

Biztonsági tesztelés forráskód alapú hibainjektálással

TÓTH GERGELY, HORNÁK ZOLTÁN

SEARCH-LAB Kft.

{gergely.toth, zoltan.hornak}@search-lab.hu

Lektorált

Kulcsszavak: biztonsági tesztelés, hibainjektálás, forráskód alapú tesztelés

Az informatikai rendszerekben megbújó kihasználható biztonsági szoftverhibák hatalmas károkat okoznak világszerte. A bemutatásra kerülő biztonsági hibákat hibainjektálás alapú teszteléssel felderítő módszer célja, hogy költséghatékony alternatívát nyújtson a jelenlegi drága biztonság-növelő lehetőségeknek, a formális validálásnak, illetve a kimerítő tesztelésnek. Az új módszer lényege, hogy a használt adatstruktúrák leírójára támaszkodva képes generikus teszt algoritmusokat futtatni, melyek közvetlenül a tesztelni kívánt kódrészletekbe injektálnak automatizáltan generált teszt vektorokat, majd képes megfigyelni, hogy ezen szélsőséges esetekben a vizsgált rendszer le tudja-e kezelni a hibát, vagy biztonsági hibához hasonló reakciót vált-e az ki.

1. Bevezetés

A programozói hibákat három csoportra oszthatjuk:

- (1) általános, *funkcionalitást befolyásoló hibák*;
- (2) *biztonsági szempontból veszélyes hibák*, melyek bizonyos esetekben azt okozhatják, hogy az adott rendszer biztonsági tulajdonságai sérülnek (de ebben a kategóriában még nem feltétlenül kihasználható a hiba, csupán fennáll annak veszélye);
- (3) végül pedig *kihasználható biztonsági lyukak*, melyeken keresztül közvetlen támadás intézhető a rendszer ellen.

A gyakorlatban akkor nevezhetünk egy rendszert biztonságosnak, ha nincs benne kihasználható biztonsági lyuk. Ennek bizonyításához azonban komplex formális módszerek vagy kimerítő tesztelés szükséges.

Ha azonban nem csak közvetlenül a kihasználható biztonsági lyukak felderítésére koncentrálunk, hanem azok okait, a biztonsági szempontból veszélyes hibákat kívánjunk felfedezni és kiküszöbölni, akkor – bár látszólag szélesebb körű tesztelést kell végrehajtani – költséghatékonyabban aránylag magas fokú biztonságot tudunk elérni.

A leggyakoribb biztonsági szempontból veszélyes hiba típusokhoz ugyanis léteznek algoritmusok, melyek ezeket költséghatékonyan, mégis nagy megbízhatósággal detektálni tudják és a legtöbb felhasználási területen az így elérhető biztonsági szint is lényeges javulást eredményezne.

A bemutatásra kerülő módszer – illetve a *Flinder* keretrendszer – lényege, hogy a tesztelendő programban (Target of Evaluation, ToE) felkutassa a biztonsági szempontból veszélyes hibákat, mielőtt azok kihasználható biztonsági lyukakká válnának. A *Flinder* erre két lehetőséget biztosít: az úgynevezett black-box (amikor a forráskód nem ismert) és a forráskód alapú tesztelést.

1.1. Black-box tesztelés

Black-box tesztelés során a *Flinder* a ToE és az úgynevezett Input Generátor (IG) eszköz közötti (például egy kliens-szerver alkalmazás esetén a kliens és a szerver közötti) üzeneteket manipulálja oly módon, hogy az algoritmikusan módosított teszt vektorok minél nagyobb valószínűséggel okozzanak anomáliát (például lefagyást vagy védelmi hibát) a ToE-ben, amit utána detektálni lehet és így fény derülhet a biztonsági hibára.

A *Flinder* az üzenetmódosítást a következőképpen végzi: beépül a tesztelendő program (ToE) és az azt input üzenetekkel működésre bíró Input Generátor (IG) közé, majd megfigyeli és manipulálja a közöttük folyó kommunikációt. Miután a ToE és IG közötti üzeneteket a *Flinder* elkapta, szükség van a nyers bináris formátum értelmezésére, egységes formátumra való transzformációjára. A *Flinder értelmező* (parser) moduljának feladata, hogy a bináris üzenetet egy belső, általános fa struktúrába, úgynevezett MSDL-be (Message Structure Description Language) konvertálja. Ehhez egy *formátum leíróra* van szüksége, ami a bináris üzenetek formátumát specifikálja, ez pedig az MFDL (Message Format Description Language) követelményeinek megfelelő XML dokumentum (egy példát az 2. ábrán mutatunk be).

Ezek után a konkrét teszt vektort a megfelelő algoritmus az MSDL módosításával generálja, amit a *szerializáló* modul (az értelmező ellentettje) transzformál vissza bináris formába, ami így jut el a címzetthez, a ToE-hez. A tesztelés és a kommunikáció során használt protokoll állapotát a *Flinder egy állapotgépben* tárolja, melynek segítségével nem csak egy kérdés-válasz jellegű kommunikációt, hanem több üzenetváltást is magába foglaló protokollt is értelmezni tud. Fontos megjegyezni, hogy az MSDL általános volta miatt a konkrét bináris formátumtól függetlenül generikus tesztelő algoritmusok futhatnak (így tehát ugyanaz az algoritmus tud futni a DER kódolású üzenetek tesztelésénél és a szöveg alapú HTTP-nél is).

1.2. Forráskód alapú tesztelés

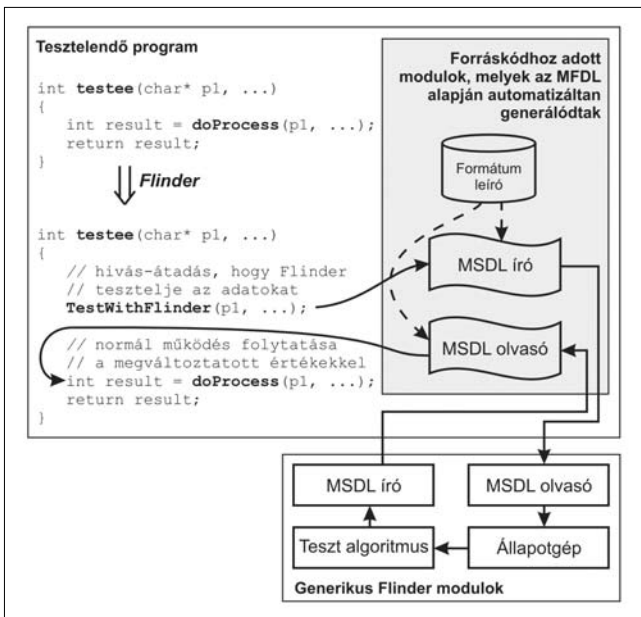
Forráskód alapú tesztelés során a cél, hogy a tesztelő algoritmusok a teszt vektorokat a program belső függvényeibe (például mint függvényparaméterek) tudják bejuttatni, a mi szóhasználatunkban *injektálni*. Ehhez a munkamódszer adaptálásra szorul: az architektúrát úgy kell módosítani (1. ábra), hogy a tesztelni kívánt programhoz hozzá kell fordítani az MFDL formátum leíró alapján automatizáltan generált forráskódú értelmezőt (MSDL író) és szerializálót (MSDL olvasó), melyek képesek a Flinder számára a tesztelendő adatstruktúrákat MSDL formátumba átírni (MSDL író), majd a teszt vektort a Flindertől fogadva az adott nyelv belső adatstruktúrájának megfelelően visszafordítani (MSDL olvasó). Maga a teszt vektor előállítás is általában: eredeti MSDL-ből kell az algoritmikusan generált teszt MSDL-t előállítani.

A black-box tesztelés menetét és a felmerült problémákat [1]-ben publikált írásunk elemzi, ebben a cikkben a forráskód alapú tesztelésre fókuszálunk, mivel ez integrálható legjobban a szoftverfejlesztési életciklusba. Itt kell megemlíteni, hogy a Flinder tervezésénél fontos szempont volt, hogy minél több modult tudjunk mind black-box, mind pedig forráskód alapú tesztelésnél is használni – ezért is hasonlít a két megközelítés architektúrája.

2. A Flinder fő tulajdonságai

A Flinder fő célja a biztonsági szempontból veszélyes hibák felderítése függetlenül attól, hogy azok kihasználhatók-e, vagy sem. Erre a feladatra számos módszer és termék létezik már, azonban a Flinder több újdonsággal is rendelkezik, melyek automatizmusokkal és új megközelítésekkel csökkentik a tesztelők által végrehajtandó munka mennyiségét:

1. ábra Forráskód alapú hibainjektálás Flinderrel



- *Formátum leíró alapú tesztelés:*

A Flinder által alkalmazott módszer lényege, hogy a teszt vektorok előállításához csak a formátum leírot használja, melyet más leírónyelvekből (pl. XML Schema [4]) akár automatizáltan is elő lehet állítani. Bár az MFDL pontos specifikációját terjedelmi okokból ez a cikk nem tartalmazza, egy könnyen megérthető példát az 2. ábra tartalmaz.

A már meglévő módszerekkel szemben a nem megfelelő működés felismeréséhez a Flinder nem igényli a helyes működés részletes specifikációját – a rendszer képes a teszt vektorokat helyes bemenetek és a formátum leírok alapján generálni.

- *Generikus, cserélhető teszt algoritmusok:*

Az architektúra további előnye, hogy a tesztelő algoritmusok generikus adatstruktúrára, az MSDL-en dolgoznak, így nem szükséges azok adaptációja az éppen tesztelni kívánt környezethez, ezek az algoritmusok úgynevezett *plug-in* rendszerben működnek a Flinderen belül. Fontos hangsúlyozni, hogy számos gyakori biztonsági hiba osztályhoz lehet ilyen általános tesztelő algoritmust készíteni (puffertúlcsordulás, egész szám túlcsordulás, vagy kódolási hibák detektálására).

- *Állapotgép:*

Megemlítendő még, hogy a Flinder bonyolultabb tesztek futtatásának segítéséhez egy úgynevezett *protokoll állapotgépet* is futtat, mely az éppen tesztelni kívánt forgatókönyv *UML state machine* [2] alapú leírását tartalmazza. Ily módon a rendszer képes komplex folyamatok illetve protokollok követésére is. Forráskód alapú tesztelésnél ez az állapotgép használható akár a program futásának nyomon követésére is.

- *Kriptogrammok támogatása:*

Flinder annak érdekében, hogy kriptográfiai eszközöket felvonultató protokollok implementációját is tesztelni tudja (pl. SSL) támogatja a különböző kriptográfiai primitíveket (lenyomatkészítés, titkosítás, digitális aláírás stb.). Az MFDL megfelelő kialakításával az értelmező képes például egy eredetileg titkosított üzenetet először dekódolni, majd tovább értelmezni (természetesen a szerializáló is támogatja ezeket a műveleteket), így akár egy teljesen rejtjelezett protokoll implementációja is tesztelhető a Flinderrel.

3. Forráskód alapú tesztelés Flinderrel

A Flinder általános ismertetése után most következnek a konkrét tesztelési módszer ismertetése:

1. Első lépésként el kell készíteni a tesztelendő adatstruktúráknak megfelelő formátum leírot. Egy egyszerű példát C nyelvű adatstruktúrákhoz a 2. ábrán láthatunk (a következő oldalon).

2. Ezután az MFDL alapján a Flinder olyan függvények forráskódját generálja, melyek képesek az MFDL-nek megfelelő adatstruktúra illetve objektum példányokat MSDL-lé konvertálni, illetve MSDL alapján egy konkrét példány változóit értékekkel feltölteni.

3. A generált forráskódot a tesztelendő programhoz fordítva előáll ezeket a belső kapcsolódási lehetőségeket (úgynevezett *hook*-okat) tartalmazó ToE, melyet a Flinder különböző konfigurációkkal fog futtatni.

4. A Flinder futtatja a ToE-t, majd a megfelelő pontokon a tesztelő algoritmus módosítja a generált MSDL-t. A teszt vektor visszainjektálása után Flinder figyel a korrekt futást (nem terminálódik-e abnormálisan a ToE, az állapotgépnél megfelelő állapotba kerül-e a tesztelt program stb.)

Ezt a lépést a Flinder többször is végrehajthatja, amennyiben egy többlépéses, úgynevezett iteratív tesztelő algoritmust hajt végre.

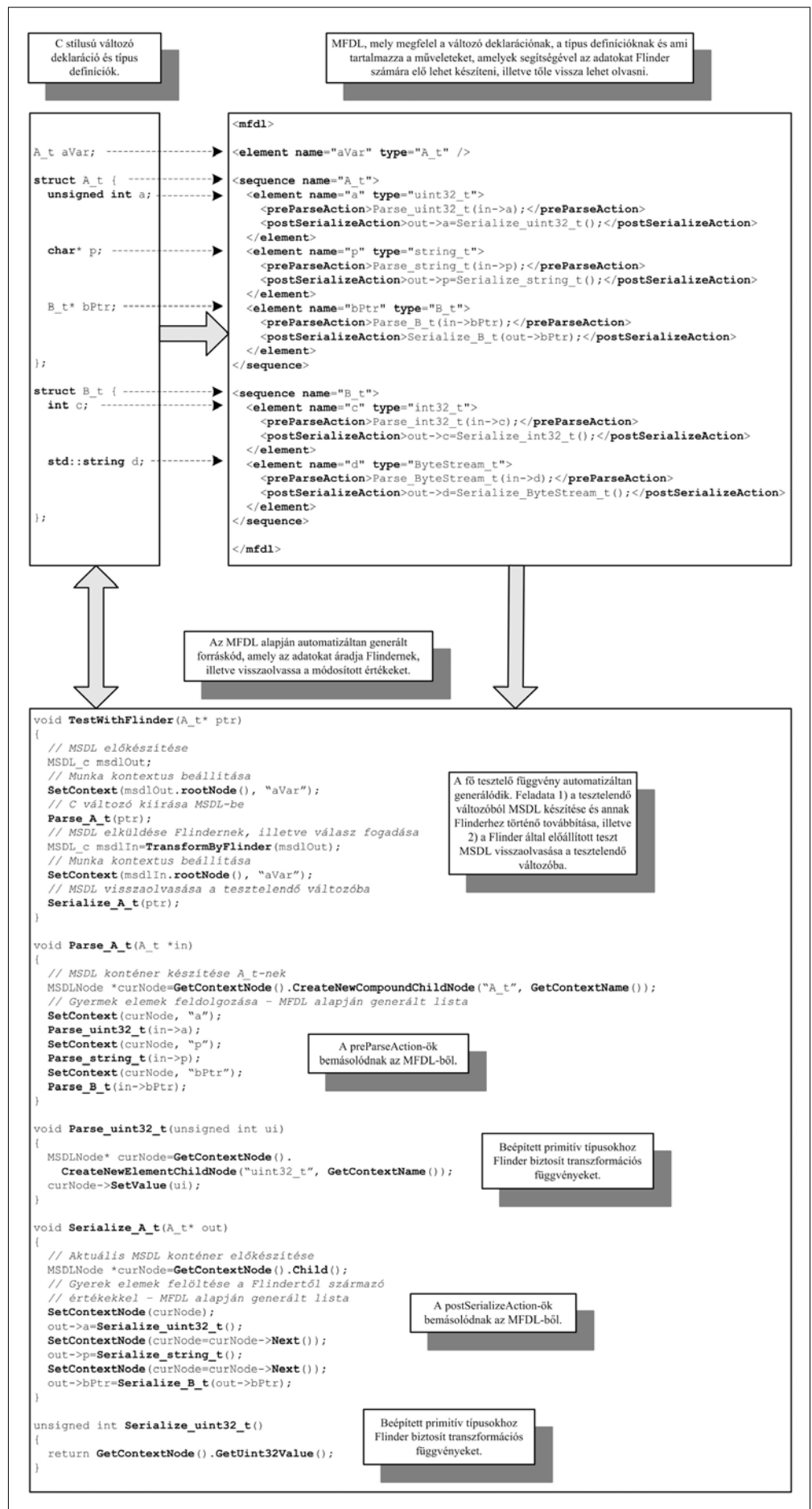
5. Végül a tesztelesek eredményeiről a Flinder jegyzőkönyvet készít, melyben részletesen összefoglalja a végzett tesztek, rögzíti a felhasznált teszt vektorokat és a ToE reakcióit.

3.1. MFDL példa

A 2. ábrán a C nyelvű adatstruktúra-MFDL megfeleltetés kerül bemutatásra, valamint részlet látható az MFDL alapján generált C nyelvű MSDL író és olvasó funkciókból.

Fontos megemlíteni, hogy bár az MFDL automatikusan generálható a típus definíciókból, a tesztelő számára megtartottuk annak lehetőségét, hogy az MSDL-változó konverziót befolyásolja.

2. ábra
Példa adatstruktúra, hozzá tartozó MFDL és generált kódrészlet



Erre szolgálnak az MFDL-ben elhelyezendő *preParseAction* és *postSerializeAction* mezők, melyek olyan, a forráskód nyelvének megfelelő kódrészleteket tartalmazhatnak, amik az adott mező konvertálását hivatottak elvégezni.

3.2. Teszt algoritmus példa

Az MFDL példát folytatva tekintsünk át egy egyszerű tesztelő algoritmust. Tételezzük fel, hogy a 3. ábrán látható tipikus, puffer túlcsoordulásos biztonsági hibát tartalmazó kódrészletet teszteljük a Flinderrel.

```
void ErroneousFunction(A_t *aPtr)
{
    // Flinder tesztelés
    TestWithFlinder(aPtr);

    // 10 bájt hosszú lokális tömb
    char locBuf[10];

    // Amennyiben aPtr->p hosszabb, mint 10 bájt,
    // úgy ez a másolás a hosszellenőrzés mellőzése
    // miatt felülírhatja a stacken tárolt függvényt
    // visszatérési címet, ami biztonsági hibát
    // eredményez.
    strcpy(locBuf, aPtr->p);
}
```

3. ábra Puffertúlcsoordulásos hibát tartalmazó függvény

Amennyiben a fenti függvény futtatása során az *A_t* típusú struktúra *p* mezője 10 bájtól hosszabb sztringre mutat, úgy a másolás túl fogja írni a lokális fix méretű tömböt. A túlírás adott esetben elérheti a veremben tárolt vezérlési adatstruktúrákat is, sőt akár a függvény visszatérési címét is módosíthatja. Ily módon egy támadó eltérítheti a program futását a függvényből való visszatéréskor. Ha pedig nem támadó szándékkal véletlenszerű értékkel íródik felül ez a terület, akkor a program nagy valószínűséggel védelmi hibát fog okozni. (A puffertúlcsoordulásos hibák veszélyéről és kihasználásuk módszeréről [3] részletes áttekintést nyújt.)

A Flinder ezen hibatípus felderítésére egy egyszerű algoritmust használ: az MFDL alapján a kapott MSDL-ben megkeresi a változó méretű mezőket (a példánkban ezek az *A_t* típus *p* tagváltozója és a *B_t* típus *d* tagváltozója) és minden meghíváskor megduplázza azok méretét (így például *p* tartalma először „XX”, majd „XXXX” stb. lesz).

Ez az algoritmus gyorsan képes a programozók által feltételezett értéktartományon kívülre jutni és túlméretes értékeket generálni, amik a méretellenőrzés hiánya vagy hibája miatt védelmi hibát okozhatnak. A hibásan lekezelt vagy nem ellenőrzött méretkorlátokat így a Flinder már detektálni tudja, és fény derülhet a biztonsági hibára. Gyorsítási lehetőség még, ha a különböző mezőket egyszerre nyújtjuk, de opció lehet a mezők soros tesztelése is.

Fontos megjegyezni, hogy mivel a tesztalgoritmusok (ebben az esetben a puffertúlcsoordulásos hibákat kereső algoritmus) az általános MSDL-en dolgoznak, ugyanaz az algoritmus alkalmazható függetlenül attól,

hogy mi is volt a konkrét bemenet. Terjedelmi okokból további algoritmusok ismertetését mellőzzük, de hasonló elven számos gyakori típushibára készíthető effektív algoritmus, melyek együttes alkalmazása lényegesen javíthatja a tesztelt programok biztonsági tulajdonságait.

4. Összefoglalás

A cikkben biztonsági hibák automatizált felderítésére mutattuk be a Flinder keretrendszert, mely forráskód alapú hibainjektáláson alapul.

A módszer lényege, hogy egy állapotgéppel követett programfutás közben generikus tesztelő algoritmusok képesek egy univerzális formátum leíró alapján a program belső függvényeibe teszt vektorokat injektálni, melyek célja a biztonsági hibák miatti abnormális működés előidézése, amit aztán a Flinder detektálni tud. A Flinder újdonsága, hogy a teszteléshez nem szükséges a helyes működés formális specifikációja, a tesztelőnek helyes bemeneteket, a kommunikáció állapotgépét (UML) és egy XML-alapú formátumleíró kell csak elkészítenie. Ezek alapján már számos gyakori biztonsági hiba típusra automatizáltan végezhető el a tesztelés.

Köszönetnyilvánítás

A projektet a Gazdasági Versenyképesség Operatív Program támogatta (GVOP-3.3.1-2004-04- 0094/3.0).

Irodalom

- [1] Tóth G., Hornák Z.:
Semi-automated detection of security-relevant programming bugs with Flinder, 2006.
www.flinder.hu
- [2] Object Management Group:
UML – Unified Modeling Language, 2005.
- [3] Aleph One: Smashing the stack for fun and profit,
Phrack 7, 1996.
- [4] W3C – World Wide Web Consortium:
XML Schema, 2004.