a szoftverfejlesztési csapatoknak? (Eszközök bevezetése, kódvizsgálati folyamat, etc.)
  - Hogyan kezeljük a feltárt biztonsági problémákat? (Dokumentálás, tesztelés és verifikáció, etc.)

Példa: Felmérés eredménye

Elements for a Plan

Assigned security expert or team lead	Who: Assign individuals to tasks and roles	How issues will be tracked and reported
Automated tools that support steps	What: List steps and success criteria for each	Team processes to triage, prioritize, and take action on reported security defects
Process checklists & requirements	When: Include steps and activities in project timeline	

2. lépés: Kockázat- és fenyegetéselemzés (1)

- Kockázatelemzés: kockázatok részletes meghatározása, számszerűsítése.
- Módszertanok:
  - STRIDE (Microsoft);
  - ASSET (NIST).

Kockázatelemzés: STRIDE módszertan

Types of threats	Examples
Spoofing	• Forging e-mail messages • Replaying authentication packets
Tampering	• Altering data during transmission • Changing data in files
Repudiation	• Deleting a critical file and deny it • Purchasing a product and deny it
Information disclosure	• Exposing information in error messages • Exposing code on Web sites
Denial of service	• Flooding a network with SYN packets • Flooding a network with forged ICMP packets
Elevation of privilege	• Exploiting buffer overruns to gain system privileges • Obtaining administrator privileges illegitimately

2. lépés: Kockázat- és fenyegetéselemzés (2)

- Fenyegetéselemzés: meghatározza, hogy a kódvizsgálat és a biztonsági tesztelés az alkalmazás mely komponenseire koncentráljon.
- Két fázisa:
  - 1. A megvédendő elemek azonosítása az alkalmazásban (pl. felhasználói adatok, bankkártyaszámok, etc.) és ezek fontossági sorrendbe állítása.
  - 2. Alkalmazás működésének megértése és a támadási lehetőségek feltérképezése. Ehhez el kell készíteni az alkalmazás komponenseinek és adatútjainak magas-szintű modelljét, majd a támadási felületeket kell meghatározni: mely interfészek fogadnak felhasználói vagy más rendszertől való bemeneti adatot. Integritás, rendelkezésre állás, bizalmasság hogyan és hol sérülhet?

2. lépés: Kockázat- és fenyegetéselemzés (3)

- A legátfogóbb fenyegetéselemzéssel sem kerülhető el a biztonsági rések „beépítése” a fejlesztés során.
- Ezért van szükség a kódvizsgálatra...

3. lépés: Kódvizsgálat

- Célja: a biztonsági problémák mihamarabbi felismerése a szoftverfejlesztési életciklusban.
- Alapja általában maga a forráskód.
- Történhet manuálisan és automatikus elemző segítségével is.
  - Teljesen nem automatizálható!
- Mit vizsgálunk?

Szoftverbiztonsági hibák taxonómiája

- A 7 „veszedelmes” ország:
  - Bemenet ellenőrzése és reprezentációja
  - Helytelen API használat
  - Biztonsági funkciók
  - Idő és állapot
  - Hibák és hibakezelés
  - Kódminőség
  - Beágyazás (encapsulation)
  - Környezet
- Valamennyi sérülékenység típus besorolható valamelyik országba.
- A sérülékenység típusok törzsekbe tartoznak.

1. Bemenet ellenőrzése és reprezentációja

- A problémák abból adódnak, ha a bemeneti adatok helyességében „vakon” bízik az azokat fogadó kód.
- Főbb törzsei:
  - *Puffer túlcsordulás*. Az allokált memória határain túli írás adatkorrupcióhoz, a program lefagyásához vagy a támadó által megadott utasítások végrehajtásához vezethet.
  - *Parancs injekció*. Nem megbízható forrásból származó parancsok, vagy nem megbízható környezetben végrehajtott parancsok azt eredményezhetik, hogy az alkalmazás a támadó rosszindulatú parancsait hajtsa végre.
  - *Cross-site scripting*. Web böngészőnek küldött ellenőrizetlen adat rosszindulatú kód (általában szkriptek) végrehajtását eredményezheti a böngészőben.
  - *Formátum sztring* (format string). A támadó által manipulálható formátum sztring puffer túlcsordulást eredményezhet.
  - *Egész túlcsordulás (integer overflow).* Az egész túlcsordulás figyelmen kívül hagyása logikai hibát vagy puffer túlcsordulást eredményezhet.

2. Helytelen API használat

- Az API szerződést jelent a meghívó program és a meghívott osztály/programkönyvtár között. A helytelen API használat leggyakoribb formája az, amikor az API-t használó szoftver nem teljesíti „szerződési kötelezettségét”.
- Példa: DNS feloldó programkönyvtár.
- Főbb törzsei:
  - *Verem inspekció (heap inspection).* Tilos a realloc() használata érzékeny adatokat tartalmazó pufferek átméretezésére.
  - *Gyakran helytelenül használt: jogosultságkezelés.* A legkisebb jogosultság elvének figyelmen kívül hagyása megsokszorozza az egyéb sérülékenységek által jelentett kockázatokat.
  - *Gyakran helytelenül használt: sztringek.* Sztringeket manipuláló függvények esetén előfordulhat puffer túlcsordulás.
  - *Ellenőrizetlen visszatérési érték.* Metódus visszatérési értékének figyelmen kívül hagyásával a váratlan állapotokat és feltételeket a program nem kezelheti megfelelő módon.

3. Biztonsági funkciók

- Ide tartoznak a hitelesítés, hozzáférés-szabályozás (access control), bizalmasság, kriptográfia és jogosultságkezelés témakörébe tartozó hibák.
- Példa törzsek:
  - *Nem biztonságosan generált véletlenszámok.* A hagyományos pszeudo-véletlenszám generátorok nem védettek a kriptográfiai támadások ellen.
  - *Legkisebb jogosultság elvének megsértése*. A megemelt jogosultsági szinthez kötött műveletek (mint például chroot() hívás) elvégzése után a már nem szükséges jogosultságoktól meg kell válni.
  - *Jelszókezelés*. A jelszavak egyszerű szövegként való tárolása a rendszer feltörését eredményezheti.
  - *Jelszókezelés: üres jelszó*. Üres karakterlánc, mint jelszó használata nem biztonságos.
  - *Jelszókezelés: „bedrótozott" jelszavak*. A „bedrótozott" jelszavak a rendszer biztonságát előre nem látható módon veszélyeztethetik.

4. Idő és állapot

- Elosztott számítógépes feldolgozás esetén fontos az idő és az állapot. Mert ahhoz, hogy több komponens kommunikáljon, állapotot kell megosztaniuk, és mindez időbe kerül.
- Főbb törzsei:
  - *Holtpont*. Az inkonzisztens zárolás holtponthoz vezethet.
  - *Fájl-hozzáférési versenyhelyzet: TOCTOU (time-of-check-to-time-of-use)*. Egy fájl tulajdonságának ellenőrzése és a fájl tényleges használata között eltelt idő a támadó által kiaknázható a jogosultsági-szintjének megemelésére.
  - *Nem biztonságos átmeneti fájl*. Az átmeneti fájlok nem biztonságos módon való létrehozása és használata támadásnak teheti ki az alkalmazás- és rendszeradatokat.
  - *Jelzéskezelők versenyhelyzetei*. Egyes jelzéskezelők megváltoztathatják a más jelzéskezelők, vagy az alkalmazás kód által megosztott állapotot, ami váratlan működést eredményezhet.

5. Hibák és hibakezelés

- Kétféle módon keletkezhet hibákkal összefüggő biztonsági rés:
  - a hibák nem megfelelő, vagy egyáltalán nem kezelése;
  - olyan hibaüzenetek előállítása, amelyek túl beszédesek (túl sok információt fednek fel a potenciális támadóknak), vagy olyan hibák produkálása, amelyeknek nehézkes a kezelésük.
- Főbb törzsei:
  - *Üres catch blokk*. A kivételek vagy egyéb hibaállapotok figyelmen kívül hagyása a támadó számára lehetővé teszi, hogy észrevétlenül hozzon létre váratlan működést a programban.
  - *Túlzottan szerteágazó catch blokk*. Túl sok kivétel kezelése komplex hibakezelő kódot eredményez, amivel megnő a valószínűsége annak, hogy az egyben biztonsági sérülékenységeket is tartalmaz.
  - *Túl sok kivétel a throws deklarációban.* Ha egy metódus túl sokféle kivételt dobhat, komplex hibakezelő kód szükséges használatánál, amivel megnő a valószínűsége annak, hogy egyben biztonsági sérülékenységeket is tartalmaz a hibakezelő kód.

6. Kódminőség

- A gyenge kódminőség kiszámíthatatlan működéshez vezet.
- Felhasználói szemszögből ez gyakran a nehezen használhatóságból érződik.
- Egy támadó számára lehetőséget teremt arra, hogy a rendszert váratlan megpróbáltatásoknak vesse alá.
- Példák:
  - *Többszörösen meghívott free() függvény*. A free() függvényt ugyanarra a memóriacímre többször meghívva puffer túlcsordulás fordulhat elő.
  - *Nem következetes implementáció*. Az operációs rendszerek vagy operációs rendszer verziók között nem megegyező implementációval rendelkező függvények megnehezítik a kód hordozhatóságát.
  - *Memóriaszivárgás (memory leak)*. A már nem használt, előzőleg lefoglalt memóriaterületek felszabadításának elmulasztása a szabad memória kimerüléséhez vezet.
  - *Elavult API használata*. Az elavult vagy érvénytelenített API függvények használata nem karbantartott kódot jelezhet.
  - *Inicializálatlan változó*. A program olyan változót használhat, ami előzőleg még nem kapott értéket.
  - *Felszabadított memóriaterület használata*. A program lefagyását okozhatja, ha előzőleg felszabadított memóriaterületre hivatkozik.

7. Beágyazás

- A határok pontos meghúzásáról szól: kire mi tartozik.
- Példák:
  - *Osztályok összehasonlítása név alapján*. Két osztály nevük alapján történő összehasonlítása azt eredményezheti, hogy a program két különböző osztályt megegyezőként kezel.
  - *Ottfelejtett hibakereső kód*. A hibakeresésnél használt kód az alkalmazásba nem szándékolt belépési pontokat jelenthet.
  - *Publikus metódus privát tömbtípusú mezőt ad vissza*. Egy privát tömb tartalma váratlan módon megváltozhat a publikus metódus által visszaadott referencián keresztül.
  - *Publikus adatmező hozzárendelése privát tömbtípusú mezőhöz*. A publikus adatmező hozzárendelése a privát tömbhöz a privát mezőhöz való publikus hozzáféréssel egyenértékű.
  - *Rendszerinformáció kiszivárogtatása*. A kiszivárogtatott rendszeradatok vagy hibakeresési információ megkönnyítheti egy külső fél számára támadási terv kidolgozását.
  - *Megbízható adatok elkülönítése*. A megbízható és nem megbízható adatok ugyanazon adatstruktúrában való tárolása azt eredményezheti, hogy a programozó összekeveri őket.

*. Környezet

- Mindazt magába foglalja, ami nem a forráskódhoz tartozik, de mégis kritikus a szoftvertermék biztonságossága szempontjából.
- Példa: a fordító nem biztonságosan optimalizál. A memóriából az érzékeny adatok nem megfelelő törlése megsérti a bizalmasság elvét, mivel más folyamat által olvashatóvá válhat.

Statikus vs. dinamikus kódelemzés

- Statikus kódelemzés: a legelterjedtebb technika.
  - Automatizált eszközökkel elemezzük magát az alkalmazás forráskódját vagy az abból előállított köztes kódot. Az automatizált statikus elemzők tudják a legtöbbféle sérülékenységet megtalálni valamennyi módszer közül. A statikus kódelemzés további előnye, hogy nem kell a szoftvert futtatni, ezért gyorsabb, mint a dinamikus elemzés.
- Dinamikus kódelemzés: a sérülékenységek olyan csoportjának feltárására való, amelyek csak a kód futtatása során jönnek elő.
  - Például az alkalmazás konfigurációjából vagy környezetéből adódóan. A tesztelés során automatizált dinamikus elemzők használhatók ilyen sérülékenységek feltárására. Emiatt a leghatékonyabbak azok az automatizált dinamikus elemzők, amelyek a vállalat meglévő tesztelési protokolljába integrálhatók. Azonban fontos kiemelni, hogy a dinamikus elemzés csak a kód azon részeit tudja vizsgálni, amely végrehajtódik, tehát a rendszer nem futtatott részeiben előforduló sérülékenységeket nem tudja megtalálni. Ebből fakadóan a dinamikus elemzés eredményei csak a tesztelés során érintett kód állapotát tükrözik.

Statikus vs. dinamikus kódelemzés (2)

- Statikus kódelemzők
  - Pro:
    - rejtett, futás során ritkán előforduló hibák felderítése;
    - helytelen memóriakezelésből adódó biztonsági rések.
  - Contra:
    - fals pozitívok nagy aránya.
- Dinamikus kódelemzők
  - Pro:
    - forráskód nélkül használható.
  - Contra:
    - lassú;
    - nehezen elérhető program-állapotokból adódó hibák figyelmen kívül maradhatnak.

Példa: FlawFinder (C/C++)

- Statikus kódelemző.
- C/C++ függvények jól-ismert problémáit tartalmazó beépített adatbázist használ az adott forráskód potenciális biztonsági réseinek előállítására, amely listát kockázat szerint rendez.

Példa: FindBugs (Java)

- Statikus kódelemző, Java bájtkód alapján dolgozik.
- Hiba-mintákat keres, amelyek tipikusan a bonyolult nyelvi elemek helytelen használatából, a rosszul használt API hívásokból, a kód karbantartása során tévesen ugyanolyan működést feltételező módosításokból illetve számos apró hibából, mint például elgépelésekből, a téves logikai művelet alkalmazásából, stb., származtathatók.

4. lépés: Tesztelés és verifikáció

- Célja annak ellenőrzése, hogy a kódvizsgálat során feltárt hibákat helyesen javították.
- Kézi módszer: penetrációs tesztek.
  - Támadható- ea rendszer?
- Automatikus eszközök:
  - Például: Retina, NESSUS, Fortify.
  - Csak ismert sérülékenységeket találnak meg!

5. lépés: Biztonsági ellenőrzés

- Ellenőrzési lista elemei:
  - Kockázatanalízis vagy fenyegetésanalízis megléte.
  - Biztonsági kódellenőrzések megléte, hibák javítása.
  - Biztonsági tesztelés sikeres (újabb komoly hibákat nem találtak, kisebb hibák javítva).
  - Biztonsági audit független biztonsági szakértőkkel (opcionális).
- Továbblépni csak akkor lehet, ha valamennyi ellenőrzési ponton megfelelt a szoftvertermék.

6. lépés: Visszacsatolás

- Számszerűsítése a biztonságos szoftverfejlesztési folyamat sikerességének.
- Mérési lehetőségek forráskód alapján:
  - Hogyan változik a talált sérülékenységek száma?
  - Sérülékenység-típusok összetétele hogyan változik?
  - A kódbázis mekkora része auditált? Ez az arány idővel hogyan módosul?
- Mérési lehetőség forráskód nélkül:
  - Milyen hosszú ideig tart a különböző prioritású problémák elhárítása? (Audittól a hibajavításig eltelt idő.)

„Összegzés”

- A szoftver biztonságossága a fejlesztésben résztvevők biztonság-tudatosságától függ, a tervezéstől a megvalósításon át a tesztelésig.
- Több szem többet lát.
- Nincs teljesen biztonságos szoftver.