Kódolás és IT biztonság (VIHIBB01)
Laboratóriumi gyakorlat

Programok visszafejtése

Szerző:
Ládi Gergő

Utoljára frissült: 2020. október 25.

Tartalomjegyzék

1. A labor célja

A labor során megismerkedhetsz a programok visszafejtésének alapjaival, majd néhány egyszerűbb visszafejtési technikát a gyakorlatban is kipróbálhatsz egy erre a célra készített mintaalkalmazáson. A gyakorlat célja, hogy bemutassuk, hogyan lehetséges egy lefordított bináris (alkalmazás) működését elemezni, megérteni és befolyásolni.

Minden hallgató kapni fog egy részben obfuszkált, C# nyelven megírt mintaalkalmazást, amely egy rosszindulatú program (malware) működését utánozza, továbbá mindenki számára elérhető lesz a dnSpy nevű program is, amely egy ingyenes és nyílt forráskódú debugger (hibakereső) és decompiler (visszafejtő) .NET nyelveken megírt alkalmazásokhoz. A feladatod az lesz, hogy megismerkedj a dnSpy visszafejtőeszközzel, segítségével megértsd a mintaalkalmazás működését, majd végül nyomon kövesd és módosítsd annak működését.

A gyakorlat elvégzése után képes leszel önállóan is népszerű visszafejtőeszközöket használni .NET-es alkalmazások megértésére és módosítására.

2. Elméleti háttér

2.1. Reverse engineering

A reverse engineering (magyarul kb. visszafejtés) az a folyamat, melynek során egy tárgyat vagy rendszert szétszedünk, elemeire bontunk, hogy jobban megérthessük, milyen alkatrészekből áll, hogyan tervezték, miként rakták össze. A szoftverek esetében a visszafejtés arra a folyamatra utal, amely során egy már kész szoftvert, egy lefordított binárist kézi vagy gépi módon, az utasítások értelmezésével megpróbálunk visszafejteni, azaz megpróbáljuk megérteni a lefordított kód működését, azt, hogy milyen célból írták a programot, és hogy ezt a célját hogyan, milyen módon teljesíti (milyen algoritmusokat, adatstruktúrákat, tervezési mintákat használtak a fejlesztők, milyen fájlokat nyit meg, milyen IP-címekhez kapcsolódik, stb.).

A reverse engineeringnek számos felhasználási területe van:

- Biztonsági elemzők visszafejthetnek szoftvereket abból a célból, hogy sérülékenységeket kereshessenek azokban
- Fejlesztők visszafejthetnek programokat azért, hogy azokhoz javításokat készítsenek akkor is, ha már az eredeti forráskód nem elérhető

- Fejlesztők azért is visszafejthetnek programokat, hogy megértsék, milyen interfészeket és adatstruktúrákat használnak, hogy utána készíthessenek olyan más szoftvereket, melyek képesek kommunikálni a visszafejtettekkel
- Malware-elemzők visszafejthetnek malware-mintákat azért, hogy rálátást szerezzenek, hogyan terjednek, egyáltalán mit csinálnak az általuk megfertőzött gépeken, de a visszafejtés abban is segíthet, hogy olyan detekciós módszereket készítsenek, amelyekkel még időben fel lehet ismerni a rosszindulatú kódot
- Crackerek visszafejthetnek szoftvereket abból a célból, hogy kiiktassák, megkerüljék a különféle másolásvédelmeket, licencellenőrzéseket (ez törvényellenes)
- Ipari kémek visszafejthetik a versenytársak szoftvereit, hogy megszerezzék az azokban használt titkos és értékes algoritmusokat (ez szintén nem legális)

(A lista nem feltétlenül teljes.)

2.2. Obfuszkáció

Az obfuszkáció az a műveletsor, amelynek során különböző ellenintézkedéseket végzünk abból a célból, hogy minél nehezebbé és időigényesebbé tegyük a szoftvereink visszafejtését, hogy ezzel elriasszuk a potenciális visszafejtőket. Az obfuszkációt nem kézzel, hanem erre szolgáló speciális szoftverekkel, obfuszkátorokkal végezzük. Ezek általában nyelvspecifikusak és fizetősek. Bár léteznek ingyenes változatok is, azok általában jóval kevesebb tudással rendelkeznek mint a fizetős társaik.

Néhány obfuszkációs technika:

- Névobfuszkáció (name obfuscation) – típusok és változónevek átnevezése értelmezhetetlen vagy nehezen olvasható nevekre
  (pl.: NetworkConnection.Initialize() → A.a())
- Sztringobfuszkáció (string obfuscation) – karakterláncok átalakítása úgy, hogy nehezebben lehessen őket megtalálni, megérteni
  (pl.: "Hello" → "\u0048\u0065\u006c\u006c\u006f")
- Sztringtitkosítás (string encryption) – a sztringeket titkosítva tároljuk, és csak szükség esetén dekódoljuk őket

- Értékobfuszkáció (value obfuscation) – numerikus konstansok kicserélése formulákra (keresést, átírást nehezíti)
  (pl.: $42 \rightarrow 2 * 20 + 100/50$)
- Erőforráscsomagolás és -tömörítés (resource packing and compression) – a program által használt különböző erőforrások (pl. hangok, képek, videók, 3D modellek) összecsomagolása, tömörítése, esetleg beágyazása az alkalmazásba abból a célból, hogy nehezebben lehessen azokat módosítani, ellopni
- Erőforrástitkosítás (resource encryption) – az erőforrások titkosítása, menet közbeni, igény szerinti dekódolással
- Vezérlésfolyam-obfuszkáció (control flow obfuscation) – függvények feldarabolása, rengeteg *goto* beszúrása, soha nem teljesülő feltételes ágak beszúrása (a logika követését, megértését nehezíti)
- Vezérlésfolyam-kilapítás (control flow flattening) – az alkalmazás teljes logikájának átalakítása egyetlen egy nagy állapotgéppé

Ezen kívül léteznek még más, bonyolultabb módszerek is, mint például a kódtitkosítás (code encryption) vagy a kódvirtualizáció (code virtualization), amelyeket tipikusan a drágább számítógépes játékok esetében alkalmaznak.

2.3. Alacsony és magas szintű nyelvek

A programnyelvek két nagy kategóriába sorolhatók aszerint, hogy mennyire fedik el a processzorok és a kezelt perifériák különbözőségét valamint a natív erőforrásokat a fejlesztőtől, továbbá mennyire fejlett a nyelvi eszköztáruk. Ez a két kategória az alacsony és a magas szint. Hagyományosan alacsony szintűnek vesszük a gépi kódot és az Assemblyt, de egyes frissebb szakirodalmak azon nyelveket is alacsony szintűnek tekintik, melyekben a memória és a natív erőforrások (fájlleírók, hálózati socketek) kezelése nem automatikus, vagy nincsenek futásidejű ellenőrzések (például tömbök túlindexelése ellen).

A magas szintű nyelvek közé tartoznak az interpretált nyelvek (szkriptnyelvek, mint például a JavaScript vagy a Python) és a menedzselt nyelvek (mint például a Java vagy a C#) is.

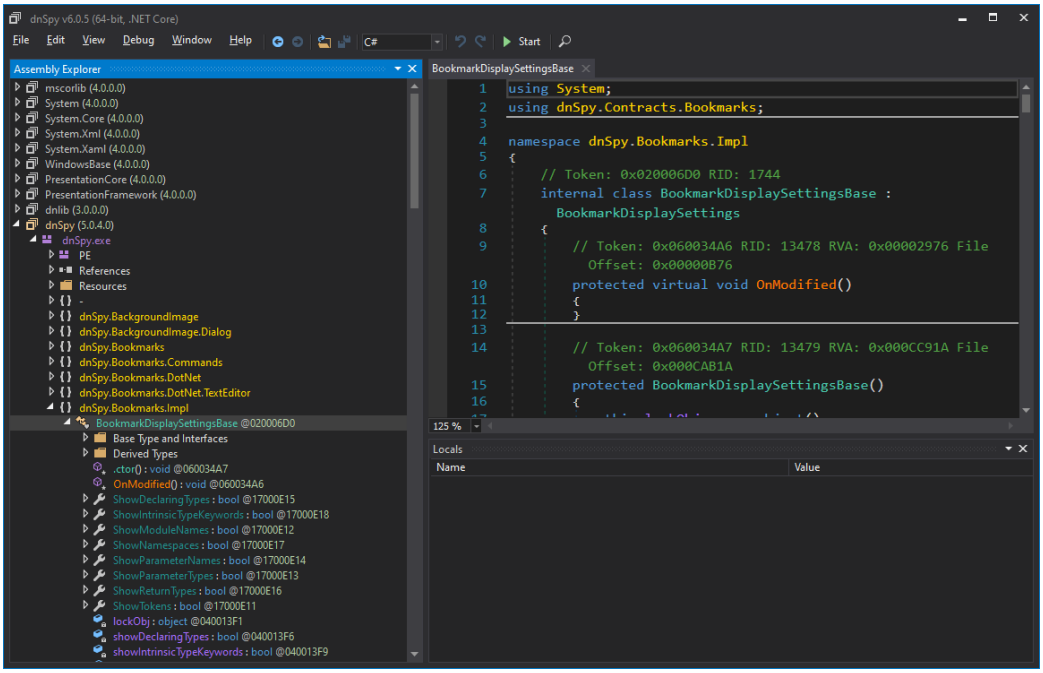
2.3.1. Menedzselt nyelvek

A menedzselt nyelvek esetében a fordítók úgynevezett köztes kódot (Intermediate Language, IL) generálnak natív binárisok helyett. Amikor az alkal-

mazást elindítjuk, egy futásidejű fordító (just-in-time compiler, JIT) felolvassa a köztes kódot, menet közben lefordítja (és optimalizálja) azt az éppen aktuális számítógépben lévő processzoron futtatható kóddá, majd végül futtatja is azt. Ettől lesznek (lehetnek) a Java és .NET alkalmazások platformfüggetlenek. A köztes kódnak viszont megvan az a hátránya, hogy sokkal egyszerűbb felépítésű és rengeteg metaadatot tartalmaz (például típusinformációk, adatstruktúrák, sőt, változónevek) a natív binárisokhoz képest, emiatt pedig általánosságban véve sokkal könnyebb is egy menedzselt nyelven megírt alkalmazást visszafejteni, mint egy olyat, amely egy nem menedzselt vagy egy alacsony szintű nyelven íródott.

3. Felhasznált eszközök

3.1. dnSpy



A dnSpy egy ingyenes és nyílt forráskódú debugger (hibakereső) és decompiler (visszafejtő), mely letölthető a GitHub oldaláról[1]. A program felhasználói felülete hasonlít a Visual Studio felületére. Belehúzhatunk .NET

[1] https://github.com/0xd4d/dnSpy

szerelvényeket (assemblyket[2]), melyeket a program betölt és visszafejt. A Visual Studiohoz hasonlóan van egy Assembly Explorer ablaka, melyen keresztül megtekinthetjük, milyen névterek, osztályok, stb. találhatók a betöltött szerelvényekben. Az osztályokat kiválasztva megtekinthetők az azokban található tagváltozók, metódusok. Ha a szerelvény futtatható jellegű (például nem csak egy függvénykönyvtár), akkor a szerkesztőben elhelyezhetünk töréspontokat is, majd el is indíthatjuk az alkalmazást. A visszafejtett, futtatandó kódot szerkeszteni is tudjuk: meglévő sorokat törölhetünk ki, újakat vehetünk fel, de akár teljesen új metódusokat, osztályokat is felvehetünk.

Fontos: a szerelvény szerkesztése után, futtatás előtt mindig mentsük el a módosításokat (File → Save Module...)! Ha ezt elfelejtjük, az átírt program helyett az előző mentett (akár az eredeti) változat fog elindulni.

3.2. A mintaalkalmazás

A mintaalkalmazás egy .NET-ben megírt, botnet klienst imitáló program. Bár az alkalmazás a működését, viselkedését tekintve malware-szerű, azt egyedileg, a labor során történő elemzés céljából fejlesztettük ki, így ~~jobbára~~ ártalmatlan, tehát akár a saját gépeden is dolgozhatsz vele, le is futtathatod. Hogy ne legyen túlságosan egyszerű az elemzés, pár alapvető obfuszkációs technikát is alkalmaztunk a programon. A program a kari Moodle-ből[3] tölthető le.

4. Feladatok

4.1. Vezetett rész

A labor elején közösen megismerkedünk a dnSpy eszközzel, annak főbb funkcióival. Betöltjük az elemzendő alkalmazást, majd az Assembly Explorer segítségével megkeressük a Program osztályt a szerelvényben. A Main függvényből az Edit Method funkciót használva kiszedjük a program indulásakori

[2]Itt és innentől kezdve az 'assembly' szó a .NET fordítási egységét jelöli, és nem pedig az ugyanilyen nevű, alacsony szintű nyelvet.

[3]https://edu.vik.bme.hu/

véletlen ideig történő várakozást. A módosításokat visszavezetjük az alkalmazásba. Megnézzük továbbá a breakpointok használatát, valamint a B osztálynak beszédesebb nevet adunk.

Részletes leírás

- Töltsük le az elemzendő fájlt a Moodle-ből. Ezt érdemes lehet a labor előtt, "otthon" megcsinálni.
- Töltsük le és indítsuk el a dnSpy alkalmazást[4]. Akinek van fent .NET Framework 4.7.2, az használhatja a dnSpy-net472.zip-et, egyébként ajánlom a dnSpy-netcore-win64.zip fájlt letölteni. Ezt is érdemes a labor előtt megcsinálni.
- Nyissuk meg az elemzendő fájlt. Ezt vagy drag-and-drop módon ("belehúzással") vagy a File → Open... menüből lehetséges megtenni.
- Bal oldalon az Assembly Explorerben keressük meg az újonnan betöltött szerelvényt, majd nyitogassuk ki a leveleit egészen addig, amíg nem látszik a Program osztály.
- Kattintsunk a Program osztályra. Jobb oldalon megjelenik a visszafejtett kód. Ez itt visszafejtett kód, és nem biztos, hogy 1:1 ugyanaz, amit a programozó leírt, viszont funkcionálisan ekvivalens azzal.
- A kódban keressük meg a Main függvényt, majd jobb kattintva a függvényen belül válasszuk az Edit Method opciót.
- Keressük meg azt a sort a 15-16. sorok környékén, ami hasonlóan néz ki: int valami = (900 + new Random().Next(-300, 300)) * 6000;
- Töröljük ki ezt a sort, és alatta a Thread.Sleep(valami) sort is. Így már indítás után nem fog várakozni az alkalmazás, és élőben is tudjuk majd vizsgálni, mit csinál. Malware-ek pont ezért szoktak néha várakozni, mielőtt valamit csinálnak, hogy nehezebb legyen elemezni őket.
- Kattintsunk a Compile gombra. Ezzel a módosításaink alapján elkészül az új IL kód, ami még csak a memóriában létezik.
- Ahhoz, hogy debuggolni is lehessen a módosításainkat, el kell menteni az alkalmazást. Ezt a File → Save Module... menüponttal lehetséges. Nyugodtan írjuk felül az előző verziót.
- Tegyünk egy breakpointot az IAsyncResult kezdetű (hálózati kapcsolatot kezdeményező) sorra (ez a 34. sor környékén lesz). Breakpointot tenni a sor számától balra látható szürkésebb csíkba kattintva vagy a

[4]https://github.com/0xd4d/dnSpy/releases

soron jobb gombbal kattintva és az Add Breakpoint opciót választva lehetséges. Egy piros teli körnek kell megjelennie a csíkban.

- Ebben a sorban a BeginConnect függvény kezdeményezi a kapcsolatot, és az első két paramétere a cél hosztneve és a port száma. (Az egeret a függvény felé húzva ki is írja ezt a dnSpy.)
- Indítsuk el az alkalmazást. Rövidesen meg kell állnia a breakpointon. Ilyenkor a piros körben sárga nyíl jelenik meg. Sajnos néha bugos lesz a dnSpy, és nem működnek a breakpointok (mintha nem a valós kóddal dolgozna). Ilyenkor piros x jelenik meg a kör sarkában, jelezve, hogy itt nem fog tudni megállni. Állítsuk le a debuggolást, az Assembly Explorerből szedjük ki a szerelvényt (jobb katt, Remove), húzzuk be újra, állítsuk be ismét a breakpointot, majd indítsuk el. Most jó lesz.
- Alul a Locals nézetben láthatók az aktuális szkópban elérhető változók. Láthatjuk, hogy az array2 (v. hasonló nevű) tömb 0. indexű eleméből szedi a hosztnevet, viszont ennek az értéke most elég zagyvaság (renjynz.ynobe.rknzcyr.ybpny). Ez azért van, mert csatlakozáskor erre még ráhívja az A osztály a függvényét.
- A zöld háromszöges Continue gombbal tudjuk folytatni a program futtatását.
- Állítsuk meg a futást.
- Szervezzük ki a csatlakozáshoz használt hosztnevet egy külön változóba, hogy meg is tudjuk nézni az értékét. Ezt az előbbiekben ismertetett szerkesztővel tudjuk megtenni. Az IAsyncResult kezdetű sor elé szúrjunk be egy új sort: string hostname = A.a(array2[0]);
- Az IAsyncResult kezdetű sorban az A.a(array2[0]) paramétert cseréljük ki arra, hogy "hostname".
- Mentsük a változtatásainkat.
- Tegyünk a sorra breakpointot, majd indítsuk el a programot.
- Meg fog állni a breakpointnál. A Local ablakban a hostname változó értékét leolvasva látszik, hogy elsőként az "erawlam.labor.example.local" címre próbál csatlakozni. Ez nem fog sikerülni, így próbálkozik tovább.
- Folytassuk a futtatást. Következő körben a [törölve] címhez próbál majd csatlakozni. Ide sikerül is csatlakoznia, így amíg a kapcsolat meg nem szakad, többet nem fogunk megállni a breakpointon. Ez a Moodleben kitöltendő vezetett feladat megoldása.

- A végső lépések egyikeként keressük meg a B osztályt. Ennek összesen egy b nevű, Socket típusú adattagja van. Ebből gondolhatjuk, hogy ez az osztály egy hálózati kapcsolatot reprezentál.
- Nevezzük át a B osztályt valami beszédesebbre, pl. NetworkConnection. Ezt az osztályba beállva, majd az Edit menüből az Edit Type opciót választva tehetjük meg.
- Az új NetworkConnection osztályunkban van egy b nevű, Socket típusú tagváltozó. Ennek is adjunk értelmesebb nevet (pl. socket). Kattintsunk jobb gombbal az "internal Socket b" sorra, majd kattintsunk az Edit Property menüpontra. A b nevet írjuk át socketre. Nem szükséges túlzásba vinni az átnevezgetést, ne ezzel menjen el az összes idő. A gyakorlatban sem szokás mindent átnevezni, csak addig a szintig, amikor már anélkül is érthető, hogy mi történik a kódban.
- Mentsük a módosításokat. Kész vagyunk a vezetett résszel :)!

4.2. Önálló rész

Az önálló rész teljesítéséhez az obfuszkált kód további részeit kell visszafejteni, megérteni, esetleg módosítani. A feladatokat ajánlott sorban megoldani, egyrészt azért, mert a feladatok egyre nehezednek, másrészt azért, mert logikailag úgy függnek össze – egy korábbi feladat megoldása segítség lehet egy későbbi feladat megoldásához.

4.2.1. 1. feladat – Védett(?) kommunikáció

Keresd meg azt a kódrészletet, ahol a malware parancsokat olvas be a vezérlőszerverétől, azaz a hálózatról fogad adatokat. Milyen kódolást, titkosítást alkalmaz a kommunikáció során (ha egyáltalán alkalmaz ilyet)?

4.2.2. 2. feladat – Működés vizsgálata I.

Mit csinálna a malware, ha a vezérlőszervertől 'LLIK' parancsot kapna?

4.2.3. 3. feladat – Működés vizsgálata II.

Mit csinálna a malware, ha a vezérlőszervertől 'XELD' parancsot kapna?

4.2.4. 4. feladat – Működés vizsgálata III.

Csatlakozás után a szerver rendszeresen (néhány másodpercenként) leküld egy parancsot. Melyik ez?

4.2.5. 5. feladat – Ókori módszerek?

Vizsgáld meg a vezérlőszerverek listáját kititkosító *A* osztály *a* függvényét! Milyen ismert algoritmust valósít meg a függvény?

4.2.6. 6. feladat – Updates are ready to install

Kaptunk egy fülest, mely szerint a most vizsgált malware hasonlít egy már korábban látottra. Annak a bizonyos másik malware-nek a vezérlőszerverei, ha kaptak egy 'REVL' parancsot, visszaválaszoltak a malware legfrissebb verziószámával (ami alapján aztán a malware el tudta dönteni, hogy van belőle újabb verzió, ideje frissíteni magát). Derítsd ki, ismeri-e a most vizsgált malware **C&C szervere** a 'REVL' parancsot, és ha igen, mit válaszol rá?