Diplomaterv

Szekeres László

2007.

DIPLOMATERV-FELADAT

MELLÉKLET – FELADATKIÍRÁS

Nyilatkozat

Alulírott, *Szekeres László*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Szekeres László
hallgató

Köszönetnyilvánítás

Szeretnék köszönetet mondani szüleimnek, akik tanulmányaimat lehetővé tették és támogattak a pályaválasztásom során.

Köszönöm a konzulenseimnek, Hornák Zoltánnak és Tóth Gergelynek, akik tapasztalatukkal, tanácsaikkal segítették és irányították munkámat.

Köszönöm Emőkének a türelméért.

Kivonat

A programfejlesztés mai technikája mellett a jelenlegi rendszerekben rendkívül sok olyan programozási hiba marad, amelyek a rendeltetésszerű használat során nagyon ritkán jelentkeznek. Azonban ezen ártalmatlannak tűnő, a hétköznapi működést legtöbbször nem is befolyásoló hibák egy rosszindulatú támadó számára gyakran olyan lehetőségeket rejtenek, amelyek segítségével könnyen visszaéléseket tud elkövetni. A probléma fontosságát és a veszély nagyságát csak növeli, hogy adott esetben a támadó számára egyetlen hiba megtalálása és kihasználása elegendő a védelmi eszközök megkerüléséhez és a rendszer feletti teljes irányítás átvételéhez. Mivel ezek a hibák rendkívül komoly veszélyt jelentenek a biztonságra, az ellenük való védekezés alapvető fontosságú.

A leggyakrabban előforduló tipikus biztonsági hibák bemutatása után, rendszerezve és csoportosítva azokat, a különböző védekezési módszereket tanulmányozza e diplomamunka. Rávilágít, hogy a hagyományos szoftverteszteléssel szemben a biztonsági tesztelés egy veszélyközpontú szemléletmódot igényel. Az ilyenfajta tesztelésnek a statikus kódelemzésen kívül a leggyakrabban alkalmzott módja a dinamikus „fuzz”-tesztelés.

A SEARCH Laboratórium keretein belül létrehoztunk egy „inteligens fuzzing”-ra képes keretrendszert, amellyel könnyebben és hatékonyabban lehet bizontsági tesztelést kivtelezni, mint a ma létező általános megoldásokkal. A rendszer gyakorlatban való alkalmazását médialejátszó alkamazások tesztelésén keresztül mutatom be.

TARTALOMJEGYZÉK

1 BEVEZETÉS

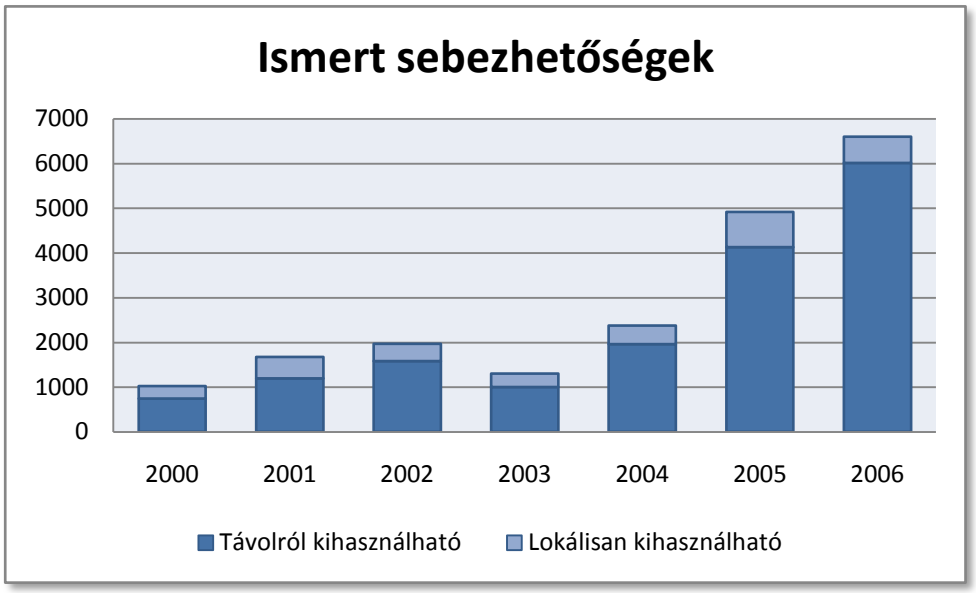
> *„Minden szoftver tartalmaz legalább egy változót, egy elágazást, egy ciklust és egy hibát.” – Murphy törvény*

Napjainkban az IT világ talán legégetőbb problémáját az informatikai biztonság helyzete jelenti. Óriási erőfeszítéssel dolgoznak a kutatók és fejlesztők a probléma enyhítésén, a helyzet érdemben mégsem javult sokat az évek folyamán, inkább csak átalakult. Rengetegféle megoldás létezik már, hogy adataink bizalmasságát és sértetlenségét megőrizzük, valamint hogy rendszereink rendelkezésre állását fenntartsuk. A kriptográfia, az operációs rendszerekbe ágyazott hozzáférés-védelmi mechanizmusok, a biztonsági protokollok mind ezeket szolgálják. Mégis nap mint nap találkozunk újabb vírusokkal, férgekkel, és segítségükkel kialakított, fertőzött, „zombi” számítógépek által terjesztett kéretlen levelekkel. Az informatikai rendszerekbe crackerek törnek be, adatokat lopnak, hamisítanak, és még sorolhatnánk. A legtöbb esetben mindezekért nem a rosszul tervezett, hanem a hibásan megvalósított rendszer, azon belül is elsősorban a rosszul implementált szoftver a felelős. Nem csak a mindennapi munkákhoz használt szövegszerkesztő, böngésző és levelező programokban fedeznek fel olyan biztonsági lyukakat, amelyeket kihasználva egy rosszindulatú támadó bizalmas információkhoz juthat, vagy akár egy számítógép felett átveheti az irányítást, hanem éppúgy a biztonsági politika betartásáért felelős operációs rendszerben, tűzfalban, vagy egy hálózatbiztonsági protokollvermet megvalósító szoftverben. Jogosan felmerülhet a kérdés, hogy például egy behatolás detektáló rendszer, vagy egy vírusirtó telepítése növeli, vagy – a benne potenciálisan megbújó biztonsági lyuk révén – éppen csökkenti a rendszer biztonságát [1].

1.1 HÁTTÉR

Az IT biztonságban a legnagyobb nehézséget az okozza, hogy biztonságos, hibamentes szoftvert írni rendkívül nehéz. Egyáltalán, mit jelent az, hogy biztonságos szoftver? Ha egy mondatban szeretnénk megfogalmazni, akkor azt mondhatnánk, hogy az a szoftver, ami azt teszi, amit szeretnénk tőle, hogy tegyen,

és – ami ugyanolyan fontos – semmi egyebet. A programfejlesztés mai technikái mellett ez sajnos nem tűnik megvalósíthatónak. Amíg a szoftvereket emberek tervezik és írják, akik nem tökéletesek, addig a szoftverek sem lesznek azok. A szoftverekben olyan hibák maradnak, amelyek sérülékennyé, sebezhetővé illetve támadhatóvá teszik azt. Hogy rálátást kapjunk arra, hogy mekkora veszélyt jelentenek ezek a sérülékenységek, statisztikát készítettem a NIST sebezhetőségi adatbázisát [2] felhasználva arról, hogy 2000-ig visszamenően hogyan alakult a talált sebezhetőségek száma, amelyek visszaélésekre adnak lehetőséget. Az 1. ábra ismerteti az eredményt.



1. ábra – Talált sebezhetőségek száma 2000-től 2006 végéig

Egyértelműen látható a növekvő tendencia, és figyelemreméltó, hogy 2004-ben és 2005-ben szinte pontosan a duplájára ugrott a publikált biztonsági rések száma a megelőző évekhez képest. Érdemes megfigyelni, hogy a talált hibák hány százalékban tesznek lehetővé távoli támadásokat. A diagram oszlopainak alsó, sötétebb része jelzi távolról is kihasználható hibákat, míg a felső a csak lokálisan kihasználhatókat. Az arány 70% és 90% között mozog az egyes években. A tavalyi évben volt a legmagasabb, ugyanis a közzétett hibák 91%-a távolról való támadást tett lehetővé.

Ma, hogy ha fény derül egy biztonsági hibára, akkor az mindaddig támadhatóvá teszi az érintett rendszereket, míg azt a hibát ki nem javítják, be nem foltozzák. *A*

rendszer védtelen egy hiba megtalálása és annak befoltozása közötti idő alatt. Hogy megbecsülhessük ennek az időtartamnak az átlagos hosszát, gondoljuk meg, hogy mi minden zajlik le e két esemény között. A hiba megtalálása után (attól függően, hogy a felfedező milyen mértékben szeretne visszaélni az információval) a hiba publikálásra kerül illetve a fejlesztőt értesítik. A fejlesztők ezután megvizsgálják a problémát, és kijavítják a hibát a programban. Ezután közzétesznek egy javítást, ami befoltozza a rést. A sebezhető szoftvert futtató számítógép felhasználója vagy a rendszergazda később ezt a javítást (patch) feltelepíti. Figyelembe véve a mindehhez szükséges időtartamot és a fenti statisztikát a talált hibák számát illetően, a mai Internetre csatlakozó rendszereinkről nyugodtan kijelenthetjük, hogy *állandó veszélynek vannak kitéve*.

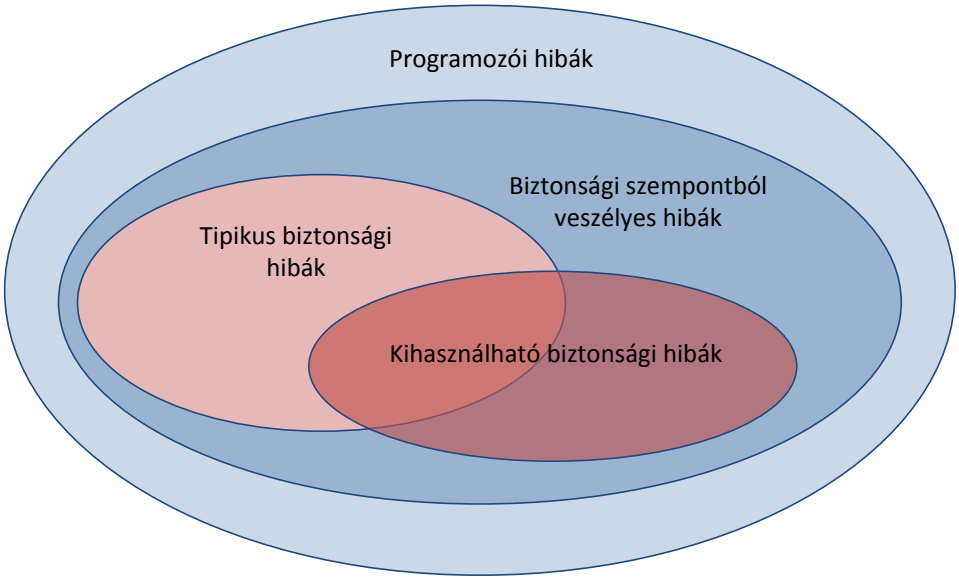
1.2 TERMINOLÓGIA

Az irodalomban nem teljesen egységes a különböző fogalmak használata, ezért a tisztánlátás érdekében mindenképpen fontos a jelen dolgozatban használt terminológia tisztázása. A következő alapfogalmakat érdemes definiálni:

- *Biztonsági követelmények*: meghatározzák a rendszer kívánt működését biztonsági szempontból (bizalmasság, sértetlenség, rendelkezésre állás).
- *Biztonsági fenyegetettség*: egy lehetséges formája a biztonsági követelmények megsértésének. Bármilyen körülmény vagy esemény, amely a bizalmasság, sértetlenség és/vagy a rendelkezésre állás megsértését *okozhatja*.
- *Sebezhetőség*: egy rendszer akkor sebezhető, ha ténylegesen lehetőség van arra, hogy a biztonsági követelmények sérüljenek (vagyis kihasználható biztonsági hibát tartalmaz).
- *Támadás*: a rendszerben lévő sebezhetőség *kihasználása* (pl. jogosulatlan hozzáférés szerzése vagy a rendszer működésének meggátolása).
- *Biztonsági hiba*: olyan konkrét hiányosság vagy funkció, amely a rendszert sebezhetővé *teheti*.

Számunkra ebben a dolgozatban e legutóbbi fogalom lesz a legfontosabb. Biztonsági hiba nagyon sok helyen kerülhet a rendszerbe: már rögtön a

követelmények meghatározásánál, vagy a rendszer tervezésénél, az implementálás során, de akár még a használat, ill. működtetés közben is okozhat egy nem megfelelő konfiguráció vagy környezet biztonsági hiányosságokat. Ez a munka a biztonsági szempontból veszélyes programozói hibákra összpontosít, vagyis azokra a hibákra melyeket egy szoftver implementálása során követnek el a programozók. A dolgozatban használt központi fogalom a biztonsági szempontból veszélyes programozói hiba, ezért vizsgáljuk meg először is azt, hogy mit is takar ez a kifejezés. A következő ábra segíti ennek bemutatását.



2. ábra – Programozói hibák csoportosítása

A programozói hibákon belül, azok a hibák, amelyek a biztonsági követelmények megsértését okozhatják, *biztonsági szempontból veszélyesek*. Az ilyen típusú hibákat nem minden esetben lehet kihasználni, hogy azzal valóban meg is sértsük a biztonsági követelményeket, de ha a hiba erre ténylegesen lehetőséget ad, akkor az már a *kihasználható biztonsági hiba kategóriába* tartozik. Ezek a hibák okozzák a valódi sebezhetőségeket. A létező rendszerekben az esetek túlnyomó többségében már valamilyen, a múltból ismert *tipikus hibákat* találnak és használnak ki a támadók. Viszonylag ritkán derül fény olyan biztonsági hibára, amihez hasonló azelőtt nem létezett.

Fontos továbbá megjegyezni, hogy egy biztonsági szempontból veszélyes hiba kihasználhatósága nem mindig dönthető el könnyen és véglegesen. Bizonyos hibák egy adott szoftver verzióban lehet, hogy nem kihasználhatók, azonban a

konfiguráció vagy a környezet megváltozása után kihasználhatókká válhatnak. Ezért célszerű ezeket a hibákat – aktuális kihasználhatóságuktól függetlenül – kijavítani és magát a fenyegetettséget megszüntetni.

1.3 Célok

A dolgozat célja, hogy áttekintse és elemezze a biztonsági réseket okozó hibák megtalálására hivatott módszereket és újabb megoldásokat mutasson be. Ezek a technikák természetesen kétélű kardnak tekinthetők, hiszen rossz kezekben újabb sebezhetőségek felfedezésére és azok kihasználásra, visszaélésre használhatók fel. Azonban a szoftverek csak annak kiadása után kerülnek a támadók kezei közé, éppen ezért fontos, hogy ezeket a technikákat még a fejlesztés alatt alkalmazzák, hogy a talált hibákat még időben ki lehessen javítani.

Természetesen ahhoz, hogy valamit megtaláljunk, tudnunk kell mit keresünk. Ahogy már említettem, a felfedezett biztonsági rések mögött a legtöbb esetben valamilyen tipikus, már ismert programozói hiba áll. Így célunk az lesz, hogy ezeket a tipikus hibákat keressük, amelyek adott estben kihasználható biztonsági lyukakhoz vezethetnek. Az ilyen veszélyesnek tartott hibákat még akkor is érdemes megkeresni és kijavítani, ha a talált hiba nem is kihasználható, hiszen a lehetséges okok eliminálásával a problémákat megnyugtatóan kezelni tudjuk. E munka célja többek között, hogy ezeket a tipikus hibákat összeszedje és elemezze.

1.4 Összefoglalás

A második fejezetben be fogom mutatni azokat a leggyakrabban előforduló tipikus programozói hibákat, amelyek biztonsági szempontból veszélyesek. A harmadik fejezetben a lehetséges biztonsági hibákról egy áttekintő rendszerezést próbálok felállítani, felhasználva és szintetizálva az irodalomban megjelent taxonómiákat. A negyedik fejezetben a bemutatott hibák elleni lehetséges védekezési stratégiákat sorakoztatom fel, a biztonságtechnológiából ismert PreDeCo[1] szemléletmód erre a területre alakított felosztása szerint. Az ötödik fejezetben a védekezési megközelítések közül a hibák megtalálásának a módszereivel foglalkozom. Ezen

[1] Preventív, detektív és korrektív megközelítések.

belül megvizsgálom a biztonsági tesztelés, valamint a hagyományos szoftvertesztelés viszonyát. A hatodik fejezetben a véletlenszerű biztonsági tesztelésnek, az ún. „Fuzzing”-nak a továbbfejlesztéséről lesz szó. A hetedik fejezetben bemutatom az általunk[2] fejlesztett, FLINDER-nek keresztelt biztonsági tesztelő keretrendszert. A nyolcadik fejezetben, a kertrendszer gyakorlatban való kipróbálását médialejátszó alkalmazások tesztelésen keresztül fogom bemutatni. Végül az eredmények értékelése és a konklúziók levonása után a jövőbeli terveimről is szót ejtek.

[2] A FLINDER keretrendszer a SEARCH Laboratórium fejlesztése, melynek tervezésében és implementálásában magam is részt vettem.

2 TIPIKUS IMPLEMENTÁCIÓS HIBÁK

Ebben a fejezetben bemutatásra kerül néhány, a legnagyobb számban előforduló biztonsági szempontból veszélyes programozási hiba. Az egyes alfejezetekben leírásra kerül az adott hibatípus oka, hogy miért okozhat a hiba sebezhetőséget, valamint, hogy hogyan lehet kihasználni az okozott sebezhetőséget a biztonsági követelmények megsértése, vagy a védelmi mechanizmusok megkerülése érdekében. Több különböző technika is ismertetésre kerül, ugyanis egyesek alkalmasak lesznek a később bemutatott védekezési módszerek kijátszására is. A kihasználási technikák bemutatásánál elsősorban az IA32/x86 architektúra a feltételezett, de kisebb eltérésekkel más architektúrákon is kivitelezhetőek.

2.1 PUFFER TÚLCSORDULÁS – AZ ÖREG HIBA

A biztonsági szempontból veszélyes programozási hibák közül a legrégebb óta jelenlévők, legsűrűbben elkövetettek és legveszélyesebbek azok, amelyek puffer túlcsorduláshoz vezethetnek. Mégis, a mai napig nem sikerült tökéletes megoldást kitalálni a problémára. A puffer túlcsordulás akkor történhet meg, amikor a szoftver egy fix hosszúságú tömböt (puffert) lefoglal a memóriában és a tömb írásakor, nem ellenőrzi annak határait. Ilyenkor egy támadónak lehetősége nyílik arra, hogy egy lefoglalt tömböt túlírva (tipikusan valamilyen túlzottan hosszú bement segítségével) felülírjon a program működése szempontjából fontos adatokat a memóriában, és így akár a saját igényei szerint befolyásolja a program futását. Ezek a hibák elsősorban a C/C++ programozási nyelv sajátosságai miatt fordulnak elő. A C nyelvben egy tömb lefoglalása után egy mutatót kapunk vissza, és a későbbiekben is csak ezzel a mutatóval hivatkozhatunk rá. Így a fordítónak is nehéz eldöntenie, hogy egy tömböt túl fogja-e írni valamilyen művelet, valamint futásidőben sincs információ a lefoglalt tömb méretéről. A határérték ellenőrzések megvalósítása ezért teljes egészében a programozóra hárul.

2.1.1 TÚLCSORDULÁS A STACK-EN

A legsűrűbben elkövetett és a legkönnyebben kihasználható eset az, amikor a programozó a szóban forgó tömböt lokális változóként definiálja [3]. Ilyenkor

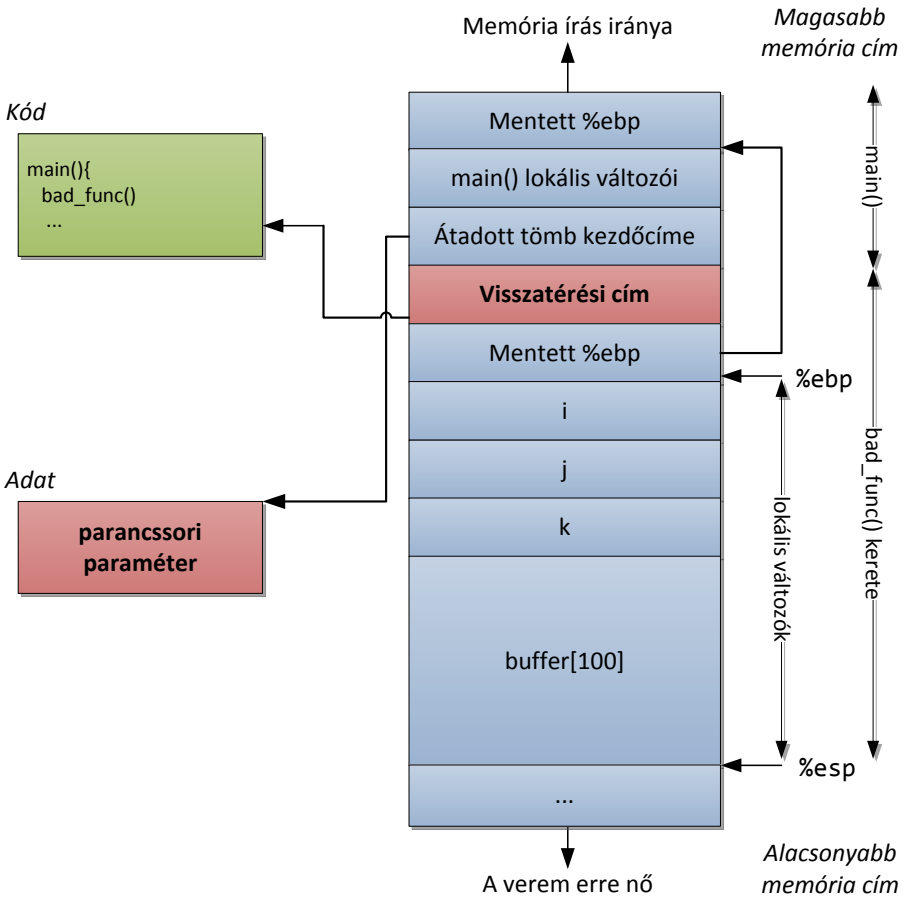
ugyanis a tömb a stack-en (vermen) tárolódik. Vegyük szemügyre a következő ábrán látható hibásan megírt programot:

```
void bad_func(char *userinput)
{
  int i=1;
  int j=2;
  int k=3;
  char buffer[100];

  strcpy(buffer, userinput);
  printf("%s", buffer);
}

int main(int argc, char *argv[])
{
  bad_func(argv[1]);
  return 0;
};
```

3. ábra – Hibás C forráskód

A program az első argumentumaként kapott karaktersorozat kezdőcímét átadja a `bad_func()` nevű függvénynek. Majd a `strcpy()` könyvtári függvény segítségével a megadott címen lévő sztringet a lokálisan deklarált `buffer` nevű tömbbe másolja.

A függvény meghívása után a stack állapota a 4. ábra látható. Az IA32/x86-os architektúrán a stack lefelé növekszik, tehát az újabban bekerült adatok alacsonyabb memóriacímre kerülnek, mint a régebbiek. A verem keretekre van osztva, ahol az egyes keretek az aktuális függvény adatait tárolják. Ilyenek a meghívott függvény argumentumai, a lokális változók és az elmentett keret cím valamint visszatérési érték. Az éppen aktuális keret kezdetét a keretmutató (`%ebp` regiszter), a keret és egyben a stack alját a veremmutató (`%esp` regiszter) tárolja.



4. ábra – A verem állapota

Amikor a példánkban a `main()` meghívja a `bad_func()`-ot, akkor a szokásos függvényhívás mechanizmus játszódik le, amely a következő lépésekből áll:

A hívó függvény:

- A stack-re írja a hívott függvény paramétereit (jelen esetben az átadott tömb kezdőcímét).
- A stack-re írja a visszatérési címet (a `call` utasítást követő első utasítás címét).

A hívott függvény prológja:

- A stack-re menti az aktuális keretmutatót (`%ebp` regisztert).
- Az új keretmutatónak az aktuális veremmutatót állítja be (`%esp` regisztert).
- Csökkenti a veremmutatót, ezzel helyet allokálva a lokális változók számára.

A függvény prológus/epilógus a fordító által automatikusan generált utasítássorozat, amely minden függvény elején/végén megtalálható. A `bad_func()` futása során a `strcpy()` a paraméterben megadott karaktersorozatot a `buffer`-rel jelölt memóriaterületre másolja. Visszatéréskor pedig, a hívott függvény epilógusa a következő további lépéseket hajtja végre:

- A veremmutatót beállítja az aktuális keretmutatóra, ezzel a lefoglalt helyet felszabadítja.
- A keretmutatót visszaállítja a stackről levett, elmentett keretmutatóra.
- A visszatérési címet szintén kiveszi a veremből és hozzárendeli a programszámlálóhoz (`eip`), hogy arról a címről folytatódjon a futás.
- Ezzel a vezérlés visszatér a hívott függvényhez, és a program folytatódik az elvárt működés szerint. *De nem minden esetben!*

2.1.1.1 A hiba kihasználása

A hibát több különböző módon is ki lehet használni, ezek közül ismertetem a legjellemzőbb támadásokat.

A visszatérési cím átírása

Ha a felhasználó a lefoglalt tömb méreténél hosszabb karakterláncot ad meg, akkor (a tényleges hossztól függően) felülírhatja a 4. ábrán látható `i, j, k` változókat, az elmentett keretmutatót, valamint a *visszatérési címet* is. Természetesen, ha nincs semmiféle ellenőrzés, akkor még sokkal tovább is írhat, de általában a legegyszerűbb a visszatérési cím átírása, jelen esetben például a `buffer` kezdőcímére, ahová a támadó tetszőleges programkódot helyezhetett. Így, amikor a függvény visszatér, nem az azt meghívó `call` utasítás után folytatódik a futás, hanem a támadó által magadott kódra ugrik a vezérlés.

A keretmutató átírása

Előfordulhat, hogy a támadó nem tudja felülírni a visszatérési értéket, mert például csak egy-két bájttal, tudja túlírni a lefoglalt tömböt, vagy pedig a visszatérési cím valamilyen technikával védve van. Ez esetben is sikerre viheti a támadását, csak az elmentett keretmutató felülírásával, ami még a visszatérési cím előtt helyezkedik el a stack-en. Tegyük fel, hogy a támadó átírja az elmentett

keretmutatót. Ilyenkor a függvény visszatérésekor a fentebb leírt módon a keretmutató a stack-re mentett és a támadó által átírt értékre áll vissza (epilógus 2. lépés). Majd a következő függvényvisszatérésnél már a veremmutató kapja meg az aktuális keretmutató értékét (epilógus 1. lépés), tehát a verem alja a támadó által bevitt értékekre mutat, ahonnan a visszatérési cím kerül majd kiolvasásra (epilógus 3. lépés). Ha tehát arra a memóriacímre írja át az elmentett keretmutatót a támadó, amely címen az ő kódsorozata kezdőcímét helyezte el, a vezérlést e módon is oda tudja irányítani. Ezzel a technikával a támadónk a visszatérési cím közvetlen átírása helyett, az általa futtatni kívánt kódsorozat címét elhelyezi valahol a memóriában és eléri, hogy a verem alja a megfelelő címre kerüljön ahhoz, hogy a `ret` utasítás *arról* a helyről vegye le a visszatérési címet.

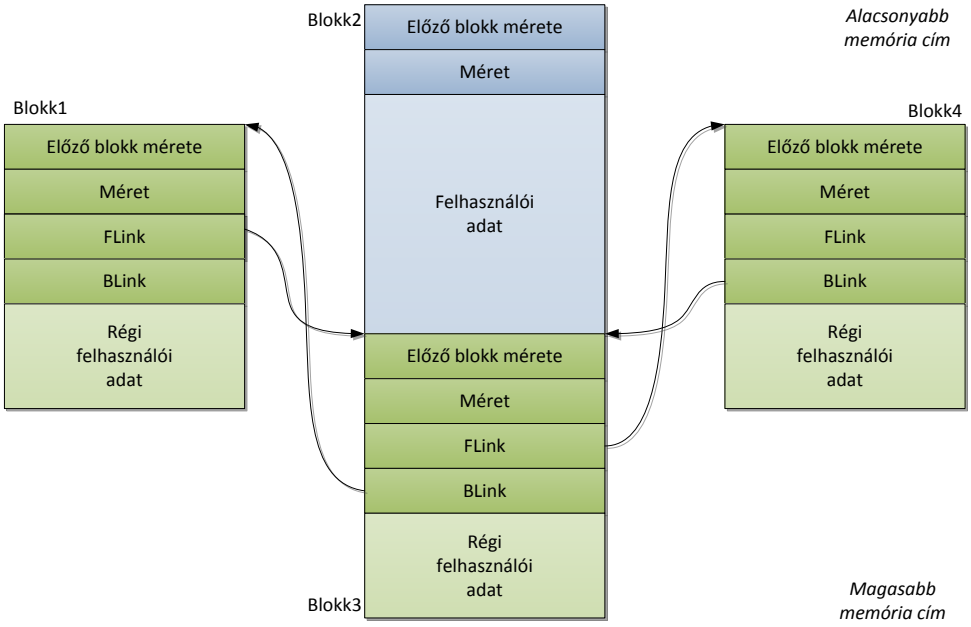
További lehetőségek

Egy stack-en történő puffer túlcsordulás esetén a támadónak szinte végtelen számú lehetősége van, ugyanis a veremben rengeteg lényeges adat tárolódik, amelynek manipulálásával befolyásolni lehet a programfutást illetve annak hatásait vagy eredményeit. Például egy függvénymutató értéke is lecserélhető a támadó kódjára mutató mutatóra, vagy az átadandó függvényparaméterek manipulálásával is elérhető hasonló eredmény. Ha nincs lehetőség elég hosszú kódot íratni a stack-re, vagy az azon való kódfuttatás nem engedélyezett, akkor egy alkalmas, már eleve a memóriában lévő függvény címére is elég átírni a visszatérési címet a megfelelő paraméterek használatával (pl. `system()` standard C függvény). Az ilyen és ehhez hasonló támadásokat „Return to libc” támadásnak nevezzük [4]. A kihasználás módja tehát mondhatni a támadó fantáziájára van bízva.

2.1.2 TÚLCSORDULÁS A HEAP-EN

Ha nem lokális változóként deklarálunk egy puffert, hanem dinamikusan foglaljuk, akkor nem a stack-en, hanem a heap-en allokálódik számára hely. Ez hasonlóképpen túlírható, ahogyan a verem esetében láttuk [5]. A heap a memória egy olyan címtartománya, ahonnan szabadon választott méretű memóriablokkokat lehet dinamikusan lefoglalni. A heap tehát egymást követő szabad illetve foglalt blokkokból (ún. chunk) tevődik össze, amelyek mérete a

memóriafoglalások, és felszabadítások függvényében változik. Két szabad blokk nem lehet közvetlenül egymás mellet, mert ha így lenne, akkor azok azonnal egybeolvadnának egy nagyobb szabad blokkba a töredezettség csökkentése érdekében. A szabad blokkok rendszerint egy duplán láncolt listában vannak tárolva, a méretük szerint rendezve. Minden blokk, a kezeléséhez szükséges összes információt, mint például a méretét vagy a szomszédos blokkok címét „önmagában" tárolja egy fejlécben. A heap egy részletét láthatjuk az 5. sematikus ábrán. A `Blokk2` foglalt, míg `Blokk1`, `Blokk3` és `Blokk4` szabad. A szabad blokkok fejlécében az előző és a saját méret után megtalálható az következő és az előző szabad blokk címe is (`FLink` és `BLink`). `Blokk2`-t közvetlenül követi a memóriában `Blokk3`. `Blokk1` és `Blokk4` elhelyezkedése a mi szempontunkból most irreleváns.



5. ábra – A heap állapota

Ebben az esetben a problémát alapvetően az fogja okozni, hogy a láncolt lista kezeléséhez szükséges adatstruktúrák közvetlenül határolják a felhasználói adatokat, hasonlóképpen, ahogyan a stack-en is a visszatérési érték, vagy a mentett keretmutató a lokális változókkal együtt tárolódnak. Amikor egy bizonyos méretű tömb allokálására kerül sor, akkor a szabad blokkok listájából az első megfelelő méretű blokk törlődik a listából és felhasználhatóvá válik, vagyis egy

lefoglalt blokká válik. Amikor pedig felszabadul egy blokk, akkor az beszúrásra kerül a lista megfelelő helyére.

2.1.2.1 A hiba kihasználása

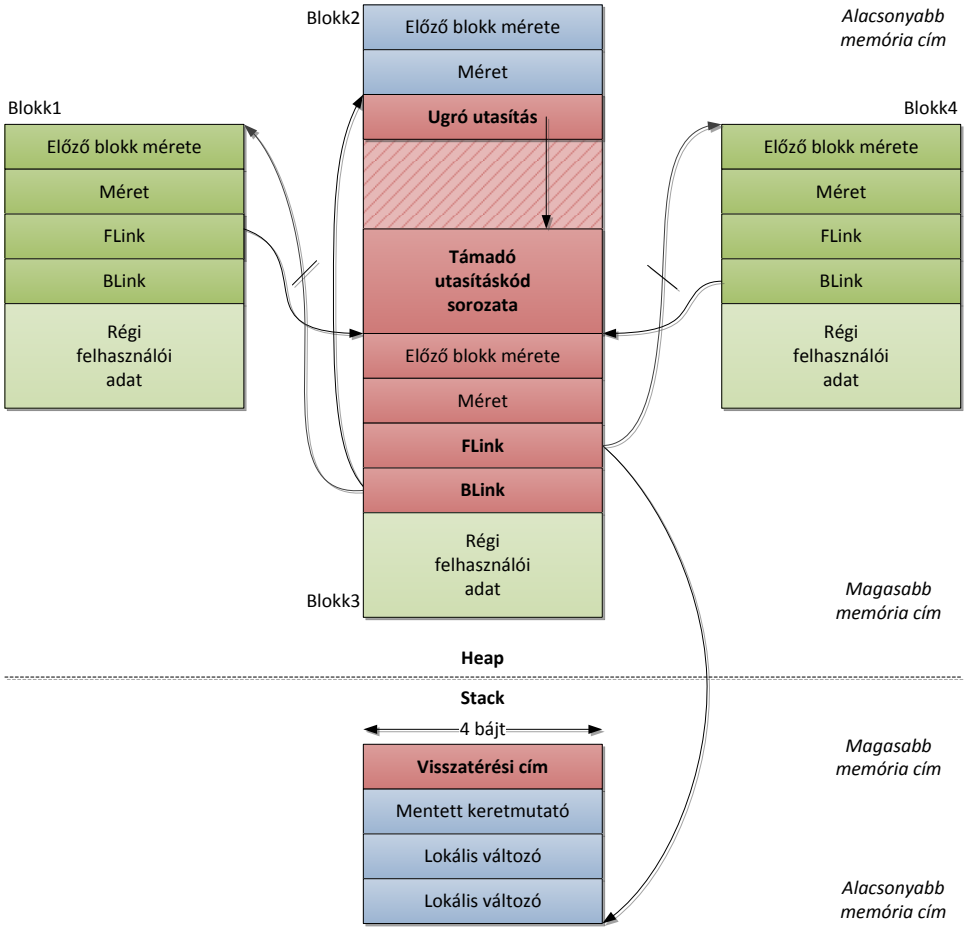
A hiba kihasználására ez esetben is több lehetősége van a támadónak. A technikák a memóriamenedzser implementációjától is függnek, tehát kicsit eltérő technikát kell alkalmazni például Windows, Linux vagy BSD operációs rendszer esetén. A következőkben ismertetett technikák bemutatása során egy általános, leegyszerűsített modellt alkalmaztam, hogy a módszer lényegére adhassak rávilágítást.

Listából való törlés kihasználása

Egy lefoglalt szabad blokk törlése a duplán láncolt listából a jól ismert két lépésben történik (C szintaktikával), ahol `P` a törlendő blokkra mutató mutató:

```
(1)     P->FLink->BLink = P->BLink;
(2)     P->BLink->FLink = P->FLink;
```

Vizsgáljuk meg, hogy mi történik akkor, ha egy támadó túlírja `Blokk2`-t a 6. ábrán látható módon. Támadónk az általa használható területnek rögtön a legelejére egy ugró utasítást helyez el, amely néhány bájttal előre ugrik. Majd az ugrás helyétől kezdve, a rendelkezésre álló területtől függően tetszőleges kódsorozatot helyez el. Az ugróutasítás és a tényleges kód közti kis részre a későbbiekben látni fogjuk, hogy miért volt szükség. Ez a terület kitölthető bármivel.



6. ábra – Túlcsordulás kihasználása a heap-en

Az ábrán látható, hogy a Blokk2 túlírásával a támadó felülírja a `Blokk3` fejlécében az `FLink` és `BLink` mutatókat. Az `FLink`-et úgy, hogy az a stack-en lévő visszatérési cím címe elé mutasson éppen 12 bájttal, `BLink`-et pedig úgy, hogy az a fent említett ugró utasításra mutasson.

Most pedig nézzük meg, hogy a Blokk3 felszabadításakor mi fog történni. A láncolt listából való törlés első lépése másképp írva (a struktúra alapján):

```
(1)      *(Blokk3->FLink+12) = Blokk3->Blink;
```

Vagyis a stack-en lévő visszatérési cím legyen egyenlő az ugró utasításunk címével. Valamint a második lépes, szintén „kifejtve”:

```
(2)      *(Blokk3->BLink+8) = Blokk3->FLink;
```

Vagyis az ugró utasításunk után következő területre írjuk be az `FLink` értékét. Ez nem válik a támadó hasznára, viszont ezért kellett kihagynia az ominózus kis

területet az ugró utasítás és a „hasznos” utasítássorozat között. Így a következő függvényvisszatéréskor, a stack-en történő puffer túlcsordulás kihasználásához hasonló módon, a vezérlés (az ugró utasításon keresztül) a támadó utasításkód sorozatára ugrik.

További lehetőségek

Ha ismét szemügyre vesszük a láncolt listából való törlés első lépését, láthatjuk, hogy az `FLink`-nek értéket adva meghatározhatunk egy tetszőleges címet, a `BLink`-nek értéket adva pedig egy tetszőleges értéket, ami törlés pillanatában az `FLink`-ben megadott címre fog íródni. Ezt „*Write-What-Where*” kondíciónak nevezi a szakirodalom. Egy ilyen helyzetet természetesen nem csak a stack-en lévő visszatérési cím átírására lehet használni. Például ELF binárisok esetén egy osztott programkönyvtár függvényeinek a címe a „Global Offset Table”-ben van tárolva. A fenti támadáshoz hasonlóan egy gyakran használt függvény címét is lecserélheti a támadó az általa a memóriába juttatott támadó kód címére, de akár a heap-en tárolt virtuális függvény mutatókkal is megteheti ezt, hogy eltérítse a programvezérlést.

Listába való beszúrás kihasználása

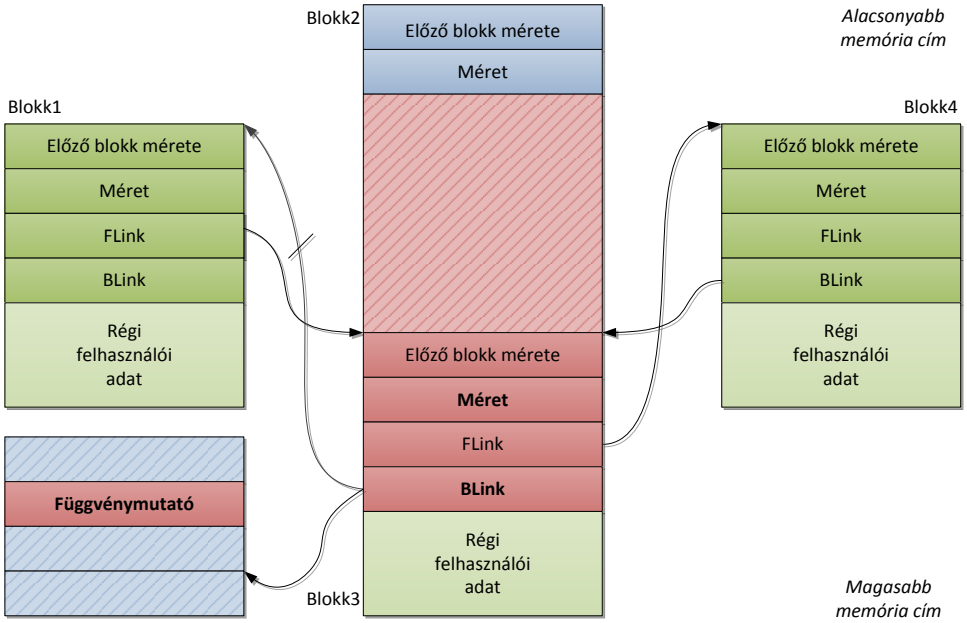
Ha felszabadításra kerül egy használt memória blokk, akkor azt be kell szúrni a szabad blokkokat nyilvántartó láncolt listába. Nézzük meg, hogyan is történik ez. A blokkok a méretük szerint növekvő sorba vannak rendezve a duplán láncolt listában. Az `UJ` mutató tartalmazza a felszabadult blokk címét, a `P` pedig a lista első elemének címét:

```
(3)     UJ = felszabadult_blokk; P = lanc_eleje;
(4)     while(P!=NULL && UJ->meret > P->meret) P = P->FLink;
(5)     UJ->FLink = P; UJ->BLink = P->BLink;
(6)     P->BLink->FLink = UJ;
(7)     P->BLink = UJ;
```

A `while` ciklus megkeresi az első szabad blokkot, amely nem kisebb az új szabad blokkunknál. A 3. sorban az beillesztett blokk mutatói frissítjük, a (4)-ben a

megelőző blokk előre mutató mutatóját, majd (5)-ben az új blokk előtt lévő blokk hátra mutató mutatóját.

Tegyük fel, hogy a támadó futtatható kódot tud elhelyezni egy blokkban, amely valamikor felszabadításra kerül, vagyis a fenti módon bekapcsolódik a láncolt lista megfelelő helyére. Ezenkívül az előzőleg ismertetett támadáshoz hasonlóan egy puffer túlcsordulást kihasználva felülírja egy szabad blokk fejlécét. Ez esetben a méret mezőt átírja egy viszonylag nagy értékre, a `BLink` mezőt pedig például egy függvénymutató címére (pontosabban a függvénymutató címének értéke mínusz 8-ra). Ezt ábrázolja a 7. rajz.



7. ábra – Blokk felszabadításának kihasználása

Amikor a támadó kódját tartalmazó blokk felszabadul, a `while` ciklus nagy valószínűséggel a `Blokk3`-nál áll meg, majd (4) alapján:

```
(6)      *(Blokk3->BLink+8) = UJ;
```

Vagyis a választott függvénymutatónk az újonnan felszabadított blokkra fog mutatni, a támadó kódjával feltöltve. Amikor a függvény meghívásra kerül, az injektált kód lefut.

2.2 A DUPLA FREE() HIBA

Gyakran előforduló probléma C/C++ programokban, hogy a program vezérlési gráfjában olyan út marad, amely során ugyanazt a memóriaterületet kétszer egymás után felszabadítja a programozó [6]. Tekintsük példaképpen a következő ábrán látható kódrészletet.

```
char* ptr = (char*) malloc(SIZE);
...
if (abrt) {
  free(ptr);
}
...
free(ptr);
```

8. ábra – Memória felszabadítás kétszer egymás után

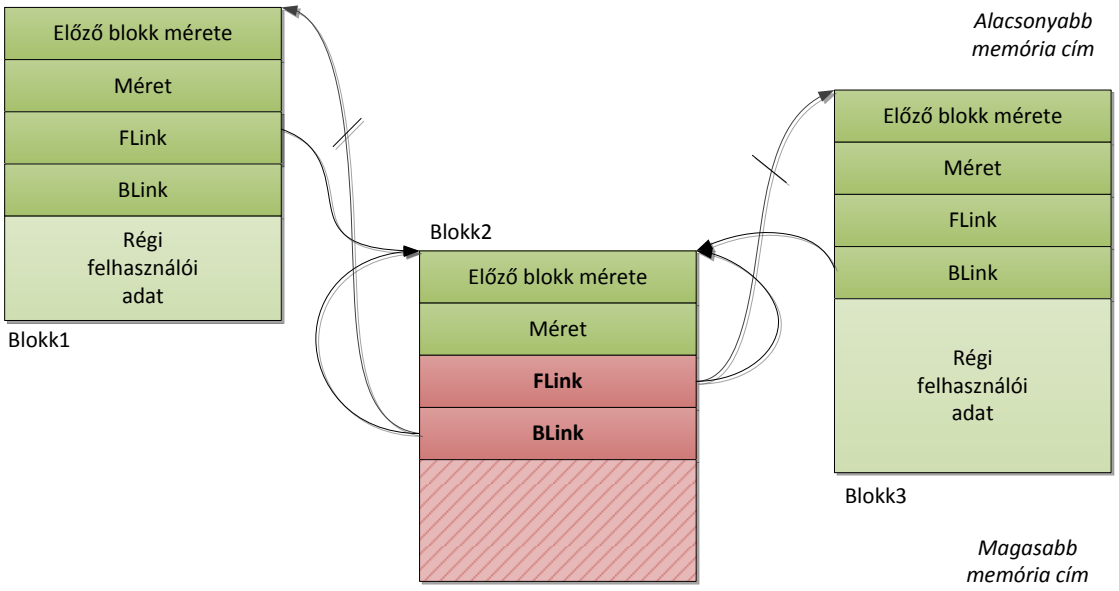
Sajnos a fordítók általában nem ellenőrzik, hogy egy mutató értéke érvényes-e még a használat pillanatában, ezért ezek a hibák gyakran rejtve maradnak, hiszen csak néhány esetben jönnek elő. Azonban egy szemfüles támadó ezt a teljesen ártatlannak tűnő hibát is fel tudja használni, hogy akár átvegye segítségével a rendszer feletti irányítást.

2.2.1 A HIBA KIHASZNÁLÁSA

Kövessük végig, hogy mi történik pontosan, ha pl. az 9. ábra látható `Blokk2`-n kétszer egymás után hajtódik végre a `free()` hívás. Az első felszabadítás alkalmával a `Blokk2` bekerül a láncolt lista megfelelő helyére, a mi esetünkben éppen a `Blokk1` és `Blokk3` közé. A második felszabadítási kísérlet alkalmával a (4)-ben lévő `while` ciklus `P`-t éppen a `Blokk2`-n fogja megállítani, hiszen az lesz az első blokk, amelyik nem kisebb saját magánál. Mivel most mind a `P`, mind az `UJ` mutató értéke `Blokk2`-re mutat, nézzük meg hogyan alakul tovább a „beillesztés”:

```
(5)     Blokk2->FLink = Blokk2; Blokk2->BLink = Blokk2->BLink;
(6)     Blokk2->BLink->FLink = Blokk2;
(7)     Blokk2->BLink = Blokk2;
```

Azt kapjuk, hogy `Blokk2`-nek az `FLink` és a `BLink` mutatója is, az 9. ábra látható módon, saját magára fog mutatni.



9. ábra – Heap állapota kétszer egymás utáni felszabadítás után

Ez után pedig azt vizsgáljuk meg, hogy a blokk legközelebbi használatbavételekor, vagyis a listából való törléskor mi történik. A szokásos két lépés, ismét átírt nevekkel:

```
(1)     Blokk3->FLink->BLink = Blokk3->BLink;
(2)     Blokk3->BLink->FLink = Blokk3->FLink;
```

Ha alaposan szemügyre vesszük a `Blokk3->FLink`, valamint a `Blokk3->BLink` átírható `Blokk3`-ra, vagyis sem az `FLink`, sem a `BLink` nem változik. Akkor mire volt jó ez az egész, kérdezhetnénk. Arra, hogy a memória menedzser a blokkot ettől függetlenül lefoglaltnak tekinti, ami azt jelenti, hogy az eddig az `FLink`-et illetve `BLink`-et tároló terület írhatóvá válik. Támadónk ismét használhatja az legelőször ismertetet heap-en történő támadást, vagyis az `FLink` helyén megadja a választ a „hova”, a `BLink` helyén pedig a „mit” kérdésre. Amikor a program újra egy Blokk2 méretű területet szeretne lefoglalni, a listából való törlés már elintézi a többit.

2.3 VERSENYHELYZETEK

Konkurens vagy megszakítható programkódokban előforduló versenyhelyzetek szintén sebezhetőséghez vezethetnek. Az egyik „legveszélyesebb” hibaforrás a szignálok használata.

Versenyhelyzet a szignál kezelésnél

Ha egy szignálkezelő függvény csak egyszer meghívható függvényeket hív meg, vagy állapotfüggő műveleteket hajt végre, akkor a memória nem várt módon módosulhat, és a program kihasználhatóvá válik. Mivel egy szignál bármikor megszakíthatja a programfutást, ezért veszélyes a szignálhoz rendelt függvénynek olyan adatokon vagy struktúrákon műveleteket végezni, amelyek esetleg inkonzisztens állapotban lehetnek a megszakítás pillanatában. A szignálkezelő rutinok egymást is megszakíthatják.

```
void *global1, *global2;
char *what;

void sh(int dummy) {
  syslog(LOG_NOTICE,"%s\n",what);
  free(global2);
  free(global1);
  sleep(10);
  exit(0);
}

int main(int argc,char* argv[]) {
  what=argv[1];
  global1=strdup(argv[2]);
  global2=malloc(340);
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
```

10. ábra – Hibás szignálkezelés

Például az fenti ábrán látható példaprogramban ugyanazt a függvényt rendelték hozzá két különböző szignálhoz, amelyben a `free()` meghívásra kerül. Így egy támadó elérheti, hogy memória felszabadítás kétszer egymás után történjen, amely a fentebb már bemutatott dupla `free()` sebezhetőséghez vezethet. Még abban az esetben is, ha a `free()` után a programozó elővigyázatosságból a felszabadított mutatóhoz `NULL`-t rendel, ha a megszakítás közvetlenül a `free()` után és a `NULL` hozzárendelése előtt történik, a sebezhetőség továbbra is fennáll.

2.4 EGÉSZ SZÁMOKKAL KAPCSOLATOS HIBÁK

Az egész számokkal kapcsolatos hibák alapvetően a számítógépek számábrázolásának korlátai miatt jelentkeznek, illetve a nem megfelelő hiba- vagy kivételkezelés miatt. Ezek a hibák általában nem okoznak önmagukban sebezhetőséget, de nagyon gyakran sebezhetővé teszik a programot a fentebb bemutatott hibák kihasználására irányuló támadásoknak [7].

Aritmetikai túlcsordulás

Az aritmetika túlcsordulás (integer overflow) akkor következik be, amikor egy egész számot nagyobbra növelünk (pl. egy összeadás vagy szorzás művelettel), mint amekkora maximális értéket tárolni tud a számábrázolás. Ha például felhasználjuk ezt a számot egy memóriafoglalásnál, elképzelhető, hogy a szám túlcsordulása miatt túl kevés memóriát foglalunk, és ezzel egy puffer túlcsordulásos sebezhetőséget hozunk létre a heap-en.

Előjelezési hiba

A legtöbb programozási nyelvben, ha a programozó definiál egy egész számot, akkor, ha csak explicite nem definiálja előjel nélkülinek az egy előjeles szám lesz. Később, ha ezt az értéket átadja egy függvénynek, amely egy előjel nélküli számot vár paraméteréül, akkor a számot implicite előjel nélkülivé konvertálja a fordító (casting), és a továbbiakban úgy is értelmezi. Ez azért jelenthet problémát, mert egy negatív szám, még előjelesként értelmezve átmegy egy puffer túlcsordulás kivédésére beszúrt maximális hosszt vizsgáló feltételen, majd azt ezt követő másolást végrehajtó függvény paramétereként már előjel nélküliként, egy nagy számmá válik, és így ismét egy puffer túlcsordulásos sebezhetőséget idéz elő.

Eltérő bitszélesség

Előfordulhat, hogy egy nagyobb méretű változót (pl. egy 32 bites integert) szeretnénk kisebb területen eltárolni (pl. egy 16 bites short változó helyén), amely nem képes azt befogadni, ezért az érték csonkolódik.

Kihasználási lehetőségek

Az ilyen típusú hibák a legtöbb esetben nem használhatók ki, hiszen a memória közvetlenül nem módosul, de ha az adott túlcsordult változót például egy tömb lefoglalásakor felhasználjuk, akkor puffer túlcsordulást idézhetünk segítségükkel elő, vagy pedig az éppen az ilyen típusú hibák elkerülésre írt hibakezelő kódrészlet kerülhető meg, például a fentebb említett csonkolási hibát kihasználva. Gyakran pedig, a megfelelő hibakezelés hiányában összeomlik a rendszer, és így egy szolgáltatás megtagadásos (Denial of Service) támadásra ad lehetőséget.

2.5 HIBA A TÖMBINDEXELÉSBEN

A tömbindexelési hiba akkor jön elő, amikor egy tömbnek a felhasználó által megadott eleméhez biztosít hozzáférést a program. Ha a program nem véd a tömbből való kicímzés ellen (akár egy integer overflow sebezhetőséget kihasználva), a fentebb bemutatott hibákhoz hasonlóan tetszőleges memóriaterületet képes megcímezni a támadó és bizonyos esetekben írhatja is azokat. Ez a típusú hiba sokkal ritkábban használható ki, mint a stack overflow, de gyakorlatilag ugyanaz a probléma jelenik meg, csak más köntösben.

2.6 A PRINTF() FORMÁTUMLEÍRÁS HELYTELEN HASZNÁLATA

A standard C library robosztus, könnyen kezelhető kiíró függvénye a `printf()`, illetve annak változatai (`nprintf()`, `sprintf()`, `snprintf()`, `fprintf()`, `vprintf()` stb.). Ezek előnyös, jól használható szolgáltatása, hogy egy formátum sztring megadásával egyszerűen leírható, hogy a megadott, különböző típusú paraméterek, a megjelenített szövegben hol és milyen alakban jelenjenek meg.

Abban az esetben azonban, ha a formátum sztring és az utána átadott paraméterek nem felelnek meg egymásnak, akkor hibás lesz a működés, amely – mint látni fogjuk – támadásra ad lehetőséget [8]. Ezen a téren a leggyakoribb hiba, hogy ha valamilyen külső bemenetből származó szöveget formátum sztringként kezelünk, ugyanis ekkor nem kontrollálható, hogy bekerülnek-e vezérlő karakterek az adott szövegbe.

Egy ilyen tipikus hiba látható az alábbi ábrán:

```
int main(int argc, char *argv[])
{
  printf(argv[1]);
  return 0;
};
```

11. ábra – Printf() hibás használata

A parancssori paraméterben vezérlő karaktereket elhelyezve, a támadó olyan „hibás működést" képes előidézni, amely révén *információkat tud kiolvasni* a program memóriájából, manipulálni képes memóriacímek tartalmát és akár át is tudja venni a vezérlést a támadott gép felett. Például a fenti programot a következő paraméterekkel meghívva:

```
%X %X %X %X %X %X
```

kiírja hexadecimális számrendszerben a stack-en tárolt értékeket (amelyek között titkosnak minősülő adatok is lehetnek), illetve a:

```
%s %s %s %s %s %s
```

mutatóként értelmezi a stack-en található értékeket, így nem csak a veremből, hanem e pointerek által mutatott memóriatartományból is egyszerűen kiirattathatunk információkat, melyek szintén lehetnek bizalmas jellegűek (pl. egy rejtjelkulcs vagy jelszó).

A vezérlő karakterek között azonban a `%n` a legérdekesebb, mert ez nem csupán a megjelenést befolyásolja, hanem memóriapozíciók felülírására is képes. Ennek a vezérlő karakternek a funkciója, hogy a paraméterként megadott pointer által mutatott memóriapozícióra kiírja, hogy az adott printf végrehajtása során eddig hány karaktert jelent meg a képernyőn. Tekintve, hogy megfelelő sztring megválasztásával a képernyőn megjelentetett karakterek számát könnyen befolyásolni lehet, gyakorlatilag megoldható, hogy a `%n` tetszőleges értéket írjon be a megcímzett memória rekeszbe. Tehát e hiba kihasználásával szintén, ahogyan azt már stack-en ill. heap-en történő túlcsordulásoknál is láttuk, a támadónak akár tetszőleges kód futtatására van lehetősége.

2.7 EGYÉB BEMENET-ELLENŐRZÉSBŐL SZÁRMAZÓ HIBÁK

A fentiekben bemutatott programozói hibák a legtöbb esetben azért vezetnek sebezhetőséghez, illetve válnak kihasználhatóvá, mert a külső felhasználótól származó adatok a memóriában keverednek a rendszer működése vagy vezérlése szempontjából fontos adatokkal, melyekről a programozó *feltételezi*, hogy változtathatatlanok. Ez többek között a Neumann-architektúra sajátossága, miszerint a programkód és az adat ugyanabban a memóriában van. Az ilyen kevert felhasználású, azaz az adat és vezérlőinformáció továbbítására egyaránt használt információs csatornák nem csak operációs rendszer szinten, hanem magasabb, alkalmazás szinten is megjelenhetnek. Az ilyen alkalmazások legjellegzetesebb kihasználási módja az ún. „befecskendezés" (injection) típusú támadás. A hiba rendkívül elterjedt és széles körben, különböző környezetekben és módokon használható ki. A probléma alapja mindenhol ugyanaz: a programozók gyakran a felhasználói bemenetek és belső utasítások egyszerű összefűzésével állítják elő a belső rendszer számára kiadott parancsokat, az adatnak szánt bemenet ellenőrzése, filterezése nélkül.

Az ilyen típusú támadások egyik legismertebb fajtája, főleg webes alkalmazásoknál, a parancs befecskendezés [9], az SQL befecskendezés [10] valamint a „Cross site scripting" [11].

2.7.1 PARANCS BEFECSKENDEZÉS

Tegyük fel, hogy egy CGI alkalmazás egy űrlapban megadható e-mail címet felhasználva, a következő utasítást hajtja végre: „`cat somefile | mail emailaddress`", ahol az *`emailaddress`* a felhasználható által megadott paraméter. A támadó ilyenkor, ha nincs megfelelő paraméter ellenőrzés, az e-mail címként a „`eve@attacker.com | rm –fr /`" karakterláncot megadva, például képes lehet letörölni a webszerver minden adatát.

2.7.2 SQL BEFECSKENDEZÉS

Olyan rendszereknél, ahol a háttérben egy SQL adatbázis működik, a felhasználó által megadott adatok általában egy SQL parancsba vagy lekérdezésbe ágyazódnak

bele. Ilyenkor, ha a támadó SQL parancs elemeket illeszt az általa megadott adatokba, az eredeti parancs értelmét meg tudja változtatni.

2.7.3 CROSS SITE SCRIPTING (XSS)

A Cross Site Scripting sebezhetőség akkor jöhet elő, ha például egy webes alkalmazás fejlesztője egy űrlapba írható adatokat nem ellenőrzi, majd később a bevitt adatokat megjeleníti. Ilyenkor a támadó általában egy JavaScript kódot helyezve az űrlapba, majd az eredményt megjelenítő URL-t elküldve áldozatának, lefuttathatja saját kódját, amivel személyes információkat szerezhet az áldozat gépéről.

2.8 ÖSSZEFOGLALÁS

A fenti sebezhetőségeket kihasználó támadásoknak különös veszélye az, hogy nem szükséges semmilyen jogosultság a támadó részéről az adott gépen, illetve az, hogy bármilyen alkalmazáson keresztül, amelyben pl. túlcsordulásos hiba van, és amelyik a külvilágból kaphat bemenetet, lehetőség nyílik betörésre[3]. Így nem elegendő pusztán az operációs rendszer, illetve a biztonsági modulok hibamentes implementációja annak érdekében, hogy e támadásokat kiküszöböljük, hanem az *összes telepített programnak hibamentesnek kellene lennie*. Ez a követelmény pedig a mai rendszereknél nem teljesül, és mint ahogy arról a bevezetőben már szó volt, ez gyakorlatilag ma nem is teljesíthető.

[3] Nem csupán az Internetről közvetlenül elérhető szerveralkalmazások tartoznak ebbe a kategóriába, hanem pl. egy egyszerű böngésző program, amely esetleg a támadó által felállított weblapon elhelyezett képet jelenít meg. Ha a képmegjelenítő modulban hiba van, a támadó sikerrel tud járni.

3 BIZTONSÁGI HIBÁK OSZTÁLYOZÁSA

A fentiekben bemutatásra került példa értékű tipikus biztonsági hibákat a leggyakrabban előforduló hibák közül választottam. A bevezetőben szó volt arról, hogy ha védekezni szeretnénk a biztonsági rések ellen, akkor pontosan tisztában kell lennünk azzal, hogy mire kell odafigyelnünk, mit kell, hogy keresünk. Más szóval, ismernünk kell az összes olyan tipikus hibát, amelyek biztonsági szempontból veszélyesek lehetnek. Ezért célszerű lenne ezeket összegyűjteni és rendszerbe foglalni, hogy később szisztematikus módszerekkel tudjunk velük foglalkozni. Erre a célra szolgál a taxonómia készítés, ami tudományos vezérelvek szerinti besorolást, rendszerezést jelent.

A biztonsági hibák többféle szempontból osztályozhatók: például a hibák kihasználási technikái szerint, a hiba oka vagy eredete szerint, a hibát tartalmazó komponensek szerint, vagy a hiba természete szerint stb. Az irodalom sok ilyen taxonómiával rendelkezik már. Ezek közül mutatok be néhányat, csak a lényegre koncentrálva. Megjegyzendő, hogy a következő osztályozások nem csak azokat a szoftverimplementációs hibákat tartalmazzák, amivel a dolgozat elsősorban foglalkozik, hanem egyes esetekben tervezési, konfigurációs és egyéb biztonsági hiányosságokat is.

3.1 ELMÉLETI OSZTÁLYOZÁSOK

Az ebben a részben bemutatásra kerülő taxonómiákat azért nevezem elméleti osztályozásoknak, mert ezekben a gyakorlatban előforduló tipikus hibákról konkrétan nincs szó, sokkal inkább a legfőbb kategóriák beazonosítása a céljuk. A nézőpontok különbözőek lehetnek, de mindegyik arra törekszik, hogy egy adott szempontból teljes legyen a kategorizálás.

3.1.1 A RISOS TANULMÁNY

A „Research Into Secure Operating Systems” [12] című tanulmány az operációs rendszereknél felmerülő biztonsági hiányosságokat sorakoztatja fel. Hét általános kategóriát állít fel, melyek a következők:

1. **Hiányos paraméter ellenőrzés**
   (pl. puffer túlcsordulás, egész túlcsordulás)
2. **Inkonzisztens paraméter ellenőrzés**
   (pl. különböző függvények különböző formátumra számítanak)
3. **Titkos adatok implicit megosztása**
   (pl. rejtett csatornák, „side channel” támadások)
4. **Aszinkron ellenőrzés / sorrendi hiba**
   (pl. versenyhelyzetek, TOCTTOU[4])
5. **Nem megfelelő azonosítás / hitelesítés / engedélyezés**
   (pl. trójai program)
6. **Megszeghető tilalom / korlát**
7. **Kihasználható logikai hiba**

Meglátásom szerint ez az osztályozás túl magas absztrakciós szinten valósult meg, így a túl tág kategóriák számomra nem voltak jól használhatók.

3.1.2 A PA MODELL

Valamivel mélyebben boncolja az operációs rendszerek biztonsági hiányosságait a Protecion Analysis modell [13]. Ennek a tanulmánynak a biztonsági hiányosságok felfedezésére általánosságban használható stratégia kidolgozása volt a célja. Ez egy minta alapú kiértékelő stratégia, amit „Pattern-directed protection evaluation”-nek is neveztek. Ennek segítségével, mindaddig ismeretlen sebezhetőségekre bukkantak az akkori operációs rendszerekben. Peter G. Neumann ‘95-ben tisztázta és leegyszerűsítette az osztályokat. Ennek a verziónak a kategóriái a következők voltak:

[4] Time Of Checking To Time Of Usage

1. **Improper protection domain initialization and enforcement**
    a. Improper choice of initial protection domain (domain)
    b. Improper isolation of implementation detail (exposed representations)
    c. Improper change (consistency of data over time)
    d. Improper naming (naming)
    e. Improper deallocation or deletion (residuals)
2. **Improper validation (validation of operands, queue management dependencies)**
3. **Improper synchronization**
    a. Improper indivisibility (interrupted atomic operations)
    b. Improper sequencing (serialization)
4. **Improper choice of operand or operation (critical operator selection errors)**

3.1.3 LANDWEHR TAXONÓMIA

Landwehr taxonómiája [14] a biztonsági hibák mai napig leggyakrabban hivatkozott rendszerezése. A taxonómia három különböző szempont szerint állítja fel a kategóriáit:

- *Hiba eredete*
- *Keletkezés ideje*
- *Hiba helye*

Ezek közül számunkra legérdekesebb a hiba eredete szerinti csoportosítás. Ebben két fő csoport van: a szándékosan megvalósított funkcióból eredő (intentional) és véletlenszerűen, programozási és/vagy tervezési hibából eredő (inadvertent) biztonsági lyukak. A nem szándékos csoporton belüli osztályok meghatározása a RISOS tanulmány alapján történtek:

<table>
<tr><td rowspan="6">Intentional</td><td rowspan="3">Malicious</td><td rowspan="2">Trojan Horse</td><td>Non-Replicating</td></tr>
<tr><td>Replicating (virus)</td></tr>
<tr><td colspan="2">Trapdoor</td></tr>
<tr><td colspan="2">Logic/Time Bomb</td></tr>
<tr><td rowspan="2">Nonmalicious</td><td rowspan="2">Covert Channel</td><td>Storage</td></tr>
<tr><td>Timing</td></tr>
<tr><td colspan="2">Other</td></tr>
<tr><td rowspan="6">Inadvertent</td><td colspan="3">Validation Error (Incomplete/Inconsistent)</td></tr>
<tr><td colspan="3">Domain Error (Including Object Re-use)</td></tr>
<tr><td colspan="3">Serialization/aliasing (Including TOCTTOU Errors)</td></tr>
<tr><td colspan="3">Identification/Authentication Inadequate</td></tr>
<tr><td colspan="3">Boundary Condition Violation
(Including Resource Exhaustion
and Violable Constraint Errors)</td></tr>
<tr><td colspan="3">Other Exploitable Logic Error</td></tr>
</table>

12. ábra – Landwehr taxonómia

3.1.4 ASLAM TAXONÓMIÁJA

Aslam a UNIX operációs rendszerre specifikus biztonsági hibákat szedte kategóriákba [15]. Ő a programozási hibákon kívül figyelembe vette programozótól független, környezetből vagy konfigurációból adódó hiányosságokat is.

<table>
<tr><td rowspan="13">Security fault</td><td rowspan="3">Operational fault</td><td rowspan="3">Configuration error</td><td colspan="2">Object installed with incorrect permissions</td></tr>
<tr><td colspan="2">Utility installed in the wrong place</td></tr>
<tr><td colspan="2">Utility installed with incorrect setup parameters</td></tr>
<tr><td colspan="4">Environment fault</td></tr>
<tr><td rowspan="11">Coding fault</td><td rowspan="9">Condition validation error</td><td rowspan="5">Input validation error</td><td>Field value correlation</td></tr>
<tr><td>Syntax error</td></tr>
<tr><td>Type & number of input fields</td></tr>
<tr><td>Missing input</td></tr>
<tr><td>Extraneous input</td></tr>
<tr><td colspan="2">Failureto handle exceptions</td></tr>
<tr><td colspan="2">Origin validation</td></tr>
<tr><td colspan="2">Access rights validation error</td></tr>
<tr><td colspan="2">Boundary condition error</td></tr>
<tr><td rowspan="2">Synchronization error</td><td colspan="2">Improper or inadequate serialization error</td></tr>
<tr><td colspan="2">Race condition error</td></tr>
</table>

13. ábra – Aslam taxonómia

1.1.1 *PIESSENS MODELLJE*

Az eddig bemutatott taxonómiák keverik a különböző absztrakciós szinteket, és átfedéseket is tartalmaznak. Piessens egy másik szempontból próbálta összeszedni a hibákat [16], mintegy „checklist”-ként szolgálva a fejlesztés különböző fázisaira.

ANALYSIS PHASE	No Risk Analysis / No Security Policy
	Biased Risk Analysis
	Unanticipated Risks
DESIGN PHASE	Crypto protocol design errors
	Relying on non-secure abstractions
	Security / Convenience tradeoff
	No logging
	Design does not capture all risks
IMPLEMENTATION PHASE	Insufficiently defensive input checking
	Non-atomic check and use
	Access validation errors
	Incorrect crypto primitive implementation
	Insecure handling of exceptional conditions
	Bugs in security logic
DEPLOYMENT PHASE	Reuse in more hostile environments
	Complex or unnecessary configuration
	Insecure defaults
MAINTENANCE PHASE	Feature interaction
	Insecure fallback

14. ábra –Piessens taxonóima

1.1.2 *WEBER TAXONÓMIÁJA*

A szinte már „de facto”-ként kezelt Landwehr taxonómia több különböző kritikát kapott az évek során. Egyik hibája, hogy a partícionálás mértéke nem megfelelően finom. Egy másik vélemény vele kapcsolatban, hogy a trójai falovak és a vírusok bevétele a rendszerbe nem helytálló, hiszen ha a fejlesztő írta bele szándékosan a programjába, akkor az nem hibának minősül, ha pedig egy támadó, akkor azt csak úgy lehetett képes véghezvinni, hogy egy valamilyen másik hibát kihasználva juttatta a rosszindulatú kódot a szoftverbe. Ezen kívül mivel Landwehr 1994-ben készítette ezt a munkáját, a rendszer már idejétmúlt: ma már jól elkülöníthető

gyakori tipikus hibák közös csoportokba vannak szedbe. Weber ezeken a hiányosságokon próbált segíteni. Látható, hogy a felbontás kicsit finomodott [17]:

<table>
<tr><td rowspan="5">Intentional</td><td rowspan="2">Malicious</td><td colspan="2">Trapdoor</td></tr>
<tr><td colspan="2">Logic/Time Bomb</td></tr>
<tr><td rowspan="3">Non-malicious</td><td rowspan="2">Covert Channel</td><td>Storage</td></tr>
<tr><td>Timing</td></tr>
<tr><td colspan="2">Inconsistent access paths</td></tr>
<tr><td rowspan="10">Inadvertent</td><td rowspan="4">Validation Error</td><td colspan="2">Addressing error</td></tr>
<tr><td colspan="2">Poor parameter value check</td></tr>
<tr><td colspan="2">Incorrect check positioning</td></tr>
<tr><td colspan="2">Identification/Authentication Inadequate</td></tr>
<tr><td rowspan="2">Abstraction Error</td><td colspan="2">Object Reuse</td></tr>
<tr><td colspan="2">Exposed Internal Representation</td></tr>
<tr><td rowspan="2">Asynchronous flaws</td><td colspan="2">Concurrency (including TOCTTOU)</td></tr>
<tr><td colspan="2">Aliasing</td></tr>
<tr><td rowspan="2">Subcomponent misuse/failure</td><td colspan="2">Resource Leak</td></tr>
<tr><td colspan="2">Responsibility Misunderstanding</td></tr>
<tr><td rowspan="2">Functionality Error</td><td colspan="2">Error handling failure</td></tr>
<tr><td colspan="2">Other security flaw</td></tr>
</table>

15. ábra – Weber taxonómia

3.2 GYAKORLATI OSZTÁLYOZÁSOK

Az eddig megismert taxonómiák elméletinek nevezhetőek olyan szempontból, hogy a kategóriák túlságosan tágak. A gyakorlatban arra jók, hogy egy konkrét hibát besoroljunk valahova, de a „kézzelfogható” típushibák felbontását egyik sem éri el. Ezzel a „top-down” szemléletű osztályozással ellentétben, gyakorlati szempontból érdemesebb lenne inkább a konkrét hibák szintjéről elindulni egy „bottom-up” szemléletmódot követve úgy, hogy felsoroljuk az egyes típushibákat, majd csoportokba foglaljuk őket. A következő hibarendszerezések már sokkal inkább erre az alsó szintű megközelítésre, vagyis a konkrét hibatípusok szintjére építkeznek.

1.1.3 SEVEN PERNICIOUS KINGDOMS

A „A hét veszedelmes birodalom” címet viselő tanulmány [18], hét fő kategóriát azonosít, és ezeken beül már jól meghatározott hibatípusokat sorakoztat fel. Tehát ez egy egyszerű, viszont annál hasznosabb kétszintű hibabesorolás. A hét fő osztály, egy-két alá tartozó hibával:

1. **Input Validation and Representation**
   (Buffer Overflow, Cross-Site Scripting, Format String, Integer Overflow, Path Manipulation, SQL Injection, String Termination Error, XML Validation, Command Injection, Resource Injection, ...)
2. **API Abuse**
   (Dangerous Function, Directory Restriction, Heap Inspection, …)
3. **Security Features**
   (Insecure Randomness, Least Privilege Violation, ...)
4. **Time and State**
   (Deadlock, File Access Race Condition: TOCTTOU, …)
5. **Errors**
   (Catch NullPointerException, Empty Catch Block, …)
6. **Code Quality**
   (Double Free, Memory Leak, Null Dereference, Obsolete, …)
7. **Encapsulation**
   (Data Leaking Between Users, Leftover Debug Code, …)

3.2.1 PLOVER

Egy másik, gyakorlati szempontból igen jól használható hibagyűjtemény a „Preliminary List of Vulnerability Examples for Researchers" [19]. Ez egy olyan többéves munka eredménye, melynek a célja az volt, hogy közismert sebezhetőség típusokhoz rendeljenek CVE-neveket. A CVE (Common Vulnerabilities and Exposures) adja az ismert sebezhetőségeknek a legelfogadottabb „szabványos" nevét. A PLOVER a legnagyobb, legrészletesebb és talán a legteljesebb mai hibatípus gyűjtemény.

3.2.2 CLASP

A CLASP (Comprehensive, Lightweight Application Security Process) gyűjteménye azoknak a fázisoknak, melyek a szoftverfejlesztés folyamatába integrálhatók, annak érdekében, hogy már a fejlesztés kezdeti szakaszában is a biztonsági követelmények figyelembe vételével dolgozzunk [20].

A tanulmány része egy „sebezhetőségek fő okai” (*Vulnerability Root-Causes*) című fejezet, amely szintén konkrét hibatípusok összegyűjtését kísérli meg a teljesség igényével, öt fő kategória alá sorolva azokat. Az öt kategória a következő:

1. Értéktartomány és típus-összeférhetetlenség
2. Környezeti hiba
3. Szinkronizációból vagy időzítésből eredő hiba
4. Protokoll hiba
5. Általános logikai hibák

3.2.3 OASIS WAS VULNERABILITY TYPES

Az OASIS (Organization for the Advancement of Structured Information Standards) egy non-profit nemzetközi konzorcium, ami az e-kereskedelemben előforduló szabványokkal foglalkozik [21]. A WAS (Web Application Security) Vulnerability Types, a webes alkalmazásokban előforduló biztonsági hibák gyűjteménye, de gyakorlatilag bármilyen más alkalmazásban is felhasználható az általános volta miatt.

3.3 TOPLISTÁK

Két hasznos listát említenék még meg. Az egyik az **OWASP (Open Web Application Security Project) Top Ten Most Critical Web Application Security Vulnerabilities,** amely a tíz leggyakoribb webes alkalmazásokban előforduló hibát sorolja fel [22]. A másik a **The 19 Deadly Sins of Software Security,** ami egy könyv címe [23], de alapjában véve egy lista a szofterfejlesztés során leginkább elkövetett 19 biztonsági hibáról.

3.3.1 OWASP TOP TEN

1. Nem ellenőrzött bemenet
2. Hibás hozzáférés védelem
3. Hibás hitelesítés és session-kezelés
4. Cross Site Scripting
5. Puffer túlcsordulás (buffer overflow)
6. Beszúrásos (injection) hibák
7. Nem megfelelő hibakezelés
8. Nem megfelelő tárolás
9. Szolgáltatás megtagadás (Denial of Service)
10. Nem biztonságos konfigurációkezelés

3.3.2 THE 19 DEADLY SINS OF SOFTWARE SECURITY

1. Puffer túlcsordulás (buffer overflow)
2. Printf (format string) hiba
3. SQL beszúrás (injection)
4. Parancs beszúrás (injection)
5. Nem megfelelő hibakezelés
6. Cross Site scripting
7. Nem védett hálózati forgalom
8. „magic" URL-ek és elrejtett adatlapok használata webes oldalaknál
9. Az SSL nem megfelelő használata
10. Gyenge jelszó alapú rendszerek használata
11. Adatok védelemének hiánya
12. Információszivárgás
13. Nem megfelelő fájl hozzáférés
14. Egész túlcsordulás (integer overflow)
15. Megadott hálózati címben való megbízás
16. Versenyhelyzet szignálkezelésben
17. Nem hitelesített kulcscsere
18. Kriptográfiai szempontból nem megfelelő véletlenszám generátor
19. Rossz használhatóság

3.4 Összefoglalás

A fentiekből jól látható, hogy a hibatípusok összeszedése és kategorizálása egyáltalán nem egyértelmű és különösen nehéz feladatnak bizonyul. Nem alakult még ki egy mindenki által elfogadott taxonómia. A fentiekben bemutatottak is más-más nézőpontokat használnak, és különböző vagy sokszor vegyes, összemosódott absztrakciós szinteken dolgoznak. A legtöbb rendszerezés még csak nem is elégíti ki a matematikai értelemben vett taxonómia fogalmát, miszerint annak minden részletre kiterjedőnek, teljesnek és kategóriáinak kölcsönösen kizárónak kell lennie. Gyakran a hibaosztályok összemosódnak. A szoftverbiztonság területén a gyakorlatban a tervezőknek, fejlesztőknek és a tesztelőknek az adott rendszerre vonatkozó fenyegetésekkel kell tisztában lenniük. Mivel egy univerzális, minden rendszerre érvényes modellt alkotni rendkívül nehéz, ezért mindig az adott rendszerre vonatkozó hibalehetőségeket kell összeszednünk. Éppen ehhez nyújtanak segítséget a fent bemutatottak úgy, hogy „checklist”-ként használhatók fel.

4 VÉDEKEZÉSI MÓDSZEREK

Ha már tudjuk, hogy mik okozhatják a problémákat, nézzük meg mit lehet ellenük tenni. A következő részben megvizsgálom, hogy mikor, hol és hogyan lehet védekezni a biztonsági hibák ellen. A „mikor" arra vonatkozik, hogy a rendszer életciklusának mely fázisában történik a védelem (pl. specifikáció, tervezés, implementálás vagy tesztelés). A „hol" arra, hogy a fejlesztett rendszer mely pontján védekezünk, ami persze nagyon sok minden lehet (pl. hardver, operációs rendszer, fordítóprogram, forráskód, hálózat pereme stb.). Végül a „hogyan" az adott módszert írja le.

Három fő kategóriát állítottam fel a PreDeCo szemléletmód szerint. A **preventív módszereknek** a hiba elkerülése a céljuk, vagyis hogy egyáltalán létre se jöhessen hiba sem a rendszerünk specifikálása, sem a tervezése és még csak az a implementálása során sem. A **detektív módszerek** célja, a már a rendszer valamely fázisában bekerült hibák minél pontosabb és kimerítőbb felderítése, lehetőleg minél hamarabb. Ezek a módszereket a legjobb esetben már az életciklus legelejétől kezdve alkalmazzák a fejlesztők (belső tesztelés, review), a legrosszabban pedig csak a megfelelő tesztelés nélkül elkészült termék kiadása után fogják rosszindulatú támadók felhasználni, hogy kihasználható biztonsági lyukakat találjanak benne. Az utolsó kategóriába tartozó módszereket inkább **enyhítő módszereknek** neveztem el a korrektív módszerek helyett, ugyanis azoknak nem lesz céljuk a rendszerben maradt hibák kijavítása, csupán azoknak valamilyen módú elfedése, vagy kihasználásuknak, vagyis a támadásoknak a megakadályozása illetve megnehezítése.

4.1 PREVENTÍV MÓDSZEREK – A HIBA MEGELŐZÉSE

A szoftverfejlesztők érdekében az áll, hogy termékük megbízható és biztonságos legyen, ezért a fejlesztés folyamán arra törekednek, hogy kiküszöböljék a programhibákat. Az alábbiakban a legelterjedtebb módszereket foglalom össze.

4.1.1 *FORMÁLIS MÓDSZEREK*

Formális módszerek alkalmazásával bizonyíthatjuk, hogy rendszerünk megfelelően, vagy akár biztonságosan működik. Ehhez szigorú specifikációra és verifikálásra van szükség, azaz matematikai (formális) nyelvek és technikák alkalmazására. A mai szoftverrendszerek mérete és funkcionalitása olyan mértékű komplexitást eredményez, hogy e módszerek használata a gyakorlatban kivitelezhetetlenné válik.

Ebbe a kategóriába sorolható például a modell alapú szoftverfejlesztés, amely arra az elméletre alapszik, hogy a szoftverfejlesztést egy formális modell megalkotásával kell kezdeni. Ez a metodológia szigorú és szisztematikus fejlesztést von maga után, amely megoldást nyújthat a legtöbb hiba elkerülésére. Azonban az ilyen típusú fejlesztések költsége túl nagy ahhoz, hogy a mindennapjainkban használt szoftverek fejlesztéséhez ezt az utat válasszák a fejlesztők.

Éppen ezért az ilyen módszereket, mint a modellellenőrzés vagy az automatikus tételbizonyítás, csak a magasabb szintű tervezés során alkalmazzák, mint például hitelesítési protokollok kidolgozására pl. a BAN [24] logikát vagy az FDR/Caper [25] rendszert.

4.1.2 *BIZTONSÁGOSABB PROGRAMOZÁSI NYELVEK HASZNÁLATA*

A mai operációs rendszerek, eszközmeghajtók és rengeteg felhasználói szoftver is C illetve C++ nyelven íródik. Ezek a nyelvek túl sok szabadságot adnak a programozónak, hogy elég biztonságosak lehessenek. Bizonyos más programozási nyelvek (pl. Ada, Java, C#, Python) szigorú típusossággal, mutatók kiküszöbölésével és egyéb biztonsági megfontolásból bevezetett szabályokkal és megkötésekkel eleve kiküszöbölnek olyan tipikus hibákat, mint pl. a C programokban sokszor előforduló puffer túlcsordulást.

Természetesen vannak olyan területek, ahol a C/C++ használata elkerülhetetlen, például teljesítménykritikus szoftvereknél. Megjegyzem, hogy léteznek, még ha csak kutatási céllal készült, olyan C nyelvre alapuló módosított nyelvek is,

amelyeket úgy terveztek, hogy ne lehessen elkövetni használatuk során a legtipikusabb hibákat. Ilyen például a Cyclone [26] vagy a Ccured [27].

4.1.3 Biztonságos programkönyvtárak

Ha mindenképpen C/C++ nyelven kell fejlesztenünk, akkor érdemes olyan programkönyvtárakat használnunk, amelyek helyettesítik a standard C könyvtár azon függvényeit, amelyek kockázatot jelentenek. Ez elsősorban a sztringkezelő függvényeket jelenti (pl. `strcpy()`). Ezekre nyújthat megoldást például a The Better String Library [28].

4.1.4 Programozási technikák

Bármilyen nyelv használata során lehet biztonsági szempontból veszélyes programozói hibát elkövetni. A szoftverfejlesztő cégek, illetve csapatok gyakran rendelkeznek olyan belső szabályzattal, amelyben megkövetelnek a fejlesztőtől bizonyos módszereket, szabályokat a programkódra vonatkozóan. Ezek általában elsősorban azt a célt szolgálják, hogy megkönnyítsék az együttműködést a projektben dolgozók között (elnevezési szokások, kódminták stb.), de definiálhatunk ilyen szabályokat biztonsági megfontolásból is. Ezek tipikusan veszélyes függvények használatának a tiltása, vagy csak megkötések tétele a használatukra, vagy bizonyos ellenőrzéseknek a megtételének kötelezővé tétele.

4.1.5 Megfelelő bemenet ellenőrzés

A legtöbb hiba abból származik, hogy a programozó nem számít arra, hogy a szoftver nem specifikáció szerinti bemenet kap, hanem feltételezi, hogy a feldolgozandó adat mindig korrekt, helyes és értelmes. Ezért talán a legfontosabb, amire egy programozónak ügyelnie kell, hogy a bemeneteket mindig megfelelően ellenőrizze a feldolgozás előtt. Ehhez pontosan specifikálni kell a bementi nyelvet, a szintaktikát, valamint az értékhatárokat és tartományokat, és az ellenőrzést ennek megfelelően kell implementálni.

4.2 Detektív módszerek – a hiba felismerése

A detektív módszerek alkalmasak arra, hogy a már bekerült hibákat megtalálják. Ezek más szóval a tesztelési módszerek. A szoftvertesztelésről illetve a biztonsági

tesztelésről a következő fejezet szól részletesen. Ki fogok térni a statikus, valamint a dinamikus vizsgálatot végrehajtó módszerekre is, azonban fontos megemlíteni, hogy a tesztelést már a szoftverfejlesztési életciklus legelején kell elkezdeni. A szoftver biztonságát alapos tesztelésnek kell kitenni a termék megjelenése előtt, mert azután még alaposabbnak lesz kitéve esetleg rosszindulatú crackerek által.

4.3 ENYHÍTŐ MÓDSZEREK – A KIHASZNÁLÁS MEGNEHEZÍTÉSE

Ha a hiba már bekerült a rendszerbe még mindig csökkenthetjük a kockázatot olyan védekezési módszerekkel, amelyek elfedik, elrejtik a hibát, vagy csak a kihasználás lehetőségét csökkentik. Ilyen védelmi megoldások megvalósíthatók hardver szinten, operációs rendszer szinten, vagy hálózati szinten is.

4.3.1 HARDVERES VÉDELEM

Hardver szinten megvalósíthatók olyan alapvető védelmi mechanizmusok, amelyek korlátozzák a hozzáférést egyes memóriaterületekhez. A puffer túlcsordulás szerű hibák kihasználása úgy történik, hogy a támadó a saját kódját, mint adat, helyezi el a stack-en vagy a heap-en, majd a vezérlést arra irányítja. Ezeknek a memóriaterületek általában nem kell futtatható kódot tartalmazniuk, csak adatokat, ezért ezeken a szegmenseken tiltani lehet a kódfuttatást. A mai processzorokban már általában van e fajta védelem (például az AMD NX bit, Intel XD bit, ARM XN bit [29]). A támadó a saját kódjára való átirányítást a stack-en általában úgy teszi meg, hogy valamilyen lokális változó túlírásával felülír egy visszatérési címet, amit szintén a stack-en tárol a számítógép. Ezt pedig egy olyan architektúrával lehetne kivédeni, amelyben a hívó és a paraméter stack két elkülönített memóriaterületen foglal helyet.

4.3.2 OPERÁCIÓS RENDSZER SZINTŰ VÉDELEM

Ha a hardver támogat valamilyen védelmi megoldást például futtatás védelmet a memóriában, akkor azt nyilván az operációs rendszernek is fel kell használnia, de léteznek olyan mechanizmusok is, amelyek nem követelnek meg hardveres védelmet (pl. PaX [30], Exec Shield [31] és W^X [32]). Ezek a megoldások természetesen nem védenek minden támadás ellen, például a „Return-to-libc” [4] támadással megkerülhetőek. Nagy segítség a támadó számára a kihasználáskor,

hogy a virtuális memóriában, azonos architektúrán a memória címek állandóak. Így tudhatja, hogy mit mire kell átírni, és hogy bizonyos „hasznos” függvények hol találhatók. Egy másik lehetőség, hogy megnehezítsük ilyenkor a támadó dolgát, ha ezt a memória elrendezést véletlenszerűvé tesszük úgy, hogy mindig változtatunk a kódszegmens, a programkönyvtárak, a stack és a heap báziscímén. Ezt a technikát ASLR-nek (Address Space Layout Randomization) hívjuk. Az operációs rendszer feladata a számítógépes biztonság egyik alapelvének betartatása is, miszerint minden modul (felhasználó, processz) csak azokhoz az erőforrásokhoz férjen hozzá, amire feltétlenül szüksége van (least privilege principle). Ennek biztosítására szolgálnak olyan hozzáférés védelmi megoldások, mint például a SELinux [33]. Ha alkalmazzuk a legkevesebb privilégium elvet, minimalizáljuk a hiba kihasználásával elérhető károkat, és így a kockázatot.

4.3.3 VÉDELEM A FORDÍTÓBAN

A fordító is alkalmazhat olyan enyhítő védekezési módszereket, amellyel a hibák kihasználását lehet megnehezíteni. Erre a legtipikusabb megoldás, hogy a fordított programot kiegészíti úgy, hogy minden újabb stack keretben ún. canary bájtokat (varázsszámokat) helyezzen el, és visszatéréskor, mielőtt törölnénk a keretet, ellenőrizzük, hogy módosultak-e ezek az ellenőrző bájtok. Ha igen akkor puffer túlcsordulás történt. Ezt implementálja például a gcc-hez írt ProPolice [34] és StackGuard [35] folt is.

4.3.4 HÁLÓZATI VÉDELEM

A hálózat szintjén is védekezhetünk a támadások ellen. Egyrészt tűzfalakkal kikényszeríthetjük a hálózat hozzáférési politikánkat. A jobb tűzfalrendszerek mély protokollelemzést végeznek, vagyis egészen az alkalmazás rétegig ellenőrzik az adatforgalmat, így eleve kiszűrhetőek olyan támadások, amelyek valamilyen ismert alkalmazás rétegbeli protokollt használó szerver vagy kliens szoftver ellen irányulnak. Például a CODE RED [36] elnevezésű féreg, amely rengeteg ISS szervert fertőzött meg, egy olyan rosszindulatúan összeállított URL kéréssel volt képes egy puffer túlcsordulásos hibát kihasználni, amely nem felelt meg az RFC szabványnak. A legtöbb tűzfal ezt átengedte a hálózaton, de például a mély protokollelemzést végző Zorp [37] kiszűrte a nem szabványos HTTP kérést.

Egy másik lehetőség az behatolás detektáló rendszer (IDS, Intrusion Detection System) alkalmazása, amely általában ismert támadási mintákat keres a hálózati forgalomban és riaszt, ha találat van [38]. Ez hasonló a víruskereső szoftverek működéséhez. Léteznek anomália-detektáláson alapuló IDS-ek is, de ezek nem bizonyultak eddig hatékonynak az ilyen típusú támadások kivédésére. Az IPS (Intrusion Prevention Sytem) pedig a tűzfal és az IDS funkciók valamilyen kombinálásával születik, ahol az IDS már nem csak riasztásra képes, hanem aktívan be is avatkozik a támadások, illetve gyanús forgalom kivédése érdekében.

4.3.5 FOLTOZÁS

Mivel nem hibamentesen kerülnek ki a szoftverek a fejlesztőktől, egy hiba felbukkanása után azt utólag kell kijavítani. Nagyon fontos, hogy ezek a hibajavítások minél gyorsabban jussanak el a felhasználókhoz, és fel is települjenek. Ezt segítik a szoftverekbe beépített automatikus patch-elő funkciók. Így a minél hamarabbi foltozással tudjuk a kihasználás kockázatát csökkenteni.

5 HIBADETEKTÁLÁS MÓDSZEREI

Az előző részben bemutatott három fő védekezési módszer közül nyilvánvalóan a preventív módszerek lennének a legjobbak, amelyek eleve meggátolják a hibák létrejöttét. Sajnos, mint ahogy arról már volt szó, a mai szoftverfejlesztési módszerek mellett nem biztosítható, hogy minden hibát el lehessen kerülni, főleg a ma még mindig népszerű C/C++ környezetben. Az enyhítő (korrektív) védekezési módszerekre inkább csak, mint a hibák veszélye által generált kockázatot csökkentő módszerekre tekinthetünk. Így mivel tudjuk, hogy „programozói hibák voltak, vannak és lesznek is", a szoftvertesztelésnek óriási fontosságú szerepe van a szoftverfejlesztésben. Először tekintsük át, hogy milyen fő típusai vannak a tesztelési módszereknek általánosságban.

5.1 ÁLTALÁNOS SZOFTVERTESZTELÉS

A szoftvertesztelés az a folyamat, amely segít felmérni az adott szoftver helyességét, teljességét és minőségét. Ezt úgy tesszük meg, hogy feltárjuk hiányosságokat, illetve a hibákat. A teszt típusokat is sok különböző szempontból lehet osztályozni. Lehet például csoportosítani a tesztelt elem szerint (pl. komponens-, integrációs- vagy rendszerteszt), az életciklusban elfoglalt hely szerint (pre-alfa, alfa, béta…), fehér doboz vagy fekete doboz szemléletmód szerint, a tesztelő személye szerint és még sok egyéb más szerint. Számunkra a legfontosabb – tekintve, hogy módszerekről van szó – hogy milyen különböző technikái vannak a szoftvertesztelésnek.

5.2 STATIKUS TESZTELÉS

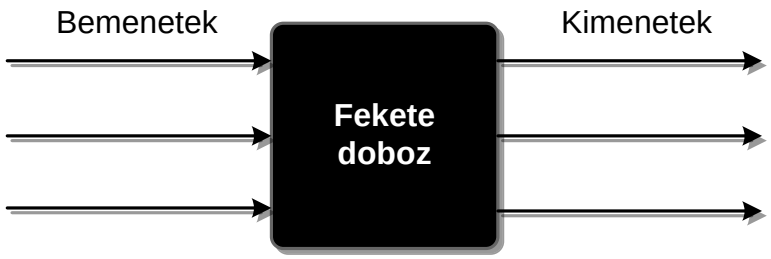
A tesztelés azon technikáját, amikor magát a szoftvert nem futtatjuk, statikus tesztelésnek vagy elemzésnek nevezzük. Ilyenkor általában csak a forráskódot ellenőrizzük, de persze lehet analizálni a lefordított binárist vagy bájtkódot is futtatás nélkül. A tesztelés történhet manuálisan is, amikor is emberi intelligenciával vizsgáljuk a kódot (code-inspection, -review), vagy pedig gépi intelligenciával, automatizáltan. Ez történhet szintaktikai elemzéssel, vagy szimbolikus végrehajtással is.

5.3 DINAMIKUS TESZTELÉS

A statikussal ellentétben, dinamikus tesztelés során a programot futtatjuk. A futás során különböző teszteseteket próbálunk ki. Arra, hogy mi alapján hozzuk létre ezeket, alapvetően kétféle megközelítés létezik. Az egyik a fekete doboz alapú, funkcionális tesztelés, a másik a fehér doboz alapú, strukturális tesztelés.

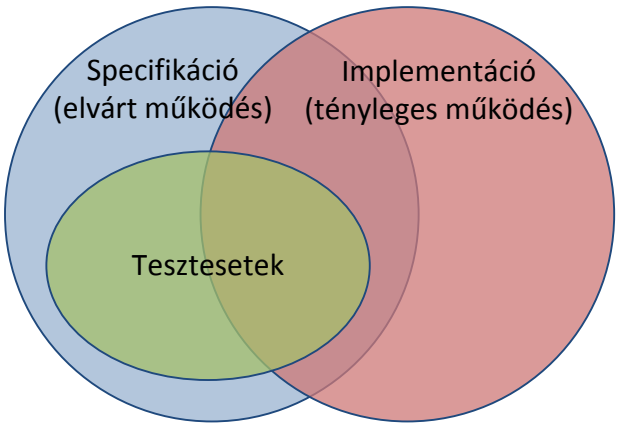
5.3.1 FUNKCIONÁLIS TESZTELÉS

A funkcionális tesztelés arra a szemléletre alapul, hogy bármely rendszer, illetve számítógépes program tekinthető egy függvénynek, amely a bemenetekhez kimeneteket rendel. A fekete doboz alapú megközelítés azt takarja, hogy nem foglalkozunk azzal, hogy ez a függvény hogyan működik, hogyan van megvalósítva, kizárólag azt vizsgáljuk, hogy az egyes bemenetekre hogyan reagál.



16. ábra – Fekete doboz modell

A tesztesetek a program specifikációja alapján kerülnek kidolgozásra. Más alapján nem is kerülhetnének, hiszen ez az egyetlen rendelkezésre álló információnk a programról, mivel a fekete dobozba nem látunk bele. Fontos látnunk, hogy ilyenkor a szoftvernek csak azt a részét tudjuk ellenőrizni, amely a specifikáció által „lefedett”. Ennek könnyebb megértése érdekében tekintsük a következő Venn-diagramot.

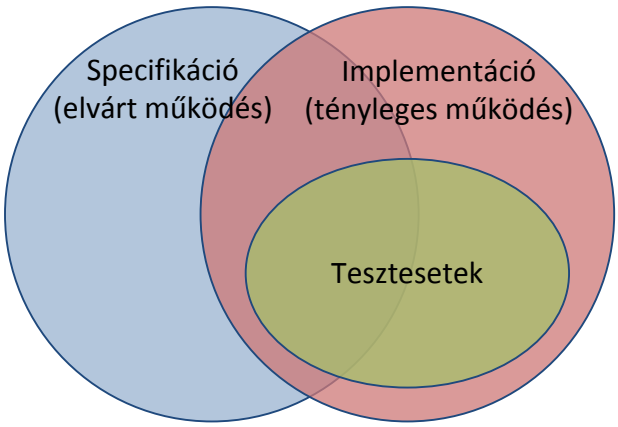


17. ábra –Tesztesetek funkcionális megközelítéssel

A bal oldali halmaz reprezentálja a program specifikációját, vagyis azt, hogy mit szeretnénk, hogy tegyen a program, a jobb oldali pedig az implementált szoftvert, vagyis azt, hogy valójában mit tesz. Tökéletes program esetén ez a két halmaz azonos lenne, de a gyakorlatban ez sajnos nem teljesül. Látható, hogy a specifikáció alapján létrehozott tesztesetek legfeljebb csak a két halmaz metszetét tudják vizsgálni.

5.3.2 STRUKTURÁLIS TESZTELÉS

A strukturális tesztelés a másik megközelítése a tesztesetek megállapításának. Ez az előzővel ellentétesen „belenéz” a dobozba, ezáltal megadja a lehetőséget a tesztelőnek arra, hogy az alapján állítsa össze tesztvektorait, hogy pontosan hogyan került implementálásra az adott szoftver. A strukturális tesztelés előnye, hogy itt konkrétan meg tudja határozni a tesztelő, hogy mit tesztelt és mérni tudja, hogy az implementációnak mely részei kerültek fedésre (code coverage) a tesztelés során. Itt már csak az implementációval foglalkozunk, tehát a tesztesetek most a jobb oldali halmazba kerülnek át.



18. ábra –Tesztesetek strukturális megközelítéssel

Látható, hogy mindkét fajta megközelítés szükséges, hiszen csak strukturális teszteléssel nem fogjuk tudni megállapítani, ha egy specifikált funkció hiányzik a rendszerből, fordítva pedig, ha egy program olyan funkciókat is tartalmaz, amely a specifikációban nem szerepel (például egy szándékos hátsó kapu), akkor azt pedig csak funkcionális teszteléssel megint csak nem fogjuk megtalálni.

5.4 SZOFTVERBIZTONSÁG TESZTELÉSE

Most hogy megvizsgáltuk milyen alapvető technikái vannak a szoftvertesztelésnek, nézzük meg, hogy hogyan viszonyulnak ezekhez a szoftverbiztonságot tesztelő módszerek.

Legelőször fontos, hogy különbséget tegyünk biztonsági szoftver és a szoftverbiztonság között. A biztonsági szoftverek valamilyen biztonsági politikát kikényszerítő funkciót látnak el (pl. hitelesítés, hozzáférés védelem, titkosítás, stb.). Ilyen funkciók persze akármilyen szoftverben lehetnek integrálva, ahol erre szükség van, nem csak olyanban, ami csak és kizárólag ezt teszi. Habár biztonsági tesztelés alá ezeknek a funkciók helyes működésének ellenőrzését is érteni szokták, ezeket alapvetően az általános szoftvertesztelési módszerekkel tehetjük meg. A szoftverbiztonság tesztelésének a célja, hogy azokat a programozói hibákat derítse fel, amelyek biztonsági szempontból veszélyesek lehetnek, amilyenek a dolgozat elején is bemutatásra kerültek. Természetesen ezek a hibák ugyanúgy megtalálhatók a biztonsági funkciók implementációjában is, de a két fogalmat el kell különítenünk egymástól.

A biztonsági teszteléshez kockázat alapú megközelítésre van szükség. Fel kell tárni, hogy az adott szoftvernek mely részei biztonság-kritikusak, és azokban milyen veszélyek növelik a kockázatot, majd ezekre koncentrálva kell a teszteket elvégezni. Biztonsági hibákat lehet statikus és dinamikus módszerekkel is felfedezni.

5.4.1 STATIKUS MÓDSZEREK

Ha manuális kódszemléről van szó, akkor a (biztonsági) szemlézőnek elsősorban azokat a részeket kell átvizsgálnia a forráskódban, amelyekhez egy esetleges rosszindulatú támadó hozzáférhet, vagyis a bementettel, illetve annak feldolgozásával összefügg. A nem megbízható csatornáról származó bementek útját végig kell követnie, és különös figyelmet kell fordítania a veszélyes függvények alkalmazására. Természetesen ezt egy visszafejtett binárison is meg lehet tenni (reverse engeniering), még ha ez egy nehezebb és fárasztóbb feladat is, de ha egy támadó számára a forráskód már nem áll rendelkezésre, a siker érdekében meg fogja tenni ezt is.

Több automatikus statikus analízist végrehajtó szoftver is létezik, amely képes kimutatni bizonyos tipikus biztonsági szempontból veszélyes programozói hibákat, a forrást elemezve. Ilyenek például az ITS4 [39] vagy az újabb Coverity [40] és még sok másik. E szoftvereknek az előnye az lenne, hogy elvben teljes kód lefedettséget tudnak biztosítani. Azonban ez nagyobb szoftvereknél a gyakorlatban nem kivitelezhető, mert túl sokáig tartana az analízis. A mai statikus ellenőrzők hátránya még, hogy túl nagy a hibás riasztások (false positive) aránya. Ez nagyban meg tudja nehezíteni használójuk munkáját.

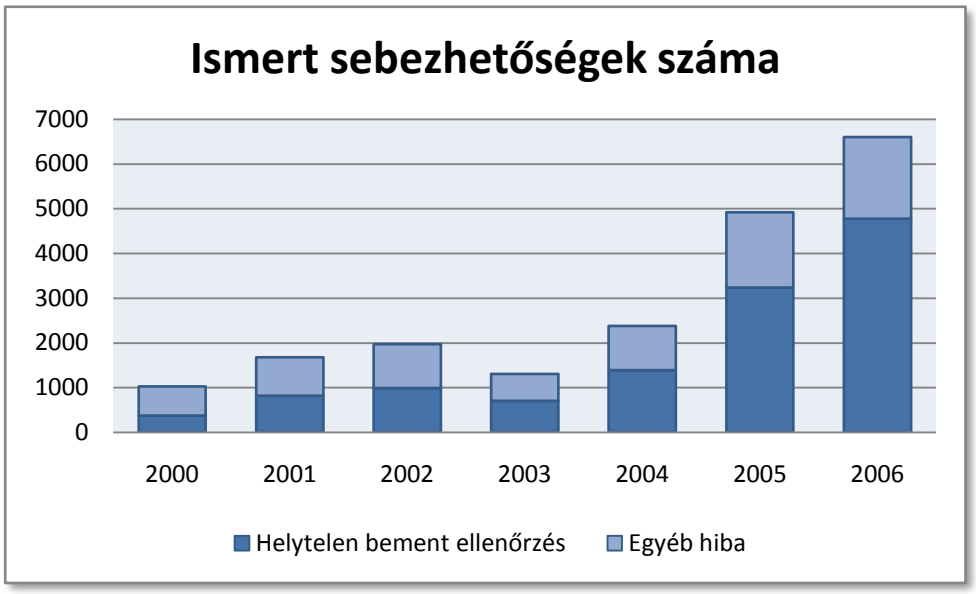
5.4.2 DINAMIKUS MÓDSZEREK

Biztonsági tesztelésről lévén szó, teszteseteinket nem elsősorban a specifikáció és nem is a kód alapján kell meghatároznunk, hanem a hiba hipotéziseink alapján. Például azt feltételezzük, hogy ha „%s”-eket szúrunk be a bemenetbe, akkor az valahol a programban egy `printf()` hibát fog okozni. Tehát fekete doboz módszert alkalmazunk, de nem a specifikációra hagyatkozva állítjuk elő a teszt vektorokat, hanem a veszélyekre és feltételezett már jól ismert, tipikus hibákra alapozva. Ennek a megközelítésnek a legegyszerűbb formája, ha feltételezve a nem

megfelelő bemenet ellenőrzést, teljesen véletlenszerű zagyvaságot adunk a program bemenetére. 1990-ben egy kutatócsoport a Washingtoni Egyetemen így tesztelt szabványos UNIX alkalmazásokat, hogy hosszú, random streameket küldtek a programok bemenetére [41]. A tesztelt programoknak kb. 30%-a elszállt, vagy kiakadt. Ezt a módszert „fuzzing”-nak nevezték el. ’95-ben megismételték újra UNIX-on [42], majd 2000-ben Windows NT-n [43] és tavaly Mac OS X-en is [44]. Az eredmények nem változtak sokat. Míg például egy szerveralkalmazásnál ezzel a módszerrel rendkívül könnyen szolgáltatás megtagadásos sebezhetőségek fedezhetők fel, azonban egy puffer túlcsordulásos hiba előbukkanására kicsi az esély, hiszen a legtöbb program valamilyen alpvető bemeneti formátumot elvár és ellenőriz, amely elvárások teljesülését nem pratktikus a véletlenre bízni. Érezhető tehát, hogy ennek a módszernek az alapján sokkal szofisztikáltabb technikák is kidolgozhatók. Ezzel a területtel a következő fejezet foglalkozik.

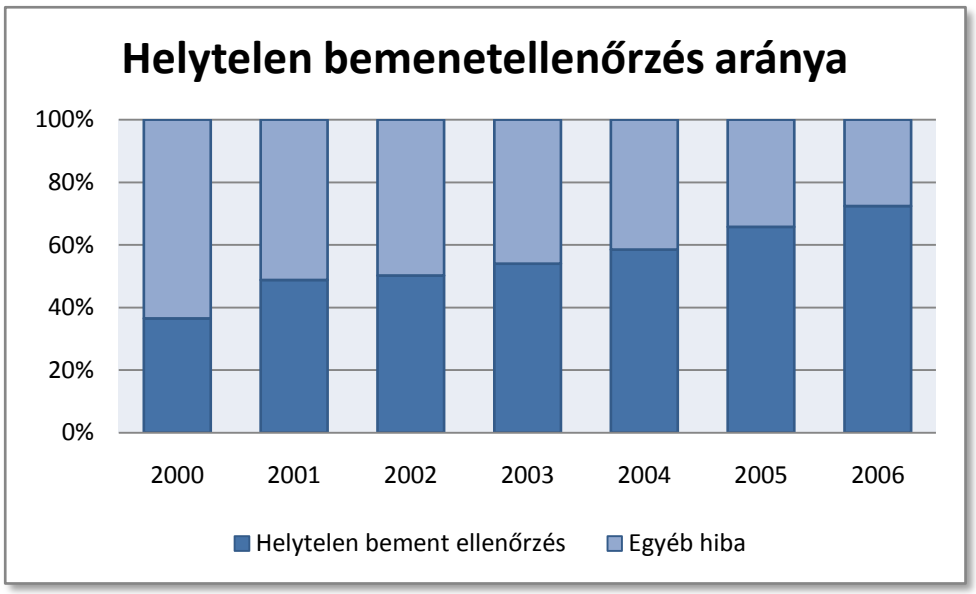
6 INTELLIGENS „FUZZING”

Hogy rávilágíthassak mennyire jelentősek a bemenet ellenőrzésből (input validation) származó biztonsági lyukak, térjünk vissza egy kicsit a bevezetőben ismertetett statisztikához. Most a helytelen bemenet ellenőrzés okozta sebezhetőségeket jelöltem más színnel a diagrammban.



19. ábra - Talált sebezhetőségek 2000-től

Hogy mekkora százalékot tesznek ki ezek a hibák, a második ábrán jobban látható.



20. ábra - Helytelen bemenet ellenőrzés aránya

Megfigyelhetjük, hogy a sebezhetőségek több mint fele ilyen típusú hibák miatt alakult ki, és az arány növekvő tendenciát mutat, ami ma már 72% körüli. Tehát mindenképpen érdemes a dinamikus, fekete doboz alapú tesztelést végrehajtani a szoftvereken, hiszen az pontosan ezeket hivatott felfedezni.

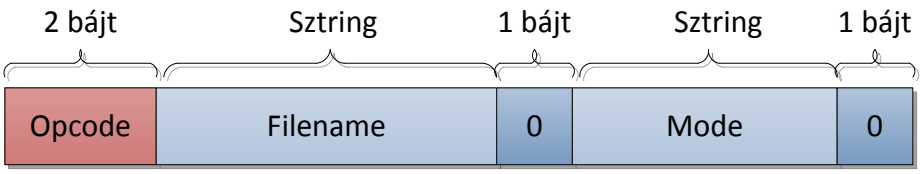
Vizsgáljuk meg tehát, hogy hogyan lehetne a Washingtoni Egyetemen alkalmazott módszert továbbfejleszteni a hatékonyság és az eredményesség növelése érdekében. Ha véletlenszerű adatot adunk a bemenetre, azt a legtöbb esetben már nagyon hamar elutasítja a szoftver, hiszen valószínűleg még csak nem is fog hasonlítani az érvényes bemeneti szintaktikához. Sokkal jobb lenne, hogyha inkább valamilyen „félig-érvényes" adatokat sikerülne valahogyan előállítanunk. Rögtön két kérdés merülhet fel. Az egyik, hogy honnan vegyük az adatokat, és hogy lesznek az azok félig érvényesek.

Az egyik megközelítés az, hogy mi hozzuk létre a tesztelésre szánt adatokat, vagy manuálisan, vagy pedig valamilyen nyelvtan definiálásával automatikusan generálva. A másik megközelítés, hogy helyes, érvényes adatokat gyűjtünk, és azokat manipuláljuk.

A másik kérdésre, vagyis hogy hogyan lesz az adat csak félig érvényes, a válasz nyilvánvalóan valamilyen heurisztika alkalmazása, amit a tesztvektorok előállításához, vagy a manipuláláshoz használunk. Ezt a tipikus hibák jellemzőinek ismeretével tudjuk megtenni, a gyakorlati tapasztalatunkat felhasználva. Ha például integer overflow típusú hibákat keresünk, akkor a bemenet azon részeire, amelyek valamilyen számot vagy méretet reprezentálnak, megpróbálunk mondjuk túl nagy számot tenni. Puffer túlcsorduláshoz vezethető hibák tesztelése esetén az általánosnál hosszabb sztringeket, hosszabb tömböket helyezünk el a bemenet megfelelő helyein, mezőin. Injection támadások lehetőségét vizsgálva vezérlőinformációkat szúrunk be az adatok közé és így tovább. Ezzel a technikával bármilyen típusú bemenetet manipulálhatunk, úgy mint fájlokat, hálózati forgalmat, a standard inputot, adatbázist, programkönyvtár API-t stb.

Felismerve az ilyen jellegű tesztelés hatékonyságát, mára már számos, valamilyen „fuzz"-tesztelésre alkalmas eszköz látott napvilágot [45]. Az egyik ilyen az Oului Egyetem biztonságos programozást kutató laborja által kifejlesztett PROTOS [46].

Ez az eszköz generálja a tesztvektorokat, méghozzá úgy, hogy a különböző inputok nyelvtanát BNF-ben (Backus-Naur Form) írja le. Ezt kiegészíti a heurisztikával, és utána a kiválasztott lehetséges mondatokat előállítja. A következő ábra a TFTP [47] protokoll Read Request és Write Requst üzenetének a felépítését és a PROTOS által feldolgozható BNF-ben leírt formáját mutatja.




```
# Request PDUs
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

<MODE> ::= ("octet" | "netascii") 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
```

21. ábra – TFTP üzenet

A PROTOS segítségével készült néhány teszt csomag, amelyek szabadon letölthetőek és használhatóak (pl. SIP vagy DNS protokollokra), de ezek nem módosíthatók és újabbak csomagok sem készíthetők. Létezik azonban néhány nyílt forráskódú fuzzer, illetve fuzzer írásához felhasználható programkönyvtár. Az egyik ilyen a SPIKE [48], amely segítségével viszonylag könnyen írhatunk tesztelő alkalmazást különböző protokollokhoz. Az SPIKE-al blokkokra oszthatjuk fel a küldendő üzeneteket. Ezeket a blokkokat a program lecseréli az előre definiálható lehetséges alternatívákkal, és a hossz mezőket is automatikusan frissíteni képes. Egy üzenetblokk megadása, előtte a méretével a következőképpen történhet:

```
s_size_string(“post”,5);
s_block_start(“Post”);
s_string_variable(“user=szekeresl”);
s_block_end(“post”);
```

22. ábra – Üzenetblokk megadása SPIKE-al

Egy másik hasonló, fuzzing API-t nyújtó keretrendszer a Peach [49]. Ezzel szintén gyorsan tudunk teszteseteket leírni egyszerűbb fájlformátumokra és

protokollokra. A bonyolultabb felépítésű fájlformátumok esetén, ahol az adatblokkok között összetettebb függőségek vannak már akadályokba ütközünk. Hasonlóan bonyolult protokollok tesztelése a fenti két eszközt használva csak úgy valósítható meg, ha szinte a teljes protokollt implementáljuk saját magunk, hiszen minden üzenetet manuálisan kell összeraknunk és elküldenünk.

7 A FLINDER KERETRENDSZER

Mi a FLINDER [50] keretrendszerrel, a PROTOS-al és a másik két említett fuzzerrel ellentétben, a tesztvektorok generálása illetve manuális magadása helyett, a helyes adatok manipulálását választottuk. Ezzel a megoldással bonyolultabb tesztelési feladatoknál, nagyon sok munka alól szabadíthatjuk meg a tesztelőt.

Az alap elgondolás az, hogy a tesztelőnek csak azt kelljen megmondani, hogy mit és milyen módszerrel szeretne manipulálni. Ezt a következő néhány alapötlettel tudjuk megvalósítani.

Tesztelés az adatformátum leírása alapján

A helyes bemeneti adatok szintaktikáját egy általános formátumleíró nyelv segítségével definiáljuk. Ennek felhasználásával értelmezni tudunk bármilyen adatot, legyen az egy fájl, vagy egy IP csomag, aminek a formátumát már egyszer leírtuk. Az értelmezés itt azt jelenti, hogy az adatot valamilyen egységes, kezelhető formátumra hozzuk, és így már könnyűszerrel tudunk e bemeneti struktúrákba hibát injektálni, a hiba hipotéziseink szerint.

Általános tesztelő algoritmusok

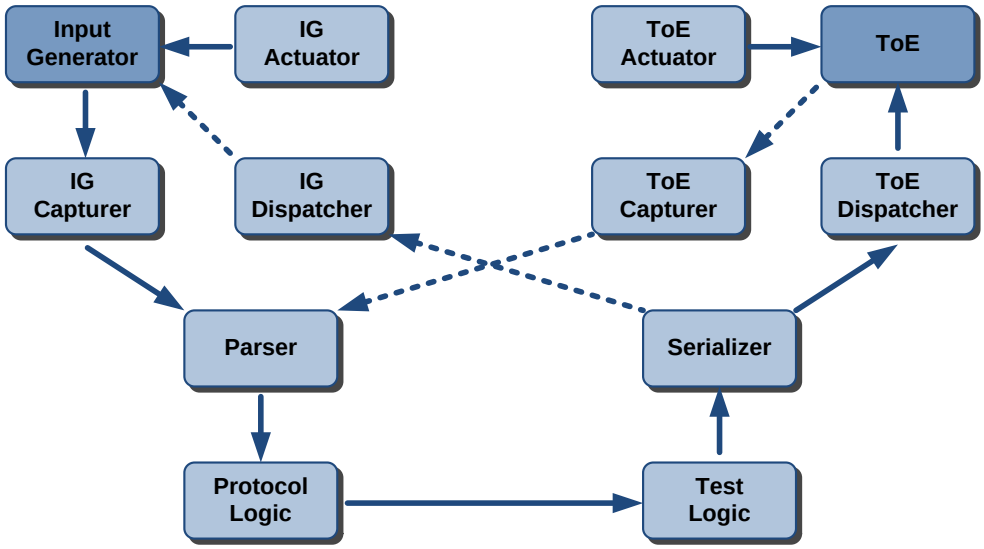
Mivel a manipulálandó adatainkat egy belső, általános struktúrává konvertáljuk, az így kapott adaton egységesen tudunk alkalmazni tesztvektor generálási mintákat. Vagyis általánosan használható algoritmusokat határozhatunk meg arra, hogy hogyan érdemes módosítani a bemenetre küldött üzeneteket annak érdekében, hogy egy bizonyos tipikus hibára fény derülhessen. Például előjelezési hibák felderítése esetén olyan számokat injektálunk hossz mezőkbe, amelyeket egy rossz implementáció nagy valószínűséggel negatívnak értelmezhet. Puffer túlcsordulást okozó hibákat megcélozva, fokozatosan növelve a változó hosszúságú mezők hosszát, könnyen válthatunk ki hibás működést. Ezeket az algoritmusokat ugyanúgy használhatjuk pl. egy ASN.1 DER [51] kódolású fájlon, mint egy webes alkalmazás XML [52] alapú protokolljának üzenetein.

Állapottartó protokollok támogatása

Összetettebb protokollok tesztelhetősége érdekében a FLINDER képes a protokoll állapotok követésére. Ennek érdekében egy állapotgépet tart fenn, és abban nyomon követi a protokoll folyását az üzenetcserék alapján. Az állapotgép az UML állapottérkép modelljére épül, ahol a tranzakciók a lehetséges üzenetküldéseket reprezentálják. Szabványos XML formátumba konvertált UML állapottérképeket tudunk importálni. A rendszer ennek segítségével tudja beazonosítani az egyes üzeneteket, így értelmezni tudja őket és a hibainjektálást véghez tudja vinni.

7.1 ARCHITEKTÚRA

Ahhoz, hogy átfogó képet kaphassunk a keretrendszerről, ebben az alfejezetben ismertetem annak architektúrális felépítését. A rendszer alapvetően jól elkülöníthető modulokból áll. Fontos látni, hogy a következőkben a modulokat általánosságban fogom bemutatni. Azok pontos működése ugyanis nagyon feladat, illetve környezetspecifikus lehet, ezért egy modulnak általában több implementációja vagy verziója létezik. A modulok kapcsolatát a következő ábra szemlélteti.



23. ábra – A FLINDER architektúra

Az ábrán a jobb felső sarokban a vizsgálat tárgyát, vagyis a tesztelendő rendszert jelöltem. Ezt „**Target of Evaluation**”-nek, rövidítve ToE-nak nevezzük. A helyes adatokat a bal felső sarokban látható **Input Generator** (IG) szolgáltatja. A ToE lehet egy hálózati alkalmazás, például egy szerver. Ilyenkor az IG egy, a szerver

elérésére alkalmas kliens. De lehet a kliens is ToE, ebben az esetben a szervert használhatjuk IG-ként. Természetesen nem csak hálózati protokollok manipulálására alkalmas a rendszer, lehet például a ToE egy szövegszerkesztő program is, az IG pedig egy helyes dokumentum. A FLINDER a ToE és az IG közé, „man-in-the-middle" szerűen beékelődve, az üzeneteket oly módon manipulálja, hogy azok minél nagyobb valószínűséggel okozzanak valamilyen anomáliát (deadlock, segmentation fault, protection error stb.).

Az IG és ToE **Actuator** kezeli az Input Generatorként használt alkalmazást és a tesztelendő alkalmazást. Ezeket a modulok indítják és állítják le őket, üzenetküldésre adnak parancsot, illetve detektálják az abnormális működést. E modulok pontos feladatai változók, vagy például amikor az Input Generatorunk egy egyszerű fájl, akkor természetesen az IG Actuatornak nincs is feladata.

A IG-tól kapott üzeneteket a IG **Capturer** elkapja (tipikusan hálózat lehallgatás vagy fájlolvasás segítségével) és a következő modul által feldolgozható formára hozza. Ez a formátum a ***BIME*** (Binary Message Envelope), amely a nyers adaton kívül tartalmaz még néhány kiegészítő adatot, mint például az üzenet iránya, időpecsét stb.

Az üzenet következő állomása a **Parser**, ami feldolgozza, értelmezi a kapott üzenetet úgy, hogy egy belső, általános fa struktúrába, az ún. ***MSDL***-be (Message Description Language) konvertálja. Ehhez természetesen szüksége van az adott formátum leírásra, amely küldött üzeneteket specifikálja. Ez a formátum leíró nyelv lesz az ***MFDL*** (Message Format Description Language).

Hálózati protokoll tesztelése esetén az MSDL ezután a **Protocol Logic**-hoz kerül, amely arra hivatott, hogy kövesse a futtatott protokoll állapotait és változóit ezáltal a tesztelés folyamán a rendszer tudja, hogy éppen hol tart a futtatott protokoll, érzékeli ha nem várt üzenet érkezik és hozzáférhetővé teszi a többi modul számára a protokoll specifikus információkat. Természetesen ehhez is szükséges egy, a protokoll állapotait és a tranzakciókat tüzelő üzeneteket specifikáló leírás.

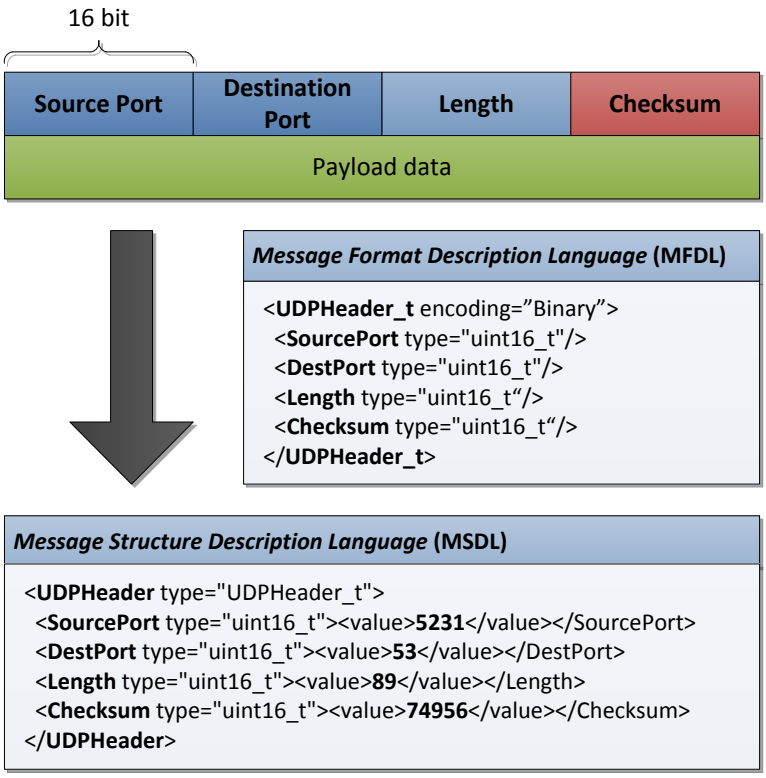
A lényegi munkát a **Test Logic** végzi. Ez a modul az, amely a már az MSDL-ben tárolt üzeneteket manipulálja a különböző általános algoritmusok szerint (ezek lesznek az ún. Test Suite-ok).

Mindezek után a Serializer visszaállítja azt az MSDL-t az eredeti formátumra, és végül a ToE **Dispatcher** küldi ki a végleges üzenetet a vizsgált rendszernek.

Látható, hogy a ToE-től a kliensnek küldött üzenetekkel is megtehető mindezt. A klienst és a ToE-t az Actuatorokkal vezérelve automatizálhatjuk az egész tesztelést. A megfelelő protokoll és adatformátum leírásokkal, és különböző teszt logikákkal, nagyon sokrétű, különböző típushibákra specifikáló teszteléseket tudunk végrehajtani.

7.2 MFDL ÉS MSDL

A Flinder tesztelő keretrendszert a Parser és Serializer modulok teszik könnyebben és jobban használhatóvá más biztonsági tesztelő alkalmazások vagy fuzzer-ekhez képest. Hogy jobban megérthessük, hogy hogyan működnek ezek a modulok, tekintsük meg a következő ábrát.



24. ábra – Példa MFDL és MSDL

Az ábra egy egyszerű UDP fejléc felépítését, MFDL leírását valamint a Parser kimeneteként kapott MDSL-t ábrázolja. A Parser az XML alapú MFDL formátumleírás alapján, a bináris adatcsomagot, az ábra alsó részén látható, szintén XML alapú MSDL reprezentációra alakítja át. A Test Logic az erre az alakra hozott adatokon tudja majd az algoritmusait lefuttatni. A Serializer pedig, a már módosított MSDL-t, ugyanaz az MFDL alapján fogja az adatokat szerializálni. Valójában a Parser bemenete és a Serializer kimenete a BIME, de az tulajdonképpen csak a bináris adat képernyőn megjeleníthető karakterekre konvertált változata, némi plusz információval kiegészítve.

7.3 „ACTION”-ÖK

A fájlformátumok illetve protokollok adatstruktúráiban nagyon sok függőség lehet. Gyakran alkalmazzák például a Type-Length-Value (TLV) struktúrák egymásbágyzását, ahol a hasznos információ a Value mezőben helyzkedik el, amelynek a hossza változó lehet. A hosszt a fix szélességű `Length` mező adja meg, a `Type` pedig az értékmező típusáról ad információt. Ilyenkor, ha például a teszt logikánk meghosszabbít egy `Value` mezőt, akkor az előtte lévő összes Length mezőt frissíteni kell. Vagy például a fent említett UDP csomag esetében, ha a `Payload` mezőt módosítjuk, nemcsak a `Length`, hanem a `Checksum` mezőt is újra kell számolnunk. Az ilyen és ehhez hasonló problémák kiküszöbölésére szolgálnak az MFDL-be ágyazható úgynevezett action-ök.

Ezeket az action utasításokat mind a Parser, mind a Serializer felhasználja. A TLV példát tekintve a Parser onnan tudja, hogy meddig tart egy Value mező, hogy az MFDL-ben az adott mezőhöz rendelünk egy action-t, ami megmondja, hogy annak a hossza egyenlő az előző `Length` mező értékével. A módosítások után pedig a Serializer úgy képes frissíteni a `Length` mezőket, hogy azokhoz egy action van rendelve, amely megmondja, hogy a mező értéke legyen egyenlő az azt követő Value mező hosszával. Nem csupán adathosszakat reprezentáló adatelemeknél kell action-öket alkalmaznunk. Gyakran alkalmaznak ellenőrző összegeken kívül például hash értékeket vagy digitális aláírást a fájlformátumok, illetve protokollok. Ezen adatmezők frissítését is automatizálni tudjuk az actionök segítségével. Az MFDL-ben definiált actionök Python [53] nyelven implementálhatóak. A

bonyolultabb műveleteket, mint például titkosítás vagy tömörítés, ily módon könnyen és újra felhasználható módon tudjuk megvalósítani. Aszerint, hogy mikor kell végrehajtani egy actiont, négy alapvető típust különböztetünk meg. Az úgynevezett `preParseAction` közvetlenül a parsolás előtt, míg a `postParseAction` közvetlenül a parsolás után hívódik meg. Hasonlóan a `preSerializeAction` a szerializáció előtt és a `postSerializeAction` az után kerül végrehajtásra.

7.4 TESZT HIERARCHIA

A FLINDER-rel való tesztelés folyamatát könnyebben átláthatjuk a tesztek hierachiájának megismerésével. A keretrendszer tervezésénél figyelembe vettük, hogy bonyolult protokollok tesztelésére is alkalmas legyen, ezért volt szükség az állapotok követésének képességére. Valójában, amikor egy egyszerű állapotnélküli protokollt tesztelünk, például amikor egy fájl módosításával hozzuk létre a tesztvektorainkat, akkor is definiálunk a rendszer számára egy állapotgépet, amelynek egy állapota van (a START állapoton kívül). Ilyenkor az egyetlen lehetséges üzenetet reprezentálja a START-ból az egyedüli állapotba vezető tranzakció. Azonban, hogy összetettebb feladatokat is könnyűszerrel kezelhessünk, a tesztelésnek a következő hierarchiáját állítottuk fel. **Test Step**-nek nevezük egyetlen üzenet átvitelét, vagyis az állapotgépen történő egyetlen lépést. Egy **Test Case** az állapotgép egyszeri végigjárása, és eközben egy vagy több speciális módosítás megtétele. **Test Suite**-nak nevezünk egy tesztelő algoritmus teljes lefutásához szükséges Test Case-ek halmazát. Végül ezeket a Test Suite-okat egy **Test Package**-ben foglalhatjuk össze.

7.5 TESZTELŐ ALGORITMUSOK

A Parser és a Serializeren kívül, a Flinder legfontosabb modulja a TestLogic. Ez a modul futtatja le az általános tesztalgoritmusokat az MSDL-en. Ezek a tesztalgoritmusok olyan módosításokat hivatottak előidézni, amelyek a vizsgált szoftverben anomáliákat vagy nem kezelt kivételeket okoznak. Az ilyen anomáliákból tudunk majd később biztonsági hibákra következtetni. Ehhez általában valamilyen határértékekre vagy szélsőértékekre kell változtatnunk az egyes adattagokat. Ilyen értékek például numerikus adatok esetén a nulla, a

negatív, vagy a számábrázolás által lehetővé tett nagyon kicsi illetve nagyon nagy számok. Változó hosszúságú adatmezők estén a hosszal tudunk elsősorban operálni. Problémát okozhatnak a túl hosszú, vagy az éppen nulla hosszú mezők vagy lezáró `0x00` nélküli C típusú karakterláncok. Ha struktúrákban gondolkozunk, akkor a határértékek lehetnek a sokszor ismétlődő vagy mélyen (rekurzívan) egymásba ágyazott struktúrák. Ezen kívül természetesen az olyan speciális hibák felderítéséhez, mint például a `printf()` hiba vagy az SQL injection, speciális tesztvektor generálási algoritmusokra van szükség. A FLINDER-ben jelenleg a következő teszt algoritmusok (Test Suite-ok) vannak implementálva.

Buffer Overflow Test Suite

A kiválasztott változó hosszúságú mezők hosszát egy megadott maximális hosszig az algoritmus minden újabb Test Case-ben meghosszabbítja és kitölti az új hossz által lefedett fennmaradó helyet a bemeneti adatstruktúrában.

Integer Overflow Test Suite

Az algoritmus elsősorban nem megfelelő castolásból, aritmetikai túlcsordulás és előjelezési hibákból eredő anomáliákat kísérel meg kiváltani a tesztelt programból. Az algoritmus a tesztelendő adatmező bitszélességének megfelelően generálja a módosított tesztvektorokat.

Format String Test Suite

A `printf()` formátumleírás helytelen használatából származó hibák megtalálására alkalmas teszt algoritmus. Speciális, formátumleíráshoz használt karaktereket illeszt a bemeneti struktúra megfelelő mezőibe úgy, hogy ha azokat a `printf()` függvény első paramétereként használ fel a programozó, akkor az kivételt okoz.

Dictionary Test Suite

Egy előre definiált szótárból helyettesít be adatokat a manipulálni kívánt adatmezőbe. Olyan esetekben használható jól, amikor a tesztelt szoftver

bementi nyelvében léteznek bizonyos kulcsszavak, amelyek összekeverés vagy ismételése okozhat hibát.

Certificate Test Suite

A tanúsítványtesztelő algoritmus egy ASN.1 DER kódolású X.509-es tanúsítvány módosításával, mutálásával hozza létre az újabb teszt vektorokat. A módosított teszt tanúsítványok számos különböző hibatípust rejtenek magukban, mint például érvénytelen DER fejlécek, érvénytelen adatok, nem megfelelő tanúsítvány struktúra, helytelen aláírás, értelmetlen érvényességi adtok stb.

7.6 HIBADETEKTÁLÁS

A tesztelő algoritmusaink alkalmazása során, ha egy hiba előjön, azt valamilyen módszerrel detektálnunk kell. A hibajelenség általában az, hogy a program „elszáll", nem várt módon kilép, nem várt vagy rossz üzenetet küld, „lefagy", time-out-ol, vagy végtelen ciklusba kerül és 100%-osan leterheli a processzort. A processzor használatot tudjuk valamilyen teljesítménymonitorozó eszközzel figyelni. A végzetes hibákat pedig a legkönnyebben egy debugger csatlakoztatásával tudjuk detektálni, amely plusz információkkal is tud szolgálni a hiba eredetéről. Léteznek olyan debugger/profiler eszközök is, melyek kifejezetten memória kezelés során fellépő hibák detektálására specializáltak. Ilyen például a Linux operációs rendszerek használható Valgrind [54]. Ez az eszköz tulajdonképpen egy virtuális gép, amely a rajta futtatott programokat először átalakítja úgy, hogy futás közben ellenőrizni tudja bizonyos feltételek teljesülését. Ilyen feltétel lehet például, hogy memória írás vagy olvasás csak helyesen lefoglalt területen történjen, vagy hogy a heap-en lefoglalt területek felszabaduljanak. Ezzel a technikával a Valgrind detektálni képes olyan memóriafolyásokat, és túlírásokat, amikre hasonló eszköz használata nélkül nem utalna semmilyen külső jel.

8 MÉDIALEJÁTSZÓ ALKALMAZÁSOK TESZTELÉSE

A FLINDER alkalmasságát olyan bonyolult protokoll implementációk tesztelésére, mint az SSL/TLS már bizonyítottam [55]. Ezúttal a keretrendszer gyakorlati alkalmazását, elsősorban médialejátszó alkalmazások tesztelésén keresztül fogom bemutatni ebben a fejezetben. Köztudott, hogy ma az Internet forgalom több mint 2/3-át P2P forgalom[5] teszi ki. Ezen forgalom szinte teljes egészében pedig videó, hang és kép fájlok átvitelében merül ki. Egy médialejátszó alkalmazás biztonsága éppen ezért kritikusnak mondható, hiszen ha például egy népszerű zenelejátszó programban sebezhetőségre talál egy támadó, akkor a P2P hálózaton való terjesztéssel könnyen, számtalan számítógépet képes támadni. Ilyen és ehhez hasonló támadással lehetséges olyan úgynevezett botnet-ek kialakítása, amelyek például elosztott szolgáltatásmegtagadásos támadásra vagy kéretlen levelek terjesztésére használható fel.

8.1 MULTIMÉDIÁS FÁJLFORMÁTUMOK ÉS MEGJELENÍTŐK

A tesztelésbe bevont fájlformátumok kiválasztásánál szempont volt a veszély nagysága, vagyis, hogy egyrészt mennyire használt, mennyire elterjedt a formátum, másrészt pedig, hogy milyen egyszerűen lehet támadni vele, más szóval milyen egyszerű a támadónak a manipulált bemenetet az áldozathoz eljuttatni. A napjainkban legelterjedtebben használt multimédiás adatokat tartalmazó hat úgynevezett konténer formátum a következő:

- *Resource Interchange File Format (RIFF)*: Az egyik legrégebbi szabvány, ebbe a csoportba tartozik például az AVI, a WAV valamint az ANI fájlformátum.
- *QuickTime Container Format (MOV)*: Az Apple szabványa, amelyre az MP4, továbbá annak egyszerűsített változata, a 3GP is épül.

[5] A Peer-To-Peer itt olyan fájlcserélő szolgáltatásokat jelent, mint például a BitTorrent vagy az EDonkey hálózat.

- *Advanced Systems Format (ASF)*: A Microsoft szabványa multimédia fájlok tárolására. E csoport tagja a Windows Media Audio/Video fájlformátum (WMA, WMV).
- *Real Media (RM)*: A RealNetworks internetes tartalomszolgáltató formátuma. Két fő képviselője a Real Media Audio illetve Video (RV, RA).
- *Flash Video (FLV)*: Webes megjelenítésre optimalizált mozgókép formátum, általában SWF fájlba beágyazva találkozhatunk vele. Ilyen formátumba vannak kódolva például a YouTube vagy a Google Video webszájtokon található videók.
- *Ogg Media File (OGG)*: Elterjedőben lévő, teljesen nyílt hang, illetve mozgókép konténer formátum szabvány.

Én hat fájlformátumot választottam ki, melyeknek elkészítettem a formális leírását az MFDL nyelven, a tesztvektorok generálása céljából. Ebből kettő videó formátum: ezek az Audio/Visual Interleaved (**AVI**) formátum és a Windows Media Video (**WMV**). Kettő hanghordozó formátum: a Waveform data (**WAV)** valamint a Windows Media Audio (**WMA**). Ezen kívül a hagyományos állóképet kódoló fájlformátumok helyett, két speciálisabb formátumot választottam vizsgálatom tárgyául. Az egyik az Animated Cursor (**ANI**) formátum. Ennek egyik oka az volt, hogy ellenőrizzem, hogy egy nemrégiben publikált [56], és sokszor és sok helyen kihasznált biztonsági hibát képes-e a FLINDER újra megtalálni. Másik oka pedig, hogy az ennek a formátumnak a kezeléséből adódó sebezhetőség valóban nagy kockázatot jelent, hiszen weboldalakba ágyazva, könnyen kihasználható a hiba támadás céljából. Az utolsó választásom pedig a True Type Font (**TTF**) formátumra esett, mert az szintén weboldalakba, Word és PDF dokumentumokba ágyazható be.

A tesztelés alá vetett szoftverek kiválasztásánál is az elsődleges szempontom a elterjedtségük volt. A következő négy alkalmazásra esett a választásom:

- Windows Media Player 9.00
- VLC Media Player 0.8.6
- WinAmp 5.11
- IrfanView 3.95

8.2 TESZTVEKTOROK ELŐÁLLÍTÁSA

Mivel ezúttal egy tesztvektort (egy manipulált fájlt) többször is felhasználtam, a tesztelést két fázisra bontottam. Az első fázisban létre hoztam a teszteseteket anélkül, hogy azt ténylegesen el is küldtem volna azokat a tesztelt alkalmazás bemenetére. A generált tesztfájlokat a különböző formátomokra egy gyűjtőkönyvtárba szerializáltattam. Így a következő fázisban már az előkészített teszt fájlokat újra fel tudtam használni a vizsgált négy alkalmazás tesztelésénél. Ehhez el kellett készítenem a választott fájlformátumokra az MFDL leírást. Valójában a hat MFDL helyett, számomra elég volt csak öt ugyanis a WMA és WMV fájlok ugyanazt az Advanced System Formátumot (ASF) használják. Ennek a formális leírása megtalálható a mellékletben. Az MFDL leírások a hivatolos specifikációk alapján készültek. A FLINDER keretrendszer használatában a formátumleírások elkészítése a legmunkásabb és legidőigényesebb feladat. Azonban ha valaki egyszer már leír egy fájlformátumot, vagy protokollt annak specifikációja alpján, az újra felhasználható lesz. Segítségükkel később új szotvereket tesztelhetünk, új tesztcsomagokat készíthetünk, vagy más Input Generátorokat használhatunk a tesztesetek előállításához[6]. Számomra az öt MFDL elkészítése nagyjából másfél hetes munktát vett igénybe.

Ezt követően, az elkészült MFDL leírásokban, az általam érdekesnek tartott adatmezőkhöz tesztelő algoritmusokat rendeltem, amelyek alapján a TestLogic végrehajthatja a módosításokat. Az „érdekes” adatmezők kiválasztása a hibahipotézisek alaján történik. Érdkesek például a hossz mezők, a változó hosszúságú mezők, a karakterláncok, a típusazonosítók stb. A hozzárendeléseket a mellékletben is megtalálható `<meta Manipulator/>` tagok segítségével lehet megtenni. A tesztesetek előállításához a következő általános algoritmusokat, vagyis TestSuite-okat használtam fel:

[6] Akár egymásba is ágyazhatunk MFDL-eket. Például képzeljünk el egy képnézegető programot, amely csak webes képmegosztó szolgáltatásokon megjelent képek böngészésére alkalmas. Ilyen esetekben, ha például a JPEG kezelést szeretnénk vizsgálni, a JPEG-et specifikáló MFDL-t a HTTP protokoll MFDL-jébe ágyazhatjuk.

- *Buffer Overflow Test Suite*: változó hosszúságú adatmezők és karakterláncok esetén (ASCII vagy UNICODE).
- *Integer Overflow Test Suite*: hosszt, elemszámot, eltolást, offszetet kódoló adatmezők esetén.
- *Format String Test Suite*: képernyőn megjeleníthető kódolást használó adatmezők esetén.
- *Dictionary Test Suite*: formátumspecifikus kulcsszvak, varázsszavak, típusazonosítók esetén, a belőlük készített szótár felhasználásával.

A fenti TestSuite-ok lefuttatása a kiválasztott adatmezőkön, a különböző formátumok esetében, a következő táblázatban látható teszvektor számot eredményezte:

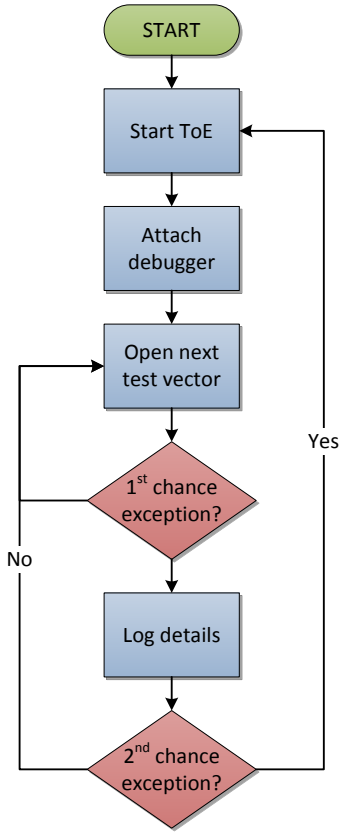
Formátum	Tesztvekorok száma
AVI	1740
WMV	2260
WAV	1190
WMA	2180
ANI	1620
TTF	760

1. táblázat – Generált tesztvektorok száma összesen

A generált tesztesetek számának aránytalansága, a különböző formátumok eltérő bonyolultágának köszönhető, illetve annak, hogy különböző számú „érdekes" adatmazők találhatók meg az egyes specifikációkban.

8.3 TESZTELÉS MENETE

A második fázis a tényleges tesztelés fázisa volt. Egyszerre egy fájlformátum-alkalmazás párost teszteltem. A tesztelés levezérlésére Pythonban implementáltam az Actuatorként működő, a tesztelést vezérléséért felelős programot, amely leegyszerűsített folyamatábrája a következő képen látható.



25. ábra – Actuátor folyamatábrája

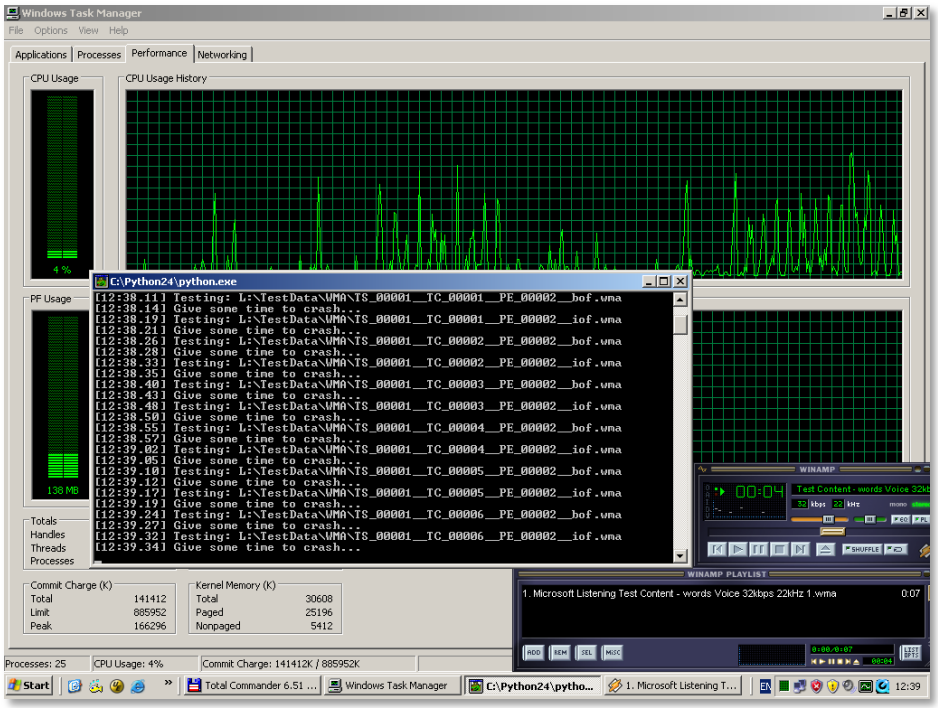
Az Actuatort felparaméterezve a vizsgált program, valamint az előre elkészített tesztvektrokat tartalmazó könyvtár elérési útjával, az az összes könyvtárban található tesztfájlt megnyitatta a ToE-val, és közben debugger csatlakoztatásával figyelte, hogy történik-e kivétel. A debugger feladatok ellátására a Win32 Debug API-t, illetve az arra épülő PyDbg Python modult [57] használtam fel. A degugger segítségével lehetőségünk van minden fellépő kivételt elkapni, még mielőtt az alkalmazás kezelné azt (first chance exception). Ha az alkalmazás nem kezel egy kivételt, akkor a debugger ismét értésítést kap (second chance exception). Ilyenkor a debugger jelenléte nélkül általában az alkalmazás összeomlik, kilép. Számunka a különböző fatális kivételek közül az illetéktelen memóriahozzáférést jelző (`EXCEPTION_ACCESS_VIOLATION`) kivétel lesz fontos, ugyanis illegális memóriaműveletek esetén általában ez a kivétel lép fel, és a kihasználhtaó biztonsági hibák, mint tudjuk, a legtöbször memóriakezelési hibákból adódnak. Amikor a programunk nem kezel egy ilyen fatális hiábát, akkor a hibából már nem tud visszatérni („elszáll”), ezért ezesetben újra kell indítanunk a tesztelt szoftvert.

Az Actuatort megvalósító Python modult `test_runner_module`-nak neveztem el, és azt az egyes fájlformátum-lejátszó párokhoz a követező ábrán látható módon paramétereztem fel. Így módon egy-egy önmagában indítható szkript segítségével tudtam elindítani a tesztelést.

```
import test_runner_module

# Settings ##########################################################
targetName      = "i_view32.exe"
startCommand    = "C:\\Program Files\\IrfanView\\i_view32.exe"
testCaseDir     = "D:\\TestData\\ANI"
crashBin        = "D:\\TestLogs\\iview_ani.bin"
#####################################################################

actuator = test_runner_module.testRunner(targetName, startCommand,\
                                         testCaseDir, crashBin)
actuator.startTesting()
```

26. ábra – Actuátor konfigurációja

A következő ábrán WinAmp WMA fájlokkal való tesztelése alatt látható képernyőkép látható.



27. ábra – Képernyőkép a WinAmp tesztelése közben

A következő táblázat foglalja össze a lefuttatott tesztesetek számát.

	AVI	WMV	WAV	WMA	ANI	TTF	Összesen
Windows Media Player	1740	2260					**4000**
VLC Media Player	1740	2260	1190	2180			**7370**
WinAmp	1740	2260	1190	2180			**7370**
IrfanView					1620	760	**2380**
Összes teszteset							**21120**

2. táblázat – Tesztesetek száma szoftver-fájlformátum páronként

Összesen 21120 teszteset futott le. A táblázatnak azon cellái, melyekben nincs szám, azt jelölni, hogy azt a formátum-ToE párost nem teszteltem. A tesztelés 3 számítógépen futott párhuzamosan és nagyjából egy hetet vett igénybe.

8.4 Eredmények

Az Actuátort úgy valósítottam meg, hogy a kivételek pontos részleteit bináris formában naplózza, méghozzá a kivétel fellépésének helye szerint csoportosítva. Sokszor nagyon sok különböző tesztvektor ugyanazt a hibát idézi elő. Számunkra elsősorban a talált hibák érdekesek, ezért a naplóban szivesebben látjuk a hibák előbukkanásának különböző helyeit, mint egyenként az összes tesztvektort, ami hibát okozott. A bináris naplót olvasható formára hozva, például a VLC Media Player – Windows Media Audio esetén, a következőt láthattuk:

```
[1] ntdll.dll:7c910f29 mov ecx,[ecx] from thread 196 caused access violation
        TS_00001__TC_01126__PE_00002__iof.wma,

[17] libasf_plugin.dll:6a186c43 mov byte [ebx+ecx],0x0 from thread 2032 caused access
violation
        TS_00001__TC_00982__PE_00002__iof.wma, TS_00001__TC_00983__PE_00002__iof.wma,
        TS_00001__TC_00984__PE_00002__iof.wma, TS_00001__TC_00985__PE_00002__iof.wma,
        TS_00001__TC_00986__PE_00002__iof.wma, TS_00001__TC_00987__PE_00002__iof.wma,
        TS_00001__TC_00988__PE_00002__iof.wma, TS_00001__TC_00989__PE_00002__iof.wma,
        TS_00001__TC_00990__PE_00002__iof.wma, TS_00001__TC_00991__PE_00002__iof.wma,
        TS_00001__TC_00992__PE_00002__iof.wma, TS_00001__TC_00993__PE_00002__iof.wma,
        TS_00001__TC_00994__PE_00002__iof.wma, TS_00001__TC_00995__PE_00002__iof.wma,
        TS_00001__TC_00996__PE_00002__iof.wma, TS_00001__TC_00997__PE_00002__iof.wma,
        TS_00001__TC_00998__PE_00002__iof.wma,

[44] libasf_plugin.dll:6a18994c mov ebx,[edx+0x34] from thread 988 caused access
violation
        TS_00001__TC_01034__PE_00002__iof.wma, TS_00001__TC_01036__PE_00002__iof.wma,
        TS_00001__TC_01038__PE_00002__iof.wma, TS_00001__TC_01040__PE_00002__iof.wma,
        TS_00001__TC_01042__PE_00002__iof.wma, TS_00001__TC_01044__PE_00002__iof.wma,
        TS_00001__TC_01046__PE_00002__iof.wma, TS_00001__TC_01048__PE_00002__iof.wma,
        TS_00001__TC_01050__PE_00002__iof.wma, TS_00001__TC_01052__PE_00002__iof.wma,
        TS_00001__TC_01054__PE_00002__iof.wma, TS_00001__TC_01056__PE_00002__iof.wma,
        TS_00001__TC_01058__PE_00002__iof.wma, TS_00001__TC_01060__PE_00002__iof.wma,
        ...
```

28. ábra – Napló az kivételekről csoportosítva

Látható a hibahelyek szerinti csoportosítás, valamint alatta a hibát okozó tesztvektorok fájlnevei. Ha egy konkrét teszteset adataira vagyunk kíváncsiak, akkor azt is megtekenthetjük. Erre ad példát a következő ábra.

```
libasf_plugin.dll:6a186c43 mov byte [ebx+ecx],0x0 from thread 1184 caused access
violation when attempting to write to 0x00000000

CONTEXT DUMP
  EIP: 6a186c43 mov byte [ebx+ecx],0x0
  EAX: 00000000 (         0) -> N/A
  EBX: 00000000 (         0) -> N/A
  ECX: 00000000 (         0) -> N/A
  EDX: 0000000a (        10) -> N/A
  EDI: d9fd0001 (3657236481) -> N/A
  ESI: 00000000 (         0) -> N/A
  EBP: 00000074 (       116) -> N/A
  ESP: 0201ee88 (  33681032) -> ~  wx>hPxhPbjj`j|P~lYj~hPjj[wHgDLhP$|\lP\lPtP`PhP
(stack)
  +00: d9fd0001 (3657236481) -> N/A
  +04: 00000018 (        24) -> N/A
  +08: 000000fc (       252) -> N/A
  +0c: 00c07e10 (  12615184) -> 6b~f|~~@c>h,bpb /bbi{0x> (heap)
  +10: 20000000 ( 536870912) -> N/A
  +14: 77c12088 (2009145480) -> N/A

disasm around:
        0x6a186c20 mov [ecx+esi+0x2],dl
        0x6a186c24 mov eax,[esp+0x2c]
        0x6a186c28 mov edi,[esp+0x44]
        0x6a186c2c movzx edx,[ebp+eax+0x6]
        0x6a186c31 add ebp,0x8
        0x6a186c34 mov [ecx+edi+0x3],dl
        0x6a186c38 add ecx,0x4
        0x6a186c3b cmp ebx,ecx
        0x6a186c3d jg 0x6a186bf2
        0x6a186c3f mov ebx,[esp+0x44]
        0x6a186c43 mov byte [ebx+ecx],0x0
        0x6a186c47 jmp 0x6a186920
        0x6a186c4c mov esi,[esp+0x2c]
        0x6a186c50 xor ebx,ebx
        0x6a186c52 add esi,ebp
        0x6a186c54 movzx ecx,[esi+0x6]
        0x6a186c58 add ebp,0x8
        0x6a186c5b movzx eax,[esi+0x7]
        0x6a186c5f mov edi,ecx
        0x6a186c61 mov edx,eax
        0x6a186c63 movzx ecx,[esi+0x5]
```

29. ábra – Konkrét teszteset által okozott kivétel részletei

8.4.1 *Számtalan „megálló” hiba*

A tesztelés során rengeteg váratlan kilépést okozó hibát tapasztaltam. A következő táblázat foglalja össze az eredményeket:

	AVI	WMV	WAV	WMA	ANI	TTF	Átlag
Windows MediaPlayer	2%	1%					**1,5%**
VLC Media Player	12%	21%	4%	18%			**13,75%**
WinAmp	23%	17%	7%	13%			**15%**
IrfanView					1%	5%	**3%**
Átlagosan	**15,7%**	**13%**	**5,5%**	**15,5%**	**1%**	**5%**	**10,3%**

3. táblázat – Hibát okozó tesztesetek aránya

Hogy ezekből a programterminálást okozó hibákból mennyi az, ami kihasználható biztonsági hiba, az további vizsgálatot igényel.

Idő hiányában csak két ilyen részletes vizsgálatra volt időm.

8.4.2 *Animált kurzor sebezhetőség*

A fejezet elején említettem, hogy az ANI formátumot azért választottam, mert egy imsert sebezhetőséget szerettem volna újra felfedezni. A FLINDER megtalálta a hibát, amlynek részleteit ebben a részben boncolgatom.

Az ANI fájlformátum, hasonlóan a többi RIFF típusú fájlformátumhoz úgynevezett „chunk”-okból áll. A chunk-ok változó méretű adatokat tárolhatnak. Tekintsük az ANI fájlformátumot leíró MFDL egy részletét a következő képen.

```
<ANI_File encoding="Binary">
  <Riff_ID type="string" msdl:bytesToLoad="4"/> <!--RIFF-->
  <File_Size type="uint32"/>
  <File_Type type="string" msdl:bytesToLoad="4"/> <!--ACON -->
  <ANI_Info_List type="ANI_Info_List_t"/>
  <ANI_Header_Chunk type="ANI_Header_Chunk_t"/>
  <ANI_Icon_List type="ANI_Icon_List_t"/>
</ANI_File>

<ANI_Header_Chunk_t>
  <Chunk_ID type="string" msdl:bytesToLoad="4"/> <!--anih-->
  <Chunk_Size type="uint32"/>
  <Size_Of type="uint32"/>
  <Frames type="uint32"/>
  <Steps type="uint32"/>
  <Width type="uint32"/>
  <Height type="uint32"/>
  <Bit_Count type="uint32"/>
  <Num_Planes type="uint32"/>
  <Display_Rate type="uint32"/>
  <Flags type="uint32"/>
  <Padding type="binContainer" msdl:bytesToLoad="0">
    <preParseAction>
      top.bytesToLoad = "%d" % (top.parent.Chunk_Size.value % 2)
    </preParseAction>
  </Padding>
</ANI_Header_Chunk_t>
```

30. ábra – Részlet az ANI MFDL formátumleírásból

Az első rész magát a fájl felépítését írja le, a második pedig az „anih” címkével ellátott fejlécet. Minden chunk a RIFF formátumú fájloknál egy 4 bájtos címkével kezdődik, majd a következő négy bájt kódolja a chunk hosszát.

A USER32.DLL `_LoadCursorIconFromFileMap()` függvénye felel az ANI fájlok memóriába való beolvasásáért. Ebben a függvényben már 2005-ben találtak [58] és javítottak [56] egy stack overflow típusú hibát. Akkor a hibát az okozta, hogy a program nem ellenőrizte az ANI fejléc hosszát, mielőtt azt egy fix méretű tömbbe másolta volna a stack-en. Az 31. ábra látható kódrészlet illusztrálja a hibás kódrészletet. A `ReadTag()` a következő chunk címkéjét és méretét, azaz a következő 8 bájtot olvassa be. A header nevű struktúra pedig fixen 36 bájtot foglal a stack-en.

```
ReadTag(file, &chunk);
if (chunk.tag == 'anih')
{
  if (chunk.size != 36)
    return 0;
  // Beolvas chunk.size darab bájtot a header struktúrába
  ReadChunk(file, &chunk, &header);
}
```

31. ábra – Hibás kódrészlet

A 4. sorban található if az MS05-002-es sorszámmal kiadott Windows patch-el került csak a kódba. Az if nélkül egyértelműen látható, hogy a művelet rendkívül egyszerűen kihasználható volt a visszatérési cím felülírásával. A javítás azonban nem volt tökéletes.

A fejléc beolvasása után, a `_LoadAniIcon()` függvényben, a fájlban tárolt további chunk-ok is beolvasásra kerülnek. Ez a függvény tulajdonképpen egy nagy switch-ből áll, ahogyan a következő pszeudokód is mutatja.

```
void _LoadAniIcon()
{
  while (EoF)
  {
    ReadTag(file, &chunk);
    switch (chunk.tag)
    {
    case 'seq ':
      ...
    case 'LIST':
      ...
    case 'rate':
      ...
    case 'anih':
      // Beolvas chunk.size darab bájtot a header struktúrába
      ReadChunk(file, &chunk, &header);
...
```

32. ábra – ANI fájl parsolásának pszeudokódja

Látható, hogy az „anih” címkével ellátott fejléc itt is beolvasásra kerül, az előzőhöz hasonló módon, viszont ezúttal már nincs ellenőrzés (a 2005-ös javított verzióban sem), hiszen, gondolhatnánk, már az elején leellenőriztük. A problémát az okozza, hogy az előző `_LoadCursorIconFromFileMap()` függvény csupán a fájl elején lévő első „anih” fejlécet ellenőrzi, hogy megfelelő-e a hossza, viszont ha valaki elhelyez egy újabb ilyen chunk-ot, azt már csak a `_LoadAniIcon()` fogja

beolvasni ellenőrzés nélkül. Így a második ANI fejlécet beszúrva és annak hosszát 36-nál nagyobbra választva a támadó újfent sikerrel járhat.

Erre a hibára a Dictionary Test Suite által generált tesztvektor segítségével derült fény. A szótárban a RIFF-specifikus cimkéket rögzítettem, amelyeket a TestSuite a `Chunk_ID` mezőkbe illesztett. Így létrejöttek olyan esetek amikor „anih" chunkból több is volt egy fájlban, érvénytelen hosszal.

8.4.3 True Type Font sebezhetőség

Amikor az IrfanView programot módosított True Type betűtípus fájlok segítségével teszteltem, az egyik tesztesetnél a Windows XP operációs rendszer STOP hibával[7] leállt. A jelenség reprodukálható volt, sőt mi több, nem csak abban az egy esetben, hanem számos további teszteset során is hasonló jelenség következett be.

A TTF betűkészlet fájlok elején található egy úgynevezett „Table Directory", amely a tartalmazott adattáblák azonosítását, ellenőrző kódját, ofszetjét és hosszát tartalmazzák. Azokban a tesztesetekben történt rendszerhiba, amelyekben olyan TTF fájlt olvastattam be, ahol az ofszet illetve a hossz túl nagy értére módosult.

Type	Name	Description
uint32_t	Tag	4 -byte identifier.
uint32_t	CheckSum	CheckSum for this table.
uint32_t	Offset	Offset from beginning of TrueType font file.
uint32_t	Length	Length of this table.

4. táblázat – Hibát okozó tesztesetek aránya

Ezekben az esetekben a hiba oka a kék képernyő szerint a következő volt: „`PAGE_FAULT_IN_NONPAGED_AREA`". Hogy alaposabban megvizsgálhassam, hogy mi történt valójában, rendszerhiba esetén teljes memóriaképet kértem az operációs rendszertől, hibakeresési információ gyanánt. A lementett memóriaképet később a WinDbg debugger segítségével elemeztem. Megtudtam, hogy a kivétel keletkezésekor a programvezérlés éppen a `win32k!sfac_CopyFontAndPrePrograms+6b` címen volt. Ez azt jelenti, hogy a

[7] A STOP hiba, ismertebb nevén „Kék Halál" vagy angolul „Blue Screen of Death", egy olyan hiba vagy kivétel a rendszermagban, amely után a rendszer már nem tud helyreállni.

win32k.sys nevű bináris sfac_CopyFontAndPrePrograms()[8] függvényén belül történt a kivétel. A debugger-től azt is megtudtam, hogy ezen belül pontosan melyik utasítás végrehajtásánál jártunk, mikor a hiba bekövetkezett:

```
bf84ee98 f3a5    rep movs dword ptr es:[edi],dword ptr [esi]
```

Ezután a IDA Pro disassembler segítségével visszafejtettem a win32k.sys-t, hogy könnyebben átláthassam a szóban forgó kódrészletet. A hibát okozó utasítás környezetében levő programkód a következő volt:

```
.text:BF84EE88 loc_BF84EE88:
.text:BF84EE88                 test    esi, esi
.text:BF84EE8A                 jz      short loc_BF84EEA8
.text:BF84EE8C                 mov     edi, [ebp+8]
.text:BF84EE8F                 mov     ecx, esi
.text:BF84EE91                 mov     edx, ecx
.text:BF84EE93                 shr     ecx, 2
.text:BF84EE96                 mov     esi, eax
.text:BF84EE98                 rep     movsd
.text:BF84EE9A                 mov     ecx, edx
.text:BF84EE9C                 and     ecx, 3
.text:BF84EE9F                 push    eax
.text:BF84EEA0                 rep     movsb
.text:BF84EEA2                 call    dword ptr [ebx+8]
.text:BF84EEA5                 add     esp, 4
```

33. ábra – Assembly programlista

A hiba a rep movsd utasítás végrehajtása közben keletkezett. Ez az utasítás %ecx darab duplaszót másol %esi-ből %edi-be. A regiszterek tartalma a hiba bekövetkeztének pillanatában a következő volt:

```
eax=00000000 ebx=e30fd138 ecx=3ffffcd7 edx=ffffffff esi=00000ca0 edi=e3103000
eip=bf84ee98 esp=aaa149f8 ebp=aaa14a04 iopl=0         nv up ei pl nz na pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010207
```

34. ábra – Regiszterek tartalma a kivétel pillanatában

Az másolást megelőző utasításban látjuk, hogy a forrást jelölő %esi az %eax-al lesz egyenlő, ami (ahogy a regiszterek állapotából látható) éppen nulla. Ezzel nem is lesz probléma, hiszen mivel kernel módban vagyunk, tudunk a nullás virtuális memória címről olvasni. Az assembly listán azt is láthatjuk, hogy a másolás célját

[8] A Microsoft közzéteszi a legfőbb rendszerkomponensekhez tartozó szimbólum (debug) adatbázisokat, így volt lehetőségem a pontos függvénynév megismerésére.

meghatározó %edi, a függvényünk első paraméterében átadott címmel lesz egyenlő (BF84EE8C címen lévő utasítás). Onnan tudjuk, hogy az első átadott paraméter van az %ebp+8 címen, mivel %ebp címen a mentett keretmutató, az %ebp+4 címen pedig a visszatérési cím foglal helyet. Ez a cím jelen esetben e3102360, amit a stack „backtrace”-en is láthatunk (bekeretezve).

```
ChildEBP RetAddr  Args to Child
aaa14a04 bf8b000c e3102360 00002000 e32fc018 win32k!sfac_CopyFontAndPrePrograms+0x6b
aaa14a4c bf8ae59f e3100bf0 aaa14a98 bf8ae058 win32k!fs__NewTransformation+0x147
aaa14a58 bf8ae058 e30fd010 e30fd074 00000001 win32k!fs_NewTransformation+0x12
aaa14a98 bf8a4539 e32fc018 00000000 00000000 win32k!bSetXform+0x3c0
aaa14abc bf8a488e e32fc018 00000000 00000000 win32k!bGrabXform+0x6d
aaa14ae0 bf8a46db e1528d08 00000003 ffffffff win32k!ttfdQueryFontData+0xd2
aaa14b2c bf8a1664 e2231018 e1528d08 00000003 win32k!ttfdSemQueryFontData+0x45
aaa14b5c bf8b077a e2231018 e1528d08 00000003 win32k!PDEVOBJ::QueryFontData+0x3c
aaa14b90 bf8b1ab8 e221d008 00000000 aaa14d1c win32k!RFONTOBJ::bGetDEVICEMETRICS+0x49
aaa14c38 bf8074d4 aaa14d1c aaa14cc0 e150d6a8 win32k!RFONTOBJ::bRealizeFont+0x1eb
aaa14cc8 bf80753c e150d588 00000000 00000002 win32k!RFONTOBJ::bInit+0x29a
aaa14ce0 bf83aeb8 aaa14d1c 00000000 00000002 win32k!RFONTOBJ::vInit+0x16
aaa14d40 804dd99f 4821108a 00000004 02e919c8 win32k!NtGdiGetWidthTable+0xbd
```

35. ábra – Stack állapota

Ha megnézzük a regiszterek állapotát, a %esi és %edi értéke már CA0-val nagyobb, mint a kezdeti érékeik, vagyis ennyi bájtot már átmásoltunk, a kivétel keletkezése előtt. Az %ecx regiszterben tarjuk nyilván, hogy hány darab duplaszót kell még másolni (tehát a bájtban megadott hossz negyedét) így az már CA0/4-el kisebb a kezdeti értékénél. A BF84EE93 címen lévő utasításban, láthatjuk, hogy a másolás megkezdése előtt %ecx-et néggyel osztjuk, azaz ez a regiszter eredetileg a bájtban megadott hosszt tárolta. Ez az érék, az összes kivételt okozó teszteset esetén FFFFFFFF volt, ami a -1 kettes komplemens számábrázolásban. Tipikus eredete az előjelezési hibának, hogy például egy hosszt számító függvény, ha hibakóddal tér vissza (-1), akkor az nincs ellenőrizve, ezért a hibakód éréke lesz a hossz éréke, amit később előjel nélküliként értelmezünk. Jelen esetben is ez történt. Azt kaptuk tehát, hogy a rendszermag térben futó utasításunk 4GB-nyi adatott szeretne átmásolni a nullás memória címről az E3102360 címre.

Láthatjuk tehát, hogy több hiba fennállása okozta a jelenséget. Mind a forráscím, mind a másolandó hossz is egy hibát jelző visszatérési érték elfogadásával keletkezett. Ha a támadó el tudja érni, hogy a hossz értelmes értéket kapjon, valamint befolyásolni tudja a cél címet (átadott paraméter), akkor a hibát

tetszőleges kód futtatására, ki tudja használni. Mivel a hiba kernel szinten jelentkezett, a „tetszőleges kód” itt valóban „bármit” jelent.

9 ÉRTÉKELÉS ÉS KONKLÚZIÓ

A leggyakrabban előforduló tipikus biztonsági hibák bemutátása után, rendszerezve és csoportosítva azokat, tanulmányoztam a különböző védekezési módszereket majd megállapítottam, hogy a szoftverbiztonság tesztelésére napjainkban óriási szükség van. Rávilágítottam, hogy a hagyományos szoftverteszteléssel szemben a biztonsági tesztelés egy veszélyközpontú szemléletmódot igényel. Az ilyenfajta tesztelésnek a statikus kódelemzésen kívül a leggyakrabban alkalmzott módja a dinamikus „fuzz”-tesztelés.

A SEARCH Laboratórium keretein belül egy „inteligensebb fuzzing”-ra képes keretrendszert hoztunk létre, amellyel könnyebben és hatékonyabban lehet bizontsági tesztelést kivtelezni, mint a ma létező általános megoldásokkal. A rendszert médialejátszó alkamazások tesztelésén keresztül próbáltam ki a gyakorlatban. Rengeteg Denial of Service szerű támadásra lehetőséget adó hibát sikerült találnom. Ezen kívül sikerült újra felfedeznem egy már ismert és rendkívül nagy veszélyt jelentő sebezhetőséget, valamint egy ismeretlen és komoly hibát a Windows XP operációs rendszer kernelében. Ezzel alátámasztottam a FLINDER tesztelő keretrendszer használhatóságát.

10 TERVEK A JÖVŐRE NÉZVE

A bemutatott TrueType font hiba kihasználási módjait még nem volt időm feltérképezni, ezért terveim között elsőként szerepel a kihasználhatóság bizonyítása, valamint az eredmény jelentése.

Természetesen célom a keretrendszer továbbfejlesztése, tökéletesítése is. A kisebb fejlesztéseken kívül (mint pl. központi naplózás megvalósítása, GUI fejlesztése stb.) hosszú távú célként a detektálható hibák körének bővítését, valamint a felismerés pontosságának növelését szeretném megoldani, az ehez szükséges tesztesetek számának csökkentése mellett. Ezt a rendszer viselkedésének alaposabb futásidejű megfigyelésén keresztül valósítanám meg és a megfigyelés eredményeinek visszacsatolásával a tesztvektor generálás folyamatába. Erre egy reményteli megközelítés például a bemenő adatok nyomon követése és hatásaik elemzése. Erre eszközül szolgálhat a Valgrind bináris instrumentálást lehetővé tevő keretrendszere.

11 FÜGGELÉK

AZ ASF MÉDIA KONTÉNER FORMÁTUM LEÍRÁSA MFDL NYELVEN

```
<mfdl>
  <!--2.  Top-level file structure-->
  <ASF_File encoding="Binary">
    <preSerializeAction>
      importGlobal("asfHelper")
    </preSerializeAction>
    <ASF_Header_Object type="ASF_Header_Object_t"/>
    <ASF_Data_Object type="ASF_Data_Object_t"/>
  </ASF_File>
  <!--2.  Top-level file structure-->
  <!--2.1 ASF object definition-->
  <ASF_Object_t>
    <postSerializeAction>
      asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)
    </postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian">
      <meta Manipulator="IofOverflow" step="4"/>
    </Object_Size_Lower>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Data type="binContainer" msdl:bytesToLoad="0">
      <preParseAction>
        size = "%s" % (top.parent.Object_Size_Lower.value-24)
        top.bytesToLoad = size
      </preParseAction>
    </Data>
  </ASF_Object_t>
  <!--2.1 ASF object definition-->
  <!--3.  ASF top-level Header Object-->
  <ASF_Header_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Number_Of_Header_Objects type="uint32" byteOrder="littleEndian"/>
    <Alignment type="uint8" byteOrder="littleEndian"/>
    <Architecture type="uint8" byteOrder="littleEndian"/>
    <Header_Subobjects type="all" msdl:childBufferSize="0">
      <preParseAction>
        size = "%s" % (top.parent.Object_Size_Lower.value-30)
        top.childBufferSize = size
      </preParseAction>
      <choice>
        <preParseAction>
          choice.value = 'ASF_Object'
          if parseBuffer[0:16] == '\xA1\xDC\xAB\x8C\x47\xA9\xCF\x11\x8E\xE4\x00\xC0\x0C\x20\x53\x65':
            choice.value = 'ASF_File_Properties_Object'
          if parseBuffer[0:16] == '\x91\x07\xDC\xB7\xB7\xA9\xCF\x11\x8E\xE6\x00\xC0\x0C\x20\x53\x65':
            choice.value = 'ASF_Stream_Properties_Object'
          if parseBuffer[0:16] == '\xB5\x03\xBF\x5F\x2E\xA9\xCF\x11\x8E\xE3\x00\xC0\x0C\x20\x53\x65':
            choice.value = 'ASF_Header_Extension_Object'
          if parseBuffer[0:16] == '\x40\x52\xD1\x86\x1D\x31\xD0\x11\xA3\xA4\x00\xA0\xC9\x03\x48\xF6':
            choice.value = 'ASF_Codec_List_Object'
          if parseBuffer[0:16] == '\x30\x1A\xFB\x1E\x62\x0B\xD0\x11\xA3\x9B\x00\xA0\xC9\x03\x48\xF6':
            choice.value = 'ASF_Script_Command_Object'
          if parseBuffer[0:16] == '\x33\x26\xB2\x75\x8E\x66\xCF\x11\xA6\xD9\x00\xAA\x00\x62\xCE\x6C':
            choice.value = 'ASF_Content_Description_Object'
          if parseBuffer[0:16] == '\x40\xA4\xD0\xD2\x07\xE3\xD2\x11\x97\xF0\x00\xA0\xC9\x5E\xA8\x50':
            choice.value = 'ASF_Extended_Content_Description_Object'
          if parseBuffer[0:16] == '\xCE\x75\xF8\x7B\x8D\x46\xD1\x11\x8D\x82\x00\x60\x97\xC9\xA2\xB2':
            choice.value = 'ASF_Stream_Bitrate_Properties_Object'
          if parseBuffer[0:16] == '\xFB\xB3\x11\x22\x23\xBD\xD2\x11\xB4\xB7\x00\xA0\xC9\x55\xFC\x6E':
            choice.value = 'ASF_Content_Encryption_Object'
          if parseBuffer[0:16] == '\x14\xE6\x8A\x29\x22\x26\x17\x4C\xB9\x35\xDA\xE0\x7E\xE9\x28\x9C':
            choice.value = 'ASF_Extended_Content_Encryption_Object'
          if parseBuffer[0:16] == '\xFC\xB3\x11\x22\x23\xBD\xD2\x11\xB4\xB7\x00\xA0\xC9\x55\xFC\x6E':
            choice.value = 'ASF_Digital_Signature_Object'
          if parseBuffer[0:16] == '\x74\xD4\x06\x18\xDF\xCA\x09\x45\xA4\xBA\x9A\xAB\xCB\x96\xAA\xE8':
            choice.value = 'ASF_Padding_Object'
        </preParseAction>
        <ASF_File_Properties_Object type="ASF_File_Properties_Object_t"/>
        <ASF_Stream_Properties_Object type="ASF_Stream_Properties_Object_t"/>
        <ASF_Header_Extension_Object type="ASF_Header_Extension_Object_t"/>
        <ASF_Codec_List_Object type="ASF_Codec_List_Object_t"/>
```

```
      <ASF_Script_Command_Object type="ASF_Script_Command_Object_t"/>
      <ASF_Marker_Object type="ASF_Marker_Object_t"/>
      <ASF_Bitrate_Mutual_Exclusion_Object type="ASF_Bitrate_Mutual_Exclusion_Object_t"/>
      <ASF_Error_Correction_Object type="ASF_Error_Correction_Object_t"/>
      <ASF_Content_Description_Object type="ASF_Content_Description_Object_t"/>
      <ASF_Extended_Content_Description_Object type="ASF_Extended_Content_Description_Object_t"/>
      <ASF_Content_Branding_Object type="ASF_Content_Branding_Object_t"/>
      <ASF_Stream_Bitrate_Properties_Object type="ASF_Stream_Bitrate_Properties_Object_t"/>
      <ASF_Content_Encryption_Object type="ASF_Content_Encryption_Object_t"/>
      <ASF_Extended_Content_Encryption_Object type="ASF_Extended_Content_Encryption_Object_t"/>
      <ASF_Digital_Signature_Object type="ASF_Digital_Signature_Object_t"/>
      <ASF_Padding_Object type="ASF_Padding_Object_t"/>
      <ASF_Object type="ASF_Object_t"/>
    </choice>
  </Header_Subobjects>
</ASF_Header_Object_t>
<!--3.  ASF top-level Header Object-->
<!--3.2 File Properties Object (mandatory, one only)-->
<ASF_File_Properties_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <File_ID type="binContainer" bytesToLoad="16"/>
  <File_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <File_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Creation_Date_Lower type="uint32" byteOrder="littleEndian"/>
  <Creation_Date_Upper type="uint32" byteOrder="littleEndian"/>
  <Data_Packets_Count_Lower type="uint32" byteOrder="littleEndian"/>
  <Data_Packets_Count_Upper type="uint32" byteOrder="littleEndian"/>
  <Play_Duration_Lower type="uint32" byteOrder="littleEndian"/>
  <Play_Duration_Upper type="uint32" byteOrder="littleEndian"/>
  <Send_Duration_Lower type="uint32" byteOrder="littleEndian"/>
  <Send_Duration_Upper type="uint32" byteOrder="littleEndian"/>
  <Preroll_Lower type="uint32" byteOrder="littleEndian"/>
  <Preroll_Upper type="uint32" byteOrder="littleEndian"/>
  <Flags type="uint32" byteOrder="littleEndian"/>
  <Minimum_Data_Packet_Size type="uint32" byteOrder="littleEndian"/>
  <Maximum_Data_Packet_Size type="uint32" byteOrder="littleEndian"/>
  <Maximum_Bitrate type="uint32" byteOrder="littleEndian"/>
</ASF_File_Properties_Object_t>
<!--3.2 File Properties Object (mandatory, one only)-->
<!--3.3 Stream Properties Object (mandatory, one per stream)-->
<ASF_Stream_Properties_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Stream_Type type="binContainer" bytesToLoad="16"/>
  <Error_Correction_Type type="binContainer" bytesToLoad="16"/>
  <Time_Offset_Lower type="uint32" byteOrder="littleEndian"/>
  <Time_Offset_Upper type="uint32" byteOrder="littleEndian"/>
  <Type_Specific_Data_Length type="uint32" byteOrder="littleEndian">
    <preSerializeAction>top.value=len(top.parent.Type_Specific_Data.value)</preSerializeAction>
  </Type_Specific_Data_Length>
  <Error_Correction_Data_Length type="uint32" byteOrder="littleEndian">
    <preSerializeAction>top.value=len(top.parent.Error_Correction_Data.value)</preSerializeAction>
  </Error_Correction_Data_Length>
  <Flags type="uint16" byteOrder="littleEndian"/>
  <Reserved type="uint32" byteOrder="littleEndian"/>
  <Type_Specific_Data type="binContainer" msdl:bytesToLoad="0">
    <meta Manipulator="binContainer"/>
    <preParseAction>
      size = "%s" % (top.parent.Type_Specific_Data_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Type_Specific_Data>
  <Error_Correction_Data type="binContainer" msdl:bytesToLoad="0">
    <meta Manipulator="binContainer"/>
    <preParseAction>
      size = "%s" % (top.parent.Error_Correction_Data_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Error_Correction_Data>
</ASF_Stream_Properties_Object_t>
<!--3.3 Stream Properties Object (mandatory, one per stream)-->
<!--3.4 Header Extension Object (mandatory, one only)-->
<ASF_Header_Extension_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Clock_Type type="binContainer" bytesToLoad="16"/>
  <Clock_Size type="uint16" byteOrder="littleEndian"/>
```

```
  <Header_Extension_Data_Size type="uint32" byteOrder="littleEndian"/>
  <Header_Extension_Subobjects type="all" msdl:childBufferSize="0">
    <preParseAction>
      size = "%s" % (top.parent.Header_Extension_Data_Size.value)
      top.childBufferSize = size
    </preParseAction>
    <choice>
      <preParseAction>
        choice.value = 'ASF_Object'
        if parseBuffer[0:16] == '\xCB\xA5\xE6\x14\x72\xC6\x32\x43\x83\x99\xA9\x69\x52\x06\x5B\x5A':
          choice.value = 'ASF_Extended_Stream_Properties_Object'
        if parseBuffer[0:16] == '\xEA\xCB\xF8\xC5\xAF\x5B\x77\x48\x84\x67\xAA\x8C\x44\xFA\x4C\xCA':
          choice.value = 'ASF_Metadata_Object'
        if parseBuffer[0:16] == '\x5D\x8B\xF1\x26\x84\x45\xEC\x47\x9F\x5F\x0E\x65\x1F\x04\x52\xC9':
          choice.value = 'ASF_Compatibility_Object'
        if parseBuffer[0:16] == '\x74\xD4\x06\x18\xDF\xCA\x09\x45\xA4\xBA\x9A\xAB\xCB\x96\xAA\xE8':
          choice.value = 'ASF_Padding_Object'
      </preParseAction>
      <ASF_Extended_Stream_Properties_Object type="ASF_Extended_Stream_Properties_Object_t"/>
      <ASF_Advanced_Mutual_Exclusion_Object type="ASF_Advanced_Mutual_Exclusion_Object_t"/>
      <ASF_Group_Mutual_Exclusion_Object type="ASF_Group_Mutual_Exclusion_Object_t"/>
      <ASF_Stream_Prioritization_Object type="ASF_Stream_Prioritization_Object_t"/>
      <ASF_Bandwidth_Sharing_Object type="ASF_Bandwidth_Sharing_Object_t"/>
      <ASF_Language_List_Object type="ASF_Language_List_Object_t"/>
      <ASF_Metadata_Object type="ASF_Metadata_Object_t"/>
      <ASF_Metadata_Library_Object type="ASF_Metadata_Library_Object_t"/>
      <ASF_Index_Parameters_Object type="ASF_Index_Parameters_Object_t"/>
      <ASF_Media_Object_Index_Parameters_Object type="ASF_Media_Object_Index_Parameters_Object_t"/>
      <ASF_Timecode_Index_Parameters_Object type="ASF_Timecode_Index_Parameters_Object_t"/>
      <ASF_Compatibility_Object type="ASF_Compatibility_Object_t"/>
      <ASF_Advanced_Content_Encryption_Object type="ASF_Advanced_Content_Encryption_Object_t"/>
      <ASF_Padding_Object type="ASF_Padding_Object_t"/>
      <ASF_Object type="ASF_Object_t"/>
    </choice>
  </Header_Extension_Subobjects>
</ASF_Header_Extension_Object_t>
<!--3.4 Header Extension Object (mandatory, one only)-->
<!--3.5 Codec List Object (optional, one only)-->
<ASF_Codec_List_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Codec_ID type="binContainer" bytesToLoad="16"/>
  <Codec_Entries_Count type="uint32" byteOrder="littleEndian"/>
  <Codec_Entries type="all" msdl:childBufferSize="0">
    <preParseAction>
      size = "%s" % (top.parent.Object_Size_Lower.value-44)
      top.childBufferSize = size
    </preParseAction>
    <Codec_Entry type="Codec_Entry_t"/>
  </Codec_Entries>
</ASF_Codec_List_Object_t>
<Codec_Entry_t>
  <Type type="uint16" byteOrder="littleEndian"/>
  <Codec_Name_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Codec_Name.value)/2</preSerializeAction>
  </Codec_Name_Length>
  <Codec_Name type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Codec_Name_Length.value*2)
      top.bytesToLoad = size
    </preParseAction>
  </Codec_Name>
  <Codec_Description_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Codec_Description.value)/2</preSerializeAction>
  </Codec_Description_Length>
  <Codec_Description type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Codec_Description_Length.value*2)
      top.bytesToLoad = size
    </preParseAction>
  </Codec_Description>
  <Codec_Information_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Codec_Information.value)</preSerializeAction>
  </Codec_Information_Length>
  <Codec_Information type="binContainer" msdl:bytesToLoad="0">
    <meta Manipulator="binContainer"/>
    <preParseAction>
      size = "%s" % (top.parent.Codec_Information_Length.value)
      top.bytesToLoad = size
    </preParseAction>
```

```
  </Codec_Information>
</Codec_Entry_t>
<!--3.5 Codec List Object (optional, one only)-->
<!--3.6 Script Command Object (optional, one only)-->
<ASF_Script_Command_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <ComandID type="binContainer" bytesToLoad="16"/>
  <Commands_Count type="uint16" byteOrder="littleEndian"/>
  <Command_Types_Count type="uint16" byteOrder="littleEndian"/>
  <CommandTypes msdl:array="0">
    <preParseAction>
      count = "%s" % (top.parent.Command_Types_Count.value)
      top.array = count
    </preParseAction>
    <CommandType type="Command_Type_t"/>
  </CommandTypes>
  <Commands msdl:array="0">
    <preParseAction>
      count = "%s" % (top.parent.Commands_Count.value)
      top.array = count
    </preParseAction>
    <Command type="Command_t"/>
  </Commands>
</ASF_Script_Command_Object_t>
<Command_Type_t>
  <Command_Type_Name_Length type="uint16">
    <preSerializeAction>top.value = len(top.parent.Command_Type_Name.value)/2</preSerializeAction>
  </Command_Type_Name_Length>
  <Command_Type_Name type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Command_Type_Name_Length.value*2)
      top.bytesToLoad = size
    </preParseAction>
  </Command_Type_Name>
</Command_Type_t>
<Command_t>
  <Presentation_Time type="uint32" byteOrder="littleEndian"/>
  <Type_Index type="uint16" byteOrder="littleEndian"/>
  <Command_Name_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Command_Name.value)/2</preSerializeAction>
  </Command_Name_Length>
  <Command_Name type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Command_Name_Length.value*2)
      top.bytesToLoad = size
    </preParseAction>
  </Command_Name>
</Command_t>
<!--3.6 Script Command Object (optional, one only)-->
<!--3.10  Content Description Object (optional, one only)-->
<ASF_Content_Description_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Title_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Title.value)</preSerializeAction>
  </Title_Length>
  <Author_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Author.value)</preSerializeAction>
  </Author_Length>
  <Copyright_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Copyright.value)</preSerializeAction>
  </Copyright_Length>
  <Description_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Description.value)</preSerializeAction>
  </Description_Length>
  <Rating_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Rating.value)</preSerializeAction>
  </Rating_Length>
  <Title type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Title_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Title>
  <Author type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
```

```
      <preParseAction>
        size = "%s" % (top.parent.Author_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Author>
    <Copyright type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Copyright_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Copyright>
    <Description type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Description_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Description>
    <Rating type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Rating_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Rating>
  </ASF_Content_Description_Object_t>
  <!--3.10  Content Description Object (optional, one only)-->
  <!--3.11  Extended Content Description Object (optional, one only)-->
  <ASF_Extended_Content_Description_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Content_Descriptors_Count type="uint16" byteOrder="littleEndian"/>
    <Content_Descriptors type="all" msdl:childBufferSize="0">
      <preParseAction>
        size = "%s" % (top.parent.Object_Size_Lower.value-26)
        top.childBufferSize = size
      </preParseAction>
      <Content_Descriptor type="Content_Descriptor_t"/>
    </Content_Descriptors>
  </ASF_Extended_Content_Description_Object_t>
  <Content_Descriptor_t>
    <Descriptor_Name_Length type="uint16" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Descriptor_Name.value)</preSerializeAction>
    </Descriptor_Name_Length>
    <Descriptor_Name type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Descriptor_Name_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Descriptor_Name>
    <Descriptor_Value_Data_Type type="uint16" byteOrder="littleEndian"/>
    <Data_Length type="uint16" byteOrder="littleEndian"/>
    <choice>
      <preParseAction>
        type = top.Descriptor_Value_Data_Type.value
        choice.value = 'BYTE_Array'
        if type == 0 : choice.value = 'Unicode_String'
        if type == 1 : choice.value = 'BYTE_Array'
        if type == 2 : choice.value = 'BOOL_4_Byte'
        if type == 3 : choice.value = 'DWORD'
        if type == 4 : choice.value = 'QWORD'
        if type == 5 : choice.value = 'WORD'
      </preParseAction>
      <Unicode_String type="Unicode_String_t"/>
      <BYTE_Array type="BYTE_Array_t"/>
      <BOOL_4_Byte type="BOOL_4_Byte_t"/>
      <DWORD type="DWORD_t"/>
      <QWORD type="QWORD_t"/>
      <WORD type="WORD_t"/>
    </choice>
  </Content_Descriptor_t>
  <!--3.11  Extended Content Description Object (optional, one only)-->
  <!--3.12  Stream Bitrate Properties Object (optional but recommended, one only)-->
  <ASF_Stream_Bitrate_Properties_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Bitrate_Records_Count type="uint16" byteOrder="littleEndian"/>
    <Bitrate_Records type="all" msdl:childBufferSize="0">
```

```
      <preParseAction>
        size = "%s" % (top.parent.Object_Size_Lower.value-26)
        top.childBufferSize = size
      </preParseAction>
      <Bitrate_Record type="Bitrate_Record_t"/>
    </Bitrate_Records>
  </ASF_Stream_Bitrate_Properties_Object_t>
  <Bitrate_Record_t>
    <Flags type="uint16" byteOrder="littleEndian"/>
    <Average_Bitrate type="uint32" byteOrder="littleEndian"/>
  </Bitrate_Record_t>
  <!--3.12  Stream Bitrate Properties Object (optional but recommended, one only)-->
  <!--3.14  Content Encryption Object (optional, 0 or 1)-->
  <ASF_Content_Encryption_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Secret_Data_Size type="uint32" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Secret_Data.value)</preSerializeAction>
    </Secret_Data_Size>
    <Secret_Data type="binContainer" msdl:bytesToLoad="0">
      <meta Manipulator="binContainer"/>
      <preParseAction>
        size = "%s" % (top.parent.Secret_Data_Size.value)
        top.bytesToLoad = size
      </preParseAction>
    </Secret_Data>
    <Protection_Type_Size type="uint32" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Protection_Type.value)</preSerializeAction>
    </Protection_Type_Size>
    <Protection_Type type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Protection_Type_Size.value)
        top.bytesToLoad = size
      </preParseAction>
    </Protection_Type>
    <Key_ID_Size type="uint32" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Key_ID.value)</preSerializeAction>
    </Key_ID_Size>
    <Key_ID type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Key_ID_Size.value)
        top.bytesToLoad = size
      </preParseAction>
    </Key_ID>
    <License_URL_Size type="uint32" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.License_URL.value)</preSerializeAction>
    </License_URL_Size>
    <License_URL type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.License_URL_Size.value)
        top.bytesToLoad = size
      </preParseAction>
    </License_URL>
  </ASF_Content_Encryption_Object_t>
  <!--3.14  Content Encryption Object (optional, 0 or 1)-->
  <!--3.15  Extended Content Encryption Object (optional, 0 or 1)-->
  <ASF_Extended_Content_Encryption_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
    <Encryption_Data_Size type="uint32" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Encryption_Data.value)</preSerializeAction>
    </Encryption_Data_Size>
    <Encryption_Data type="string" msdl:bytesToLoad="0">
      <meta Manipulator="string"/>
      <preParseAction>
        size = "%s" % (top.parent.Encryption_Data_Size.value)
        top.bytesToLoad = size
      </preParseAction>
    </Encryption_Data>
  </ASF_Extended_Content_Encryption_Object_t>
  <!--3.15  Extended Content Encryption Object (optional, 0 or 1)-->
  <!--3.16  Digital Signature Object (optional, 0 or 1)-->
  <ASF_Digital_Signature_Object_t>
    <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
    <Object_ID type="binContainer" bytesToLoad="16"/>
    <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
    <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
```

```
  <Signature_Type type="uint32" byteOrder="littleEndian"/>
  <Signature_Data_Length type="uint32" byteOrder="littleEndian">
    <preSerializeAction>top.value = len(top.parent.Signature_Data.value)</preSerializeAction>
  </Signature_Data_Length>
  <Signature_Data type="binContainer" msdl:bytesToLoad="0">
    <meta Manipulator="binContainer"/>
    <preParseAction>
      size = "%s" % (top.parent.Signature_Data_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Signature_Data>
</ASF_Digital_Signature_Object_t>
<!--3.16  Digital Signature Object (optional, 0 or 1)-->
<!--3.17  Padding Object (optional, 0 to many)-->
<ASF_Padding_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Padding_Data type="binContainer" msdl:bytesToLoad="0">
    <preParseAction>
      size = "%s" % (top.parent.Object_Size_Lower.value-24)
      top.bytesToLoad = size
    </preParseAction>
  </Padding_Data>
</ASF_Padding_Object_t>
<!--3.17  Padding Object (optional, 0 to many)-->
<!--4.1 Extended Stream Properties Object (optional, 1 per media stream)-->
<ASF_Extended_Stream_Properties_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Start_Time_Lower type="uint32" byteOrder="littleEndian"/>
  <Start_Time_Upper type="uint32" byteOrder="littleEndian"/>
  <End_Time_Lower type="uint32" byteOrder="littleEndian"/>
  <End_Time_Upper type="uint32" byteOrder="littleEndian"/>
  <Data_Bitrate type="uint32" byteOrder="littleEndian"/>
  <Buffer_Size type="uint32" byteOrder="littleEndian"/>
  <Initial_Buffer_Fullness type="uint32" byteOrder="littleEndian"/>
  <Alternate_Data_Bitrate type="uint32" byteOrder="littleEndian"/>
  <Alternate_Buffer_Size type="uint32" byteOrder="littleEndian"/>
  <Alternate_Initial_Buffer_Fullness type="uint32" byteOrder="littleEndian"/>
  <Maximum_Object_Size type="uint32" byteOrder="littleEndian"/>
  <Flags type="uint32" byteOrder="littleEndian"/>
  <Stream_Number type="uint16" byteOrder="littleEndian"/>
  <Stream_Language_ID_Index type="uint16" byteOrder="littleEndian"/>
  <Average_Time_Per_Frame_Lower type="uint32" byteOrder="littleEndian"/>
  <Average_Time_Per_Frame_Upper type="uint32" byteOrder="littleEndian"/>
  <Stream_Name_Count type="uint16" byteOrder="littleEndian"/>
  <Payload_Extension_System_Count type="uint16" byteOrder="littleEndian"/>
  <Stream_Names type="Stream_Names_t"/>
  <Payload_Extension_Systems type="Payload_Extension_Systems_t"/>
  <Stream_Properties_Object type="ASF_Stream_Properties_Object_t"/>
</ASF_Extended_Stream_Properties_Object_t>
<Stream_Names_t>
  <Language_ID_Index type="uint16" byteOrder="littleEndian"/>
  <Stream_Name_Length type="uint16" byteOrder="littleEndian"/>
  <Stream_Name type="string" msdl:bytesToLoad="0">
    <preParseAction>
      size = "%s" % (top.parent.Stream_Name_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Stream_Name>
</Stream_Names_t>
<Payload_Extension_Systems_t>
  <Extension_System_ID type="binContainer" bytesToLoad="16"/>
  <Extension_Data_Size type="uint16" byteOrder="littleEndian"/>
  <Extension_System_Info_Length type="uint32" byteOrder="littleEndian"/>
  <Extension_System_Info type="binContainer" msdl:bytesToLoad="0">
    <preParseAction>
      size = "%s" % (top.parent.Extension_System_Info_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Extension_System_Info>
</Payload_Extension_Systems_t>
<!--4.1 Extended Stream Properties Object (optional, 1 per media stream)-->
<!--4.7 Metadata Object (optional, 0 or 1)-->
<ASF_Metadata_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Description_Records_Count type="uint16" byteOrder="littleEndian"/>
```

```
  <Description_Records type="all" msdl:childBufferSize="0">
    <preParseAction>
      size = "%s" % (top.parent.Object_Size_Lower.value-26)
      top.childBufferSize = size
    </preParseAction>
    <Description_Record type="Description_Record_t"/>
  </Description_Records>
</ASF_Metadata_Object_t>
<Description_Record_t>
  <Reserved type="uint16" byteOrder="littleEndian"/>
  <Stream_Number type="uint16" byteOrder="littleEndian"/>
  <Name_Length type="uint16" byteOrder="littleEndian">
    <preSerializeAction>
      top.value = len(top.parent.Name.value)
    </preSerializeAction>
  </Name_Length>
  <Data_Type type="uint16" byteOrder="littleEndian"/>
  <Data_Length type="uint32" byteOrder="littleEndian"/>
  <Name type="string" msdl:bytesToLoad="0">
    <meta Manipulator="string"/>
    <preParseAction>
      size = "%s" % (top.parent.Name_Length.value)
      top.bytesToLoad = size
    </preParseAction>
  </Name>
  <choice>
    <preParseAction>
      type = top.Data_Type.value
      choice.value = 'BYTE_Array'
      if type == 0 : choice.value = 'Unicode_String'
      if type == 1 : choice.value = 'BYTE_Array'
      if type == 2 : choice.value = 'BOOL_2_Byte'
      if type == 3 : choice.value = 'DWORD'
      if type == 4 : choice.value = 'QWORD'
      if type == 5 : choice.value = 'WORD'
    </preParseAction>
    <Unicode_String type="Unicode_String_t"/>
    <BYTE_Array type="BYTE_Array_t"/>
    <BOOL_2_Byte type="BOOL_2_Byte_t"/>
    <DWORD type="DWORD_t"/>
    <QWORD type="QWORD_t"/>
    <WORD type="WORD_t"/>
  </choice>
</Description_Record_t>
<!--4.7 Metadata Object (optional, 0 or 1)-->
<!--4.12  Compatibility Object (optional, only 1)-->
<ASF_Compatibility_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <Profile type="uint8" byteOrder="littleEndian"/>
  <Mode type="uint8" byteOrder="littleEndian"/>
</ASF_Compatibility_Object_t>
<!--4.12  Compatibility Object (optional, only 1)-->
<!--5.  ASF top-level Data Object-->
<ASF_Data_Object_t>
  <postSerializeAction>asfHelper.setEncodingBuffer(top.Object_Size_Lower, top)</postSerializeAction>
  <Object_ID type="binContainer" bytesToLoad="16"/>
  <Object_Size_Lower type="uint32" byteOrder="littleEndian"/>
  <Object_Size_Upper type="uint32" byteOrder="littleEndian"/>
  <File_ID type="binContainer" bytesToLoad="16"/>
  <Total_Data_Packets_Lower type="uint32" byteOrder="littleEndian"/>
  <Total_Data_Packets_Upper type="uint32" byteOrder="littleEndian"/>
  <Reserved type="uint16" byteOrder="littleEndian"/>
  <Data_Packets type="all" msdl:childBufferSize="0">
    <preParseAction>
      size = "%s" % (top.parent.Object_Size_Lower.value-50)
      top.childBufferSize = size
    </preParseAction>
    <Data_Packet type="Data_Packet_t"/>
  </Data_Packets>
</ASF_Data_Object_t>
<!--5.  ASF top-level Data Object-->
<!--5.2 ASF data packet definition-->
<Data_Packet_t>
  <Error_Correction_Flags type="uint8" byteOrder="littleEndian"/>
  <Error_Correction_Data type="binContainer" bytesToLoad="2"/>
  <Length_Type_Flags type="uint8" byteOrder="littleEndian"/>
  <Property_Flags type="uint8" byteOrder="littleEndian"/>
  <Padding_Length type="uint8" byteOrder="littleEndian"/>
  <Send_Time type="uint32" byteOrder="littleEndian"/>
  <Duration type="uint16" byteOrder="littleEndian"/>
  <Stream_Number type="uint8" byteOrder="littleEndian"/>
```

```
    <Media_Object_Number type="uint8" byteOrder="littleEndian"/>
    <Offset_Into_Media_Object type="uint32" byteOrder="littleEndian"/>
    <Replicated_Data_Length type="uint8" byteOrder="littleEndian">
      <preSerializeAction>top.value = len(top.parent.Replicated_Data.value)</preSerializeAction>
    </Replicated_Data_Length>
    <Replicated_Data type="binContainer" msdl:bytesToLoad="0">
      <meta Manipulator="binContainer"/>
      <preParseAction>
        size = "%s" % (top.parent.Replicated_Data_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Replicated_Data>
    <Payload  type="binContainer" bytesToLoad="1487"/>
    <Padding_Data  type="binContainer" bytesToLoad="4">
    </Padding_Data>
  </Data_Packet_t>
  <!--5.2 ASF data packet definition-->
  <!--Common  Data Types-->
  <Unicode_String_t>
    <Value type="string" msdl:bytesToLoad="0">
      <preParseAction>
        size = "%s" % (top.parent.parent.Data_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Value>
  </Unicode_String_t>
  <BYTE_Array_t>
    <Value type="binContainer" msdl:bytesToLoad="0">
      <preParseAction>
        size = "%s" % (top.parent.parent.Data_Length.value)
        top.bytesToLoad = size
      </preParseAction>
    </Value>
  </BYTE_Array_t>
  <BOOL_2_Byte_t>
    <Value type="uint16" byteOrder="littleEndian"/>
  </BOOL_2_Byte_t>
  <BOOL_4_Byte_t>
    <Value type="uint32" byteOrder="littleEndian"/>
  </BOOL_4_Byte_t>
  <DWORD_t>
    <Value type="uint32" byteOrder="littleEndian"/>
  </DWORD_t>
  <QWORD_t>
    <Value_Lower type="uint32" byteOrder="littleEndian"/>
    <Value_Upper type="uint32" byteOrder="littleEndian"/>
  </QWORD_t>
  <WORD_t>
    <Value type="uint16" byteOrder="littleEndian"/>
  </WORD_t>
  <!--Common  Data Types-->
</mfdl>
```

12 IRODALOMJEGYZÉK

[1] R. Giobbi, *Avast! antivirus buffer overflow vulnerability*. US-CERT, 2007.

[2] National Vulnerability Database. Web page: http://nvd.nist.gov/.

[3] A. One, "Smashing The Stack For Fun And Profit", *Phrack*, vol. 7, November 1996.

[4] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns", *Security & Privacy Magazine, IEEE*, vol. 2, pp. 20-27, 2004.

[5] Matt Conover and w00w00 Security Team, "w00w00 on Heap Overflows", 1999.

[6] Anonymous, "Once upon a free()", *Phrack*, vol. 11, 2001.

[7] Blexim, "Basic Integer Overflows", *Phrack*, vol. 11, 2002.

[8] A. Thuemmel, "Analysis of format string bugs", *Manuscript*, 2001.

[9] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications", *Annual Symposium on Principles of Programming Languages*, pp. 372-382, 2006.

[10] C. Anley, "Advanced SQL Injection In SQL Server Applications", *White paper, Next Generation Security Software Ltd*, 2002.

[11] A. Klein, "Cross Site Scripting Explained", *Sanctum White Paper*, 2002.

[12] R.P. Abbott et al., "Security Analysis and Enhancements of Computer Operating Systems", 1976.

[13] R. II and D. Hollingworth, *Protection Analysis: Final Report*. Technical Report ISI/SR-78-13, University of Southern California/Information Sciences Institute, Marina Del Rey, CA, May 1978.(page93).

[14] C.E. Landwehr et al., "A taxonomy of computer program security flaws", *ACM Computing Surveys (CSUR)*, vol. 26, pp. 211-254, 1994.

[15] T. Aslam, "*A Taxonomy of security faults in the UNIX operating system",* Purdue University, 1995.

[16] F. Piessens, "A taxonomy of causes of software vulnerabilities in Internet software", *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, pp. 47-52, 2002.

[17] S. Weber, P.A. Karger, and A. Paradkar, "A software flaw taxonomy: aiming tools at security", *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-7, 2005.

[18] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors", *Security & Privacy Magazine, IEEE*, vol. 3, pp. 81-84, 2005.

[19] Preliminary List of Vulnerability Examples for Researchers. Web page: http://cve.mitre.org/docs/plover.

[20] Comprehensive, Lightweight Application Security Process. Web page: http://www.securesoftware.com.

[21] OASIS Application Vulnerability Description Language. Web page: http://www.oasis-open.org/committees/avdl.

[22] Open Web Application Security Project. Web page: http://www.owasp.org.

[23] M. Howard, D. LeBlanc, and J. Viega, *19 deadly sins of software security*. McGraw-Hill/Osborne, 2005.

[24] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Transactions on Computer Systems (TOCS)*, vol. 8, pp. 18-36, 1990.

[25] G. Lowe, "Casper: A compiler for the analysis of security protocols", *Journal of Computer Security*, vol. 6, pp. 53-84, 1998.

[26] T. Jim et al., "Cyclone: A safe dialect of C", *USENIX Annual Technical Conference*, pp. 275–288, 2002.

[27] G.C. Necula et al., "CCured: type-safe retrofitting of legacy software", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, pp. 477-526, 2005.

[28] The Better String Library. Web page: http://bstring.sourceforge.net.

[29] NX bit - Wikipedia, the free encyclopedia. Web page: http://en.wikipedia.org/wiki/NX_bit.

[30] Homepage of PaX. Web page: http://pax.grsecurity.net/.

[31] Limiting buffer overflows with ExecShield. Web page: http://www.redhat.com/magazine/009jul05/features/execshield/.

[32] W^X - Wikipedia, the free encyclopedia. Web page: http://en.wikipedia.org/wiki/W%5EX.

[33] B. McCarty, *Selinux*. O&Reilly, 2004.

[34] H. Etoh, "ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks", *IBM (April 2003),* Web page: http://www. trl. ibm. com/projects/security/ssp.

[35] C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks".

[36] D. Moore and C. Shannon, "Code-Red: a case study on the spread and victims of an internet worm", *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment*, pp. 273-284, 2002.

[37] M. Bauer, "Application Proxying with Zorp", *Linux Journal*, pp. 30-33, 2004.

[38] M. Roesch, "Snort-Lightweight Intrusion Detection for Networks", *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999.

[39] J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code", *Proceedings of the 16th Annual Computer Security Applications Conference*, pp. 257, 2000.

[40] A. Almossawi, K. Lim, and T. Sinha, *Analysis tool evaluation: Coverity prevent, May 2006*.

[41] B.P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities", *Communications of the ACM*, vol. 33, pp. 32-44, 1990.

[42] B.P. Miller, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. University of Wisconsin-Madison, Computer Sciences Dept, 1995.

[43] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing".

[44] B.P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing", *Proceedings of the 1st international workshop on Random testing*, pp. 46-54, 2006.

[45] P. Oehlert, "Violating assumptions with fuzzing", *Security & Privacy Magazine, IEEE*, vol. 3, pp. 58-62, 2005.

[46] R. Kaksonen, "A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis)", *Espoo. Technical Research Centre of Finland, VTT Publications*, vol. 447, pp. 951-38.

[47] K. Sollins, *The TFTP Protocol (Revision 2)*. RFC-783, MIT, June 1981.

[48] D. Aitel, "The Advantages of Block-Based Protocol Analysis for Security Testing", *Immunity Inc., February*, 2002.

[49] M. Eddington, *Peach fuzzer framework*. Web page: http://peachfuzz.sourceforge.net/.

[50] SEARCH-Lab, *Flinder*. Web page: http://www.flinder.hu/.

[51] W. Gora, *ASN. 1-Abstract Syntax Notation One*. Datacom, 1987.

[52] T. Bray, J. Paoli, and C.M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", *W3C Recommendation*, vol. 6, 2000.

[53] G. Van Rossum, "Python programming language", *CWI, Department CST, The Netherlands*, 1994.

[54] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision".

[55] L. Szekeres, *Biztonsági szempontból veszélyes programozói hibák felderítése automatizált módszerekkel*. Students' Scientific Conference (TDK), Budapest University of Technology and Economics, 2006.

[56] Microsoft Security Bulletin MS05-002, "Vulnerability in Cursor and Icon Format Handling Could Allow Remote Code Execution (891711)", 2005.

[57] P. Amini, "PyDbg - Python Win32 Debugger". Web page: http://pedram.redhive.com/PaiMei/docs/PyDbg/.

[58] Yuji Ukai, "Windows ANI File Parsing Buffer Overflow", 2005.