Szoftverbiztonság

SZEKERES LÁSZLÓ, TÓTH GERGELY

SEARCH-LAB Kft.
{laszlo.szekeres, gergely.toth}@search-lab.hu

Kulcsszavak: szoftverbiztonság, puffer-túlcsordulás, szoftvertesztelés, fuzzing, statikus elemzés, verifikáció

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

1. Bevezetés

A mai társadalmunk nagymértékben függ az informatikai rendszerektől, és ez a függőség rohamos mértékben növekszik. Ez újabb és újabb biztonsági követelmények elé is állítja rendszereinket, melyeknek a mai technológia sokszor nem tud megfelelni. A szoftverfejlesztés ma egyike a legnehezebb mérnöki feladatoknak. Elsősorban ennek köszönhető az, hogy a működésben lévő szoftverekben rengeteg a hiba, sokszor nem megfelelően működnek, valamint megbízhatatlanok, nem robusztusak. Sajnos ezek a hibák gyakran nem csupán a funkcionalitásban okoznak hiányosságokat, hanem biztonsági szempontból is: biztonsági lyukakat, sebezhetőségeket eredményeznek.

A cikkben áttekintést szeretnénk adni az olvasónak a mai szoftverbiztonság helyzetéről. A második szakaszban tisztázzuk a szoftverbiztonság szó jelentését. Ezután ismertetjük a probléma jelentőségét, kiterjedtségét és a benne rejlő kockázatokat. A negyedik szakaszban bemutatunk néhányat azokból a tipikus szoftverhibákból, melyek a problémák gyökereit képezik. Ezekután bemutatjuk a hagyományos védekezési módszereket a szoftver sebezhetőségei ellen, miközben rávilágítunk e módszerek hiányosságaira. Az ezt követő szakasz azokat a lehetőségeket veszi számba, melyek a fejlesztők rendelkezésére állnak, hogy elkerüljék vagy időben megtalálják a veszélyt okozó hibákat. Végül kitekintést teszünk a jövő felé és bemutatunk néhány, a területet érintő reményteli kutatási irányzatot.

2. Mi a szoftverbiztonság?

A szoftverbiztonság szó alatt az irodalom két jól elkülöníthető területet is ért. A két területet egymástól elválasztó kérdés az, hogy kit védünk, illetve kit tekintünk támadónak. Az egyik lehetőség, amikor magát a szoftvert, pontosabban annak fejlesztőjét védjük a szoftvere illegális másolásától, visszafejtésétől (reverse engineering) vagy rosszindulatú módosításától. Ebben a modellben magát a szoftvert tekintjük „ártatlannak", amely nem bízik az őt futtató, potenciálisan rosszindulatú hosztban. Ez a terület elsősorban a szellemi tulajdonjogok védelméről szól. A továbbiakban ezzel a területtel nem is foglakozunk. Azoknak, akik érdeklődnek a téma iránt Collberg áttekintő cikkét [1] ajánljuk.

A másik terület, amivel ez a cikk is foglalkozik, nem a szoftver védelmére, hanem az azt futtató hoszt illetve a felhasználó védelmére összpontosít. Vagyis a modell itt a szoftvert, a programkódot tarja megbízhatatlannak a hoszt vagy a felhasználó szempontjából. Milyen veszélyeknek van kitéve ebben a modellben a felhasználó?

Az egyik veszélytípust az úgy nevezett rosszindulatú kódok (malicious code) alkotják, melyek szándékosan valamilyen nem megengedett műveletet szeretnének a hoszton végrehajtani. Ilyenek a mindenki által jól ismert vírusok, férgek, trójai programok, kémszoftverek, logikai bombák és így tovább. A másik veszélyforrás, amely az imént említett rosszindulatú kódok túlnyomó többségének létezését is lehetővé teszi, az a számítógépen futtatott operációs rendszerben és alkalmazásokban lévő, általában nem szándékosan elkövetett hibákból adódó sebezhetőségek jelenléte. A továbbiakban szoftverbiztonság alatt ezt az utólag felvázolt területet értjük, vagyis a nem megbízható kód modellt tekintjük relevánsnak.

Továbbá nem összekeverendő a szoftverbiztonság fogalma azokkal a biztonsági szoftverekkel, amelyek éppen valamilyen biztonsági funkciót valósítanak meg (például rejtjelezés, naplózás, hozzáférés-védelem). Ide sorolhatók még például a tűzfal vagy vírusirtó programok. Hogy érzékeltessük a két terület közötti különbséget, jogosan tehetnénk fel a kérdést, hogy egyáltalán egy behatolásdetektáló rendszer, vagy egy ví-

rusirtó telepítése növeli-e, vagy – a benne potenciálisan megbújó biztonsági lyukak révén – éppen csökkenti egy rendszer biztonságát [2].

Tulajdonképpen mit is jelent az, hogy biztonságos egy szoftver? Ha egy mondatban szeretnénk megfogalmazni, akkor azt mondhatnánk, hogy az a szoftver biztonságos, ami azt teszi, amit elvárunk tőle, hogy tegyen, és – ami ugyanolyan fontos – semmi egyebet. A programfejlesztés mai technikái mellett ez sajnos nem tűnik megvalósíthatónak. A szoftverekben olyan hibák maradnak, amelyek sérülékennyé, sebezhetővé és támadhatóvá teszik az azt futtató rendszert.

3. Mekkora a probléma?

Nap mint nap olvashatunk híreket számítógépes betörésekről, a levélszemétről (spam), botnetekről, vírusokról és férgekről (worm). Ezen problémák mérhetetlen károkat, dollárbilliókban mérhető veszteségeket okoznak évente. Ha jobban a dolgok mögé nézünk, akkor kiderül, hogy az illetéktelen számítógépes behatolások valamilyen szoftversebezhetőség kihasználásán keresztül történnek meg. Az internetes férgek gyors terjedését szintén programozói hibák, illetve az azok által keltett szoftverbiztonsági sebezhetőségek teszik lehetővé.

A hibát kihasználva a férgek egy rejtett hátsó ajtó program (rootkit) telepítésével zombi gépekké változtatják az internetre csatlakozó számítógépeket, melyek botnet hálózatot alakítanak ki, amit pedig tömeges spamküldésre, valamint összehangolt szolgáltatásmegtagadásos támadásokra használnak. Az asztali operációs rendszerek többsége olyan kémprogramokkal fertőzött, amelyek bizalmas információkat küldenek annak felhasználójáról. Ezek a programok az esetek túlnyomó többségében úgy települnek fel, hogy a felhasználó meglátogat egy rosszindulatú weboldalt, amely a böngészőben vagy annak valamely pluginjében lévő szoftverhibát kihasználva tetszőleges kódot tud lefuttatni a számítógépen. Látható tehát, hogy az infokommunikációs rendszerek legnagyobb problémáit és kockázatait alapvetően a rossz minőségű szoftver okozza.

A fenyegetettség nagysága ráadásul folyamatosan növekszik. A növekvő összekötöttség, az Internetre csatlakozó eszközök és szolgáltatások egyre növekvő száma (gondoljunk csak a mobiltelefonokra vagy a népszerű webes szolgáltatásokra) a támadási lehetőségek számát is növeli. Ezen kívül a szoftverek komplexitása is növekszik. A Windows XP operációs rendszer 40, míg a Windows Server 2003 már 50 millió sornyi forráskódból állt. Minél több sor kód, annál több programozói hiba, és minél több programozói hiba, annál több potenciális biztonsági sebezhetőség kerül a végtermékekbe. Az *1. ábrán* statisztika látható a 2002 és 2006 között talált és publikált szoftveres sebezhetőségek számáról, a NIST [3] sebezhetőségi adatbázisa alapján.

Érdemes megfigyelni az exponenciálisan növekvő tendenciát a 2003-as évtől kezdődően. Ma, hogyha fény derül egy biztonsági hibára, akkor az mindaddig támadhatóvá teszi az érintett rendszereket, amíg azt a hibát ki nem javítják, be nem foltozzák. Ha figyelembe vesszük ezt az időrést és a fenti statisztikát a talált hibák számát illetően, a mai Internetre csatlakozó rendszereinkről nyugodtan kijelenthetjük, hogy állandó veszélynek vannak kitéve.

4. Néhány tipikus hiba és sebezhetőség

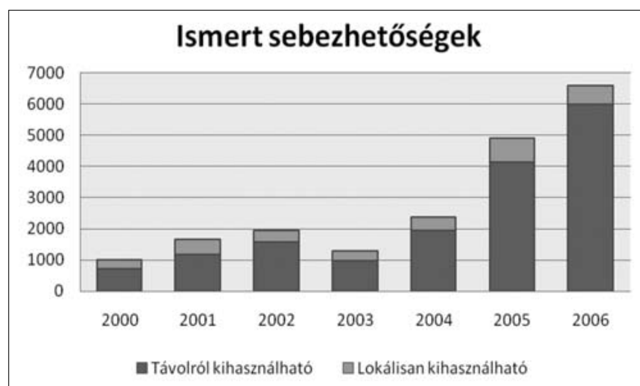
Biztonsági hiba nagyon sok helyen kerülhet a rendszerbe: már rögtön a követelmények meghatározásánál, vagy a rendszer tervezésénél, az implementálás során, de akár még a használat, illetve működtetés közben is okozhat egy nem megfelelő konfiguráció vagy környezet biztonsági hiányosságokat. Ebben a szakaszban bemutatásra kerül néhány, a legnagyobb számban előforduló biztonsági szempontból veszélyes, az implementáció során elkövetett programozási hiba.

Puffertúlcsordulás – az öreg hiba

A biztonsági szempontból veszélyes programozási hibák közül a legrégebb óta jelenlévők, nagyon sűrűn elkövetettek és legveszélyesebbek azok, amelyek puffertúlcsorduláshoz vezethetnek. A puffertúlcsordulás akkor történhet meg, amikor a szoftver egy fix hosszúságú tömböt (puffert) lefoglal a memóriában és a tömb írásakor nem ellenőrzi annak határait. Ilyenkor a támadónak lehetősége nyílik arra, hogy egy lefoglalt tömböt túlírva (tipikusan valamilyen túlzottan hosszú bemenet segítségével) felülírjon a program működése szempontjából fontos adatokat a memóriában. Ezek a hibák elsősorban a C/C++ programozási nyelv sajátosságai miatt fordulnak elő.

A legnagyobb veszély akkor jelentkezik, ha a szóban forgó fix hosszúságú tömböt lokális változóként definiálják, ugyanis ilyenkor a tömb a vermen (stack) tárolódik, amiből következően a tömb határán túlírva lehetőség nyílik a függvény visszatérési címének felülírására és ezáltal az eredeti futási útvonal eltérítésére. Vegyük szemügyre például az *2. ábrán* látható hibásan megírt programot.

1. ábra
Talált sebezhetőségek száma 2000 és 2006 között



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

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

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

2. ábra Hibás C forráskód

A program az első argumentumaként kapott karaktersorozat kezdőcímét átadja a `bad_func` nevű függvénynek. Ezután a `strcpy()` könyvtári függvény a megadott címen lévő karakterláncot a lokálisan deklarált `buffer` nevű tömbbe másolja. A függvény meghívásakor (az x86 architektúrát feltételezve) a verem állapota a *3. ábrán* látható.

Amikor a megadott karakterláncot a `buffer` változóba másolja a `strcpy()` függvény, a lefoglalt hely kezdőértékétől addig írja a verem tartalmát, amíg a bemeneti karakterlánc végét jelző `0` értékű bájtot el nem éri. Tehát ha a bemenetnek egy 100 hosszú karakterláncnál hosszabbat adunk meg, akkor a másoláskor a vermen feljebb lévő elemek is átíródnak. Láthatjuk, hogy a visszatérési cím is felülírható, tehát a támadó tetszőleges kódsorozatra képes átirányítani a program futását.

A túlcsordulás legegyszerűbb kihasználási módja az, ha egyből magában a bemenetben olyan futtatható kódsorozatot helyez el a támadó, amely például egy hálózati porton keresztül hozzáférhetővé teszi a parancssort, és a visszatérési címet úgy írja felül, hogy az erre a kódsorozatra mutasson. Így a függvényvisszatéréskor a támadó kódja lefut, ezáltal csatlakozni tud a géphez és átveheti az irányítást. Ha nem lokális változóként deklarálunk egy puffert, hanem dinamikusan foglaljuk, akkor nem a stacken, hanem a heapen allokálódik számára hely. Ez hasonlóképpen túlírható és elérhető, hogy tetszőleges memóriacímet tetszőleges értékre módosítsunk, ami a fent említett támadásokra, azaz tetszőleges kódsorozat futtatására biztosít lehetőséget [4].

Egész számokkal kapcsolatos hibák

Az egész számokkal kapcsolatos hibák alapvetően a számítógépek számábrázolásának korlátai miatt jelentkeznek, illetve a nem megfelelő hiba- vagy kivételkezelés miatt. Ezek a hibák általában nem okoznak önmagukban sebezhetőséget, de nagyon gyakran miattuk jönnek létre puffer túlcsordulásra lehetőséget adó sebezhetőségek [5].

a) Aritmetikai túlcsordulás

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

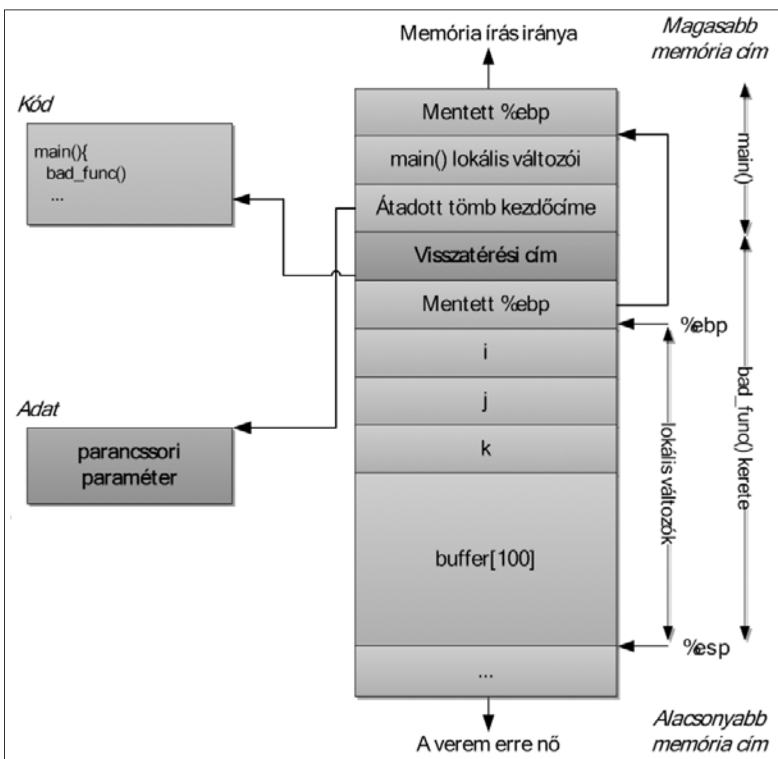
b) Előjelezési hiba

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

c) Eltérő bitszélesség

Előfordulhat, hogy egy nagyobb méretű változót (például egy 32 bites integert) szeretnénk kisebb területen eltárolni (például egy 16 bites short változó helyén), amely nem képes azt befogadni, ezért az érték csonkolódik. Természetesen egy csonkolódott változóval való memóriafoglalás vagy paraméterellenőrzés is okozhat sebezhetőséget.

3. ábra A verem állapota



A printf() formátumleíró helytelen használata

A szabványos C könyvtár közkedvelt kiíró függvénye a `printf()`, illetve annak változatai. Ezek előnyös, jól használható szolgáltatása, hogy egy formátumleíró sztring megadásával egyszerűen leírható, hogy a megadott, különböző típusú paraméterek, a megjelenített szövegben hol és milyen alakban jelenjenek meg.

Abban az esetben azonban, ha a formátumleíró kívülről módosítható, az támadásra ad lehetőséget [6]. Egy ilyen tipikus hiba látható a *4. ábrán*.

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

4. ábra printf() hibás használata

A parancssori paraméterben vezérlő karaktereket elhelyezve, a támadó olyan „hibás működést” képes előidézni, amely révén információkat tud kiolvasni a program memóriájából, manipulálni képes memóriacímek tartalmát és akár át is tudja venni a vezérlést a támadott gép felett.

Például a fenti programot a „`%X%X%X`” paraméterrel meghíva, a program kiírja hexadecimális számrendszerben a vermen tárolt értékeket (amelyek között titkosnak minősülő adatok is lehetnek). A „`%s%s%s`” paraméterrel pedig mutatóként értelmezi a stack-en található értékeket, így nem csak a veremből, hanem e pointerek által mutatott memóriatartományból is egyszerűen kiírattathatunk információkat, melyek szintén lehetnek bizalmas jellegűek (egy rejtjelkulcs vagy jelszó).

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

Web-es típushibák

Napjainkban egyre nagyobb hangsúlyt kapnak a webes szolgáltatások, ezért az e rendszerekben előforduló egy-két típushibáról is említést teszünk. Ahogyan az előző, C/C++ programoknál előforduló hibák esetében is, itt is egyrészt a nem megfelelő bemenetellenőrzés, valamint a mögöttes rendszer felépítése, tulajdonságai hibáztathatóak sebezhetőségekért.

a) Parancs befecskendezés

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

b) SQL befecskendezés

Olyan rendszereknél, ahol a háttérben egy SQL adatbázis működik, a felhasználó által megadott adatok sokszor egy SQL parancsba vagy lekérdezésbe ágyazódnak bele. Ilyenkor, ha a támadó SQL parancs elemeket illeszt az általa megadott adatokba, az eredeti parancs értelmét meg tudja változtatni, tipikusan például egy adatbázisból történő jelszóellenőrzést meg tud kerülni.

c) Cross-site scripting (XSS)

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

5. Hagyományos védekezési módszerek

A hagyományos védekezési módszerek a szoftverekben rejlő potenciális sebezhetőségek ellen védekeznek valamilyen módon magán a hoszton, vagy a belső hálózat határvonalán.

Hozzáférés védelem

Az operációs rendszerek (OS) egyik fő feladata a számítógépes biztonság egyik alapelvének betartatása, miszerint minden modul (felhasználó, processz) csak azokhoz az erőforrásokhoz férjen hozzá, amire feltétlenül szüksége van (least privilege principle). Az OS által biztosított hozzáférés védelmi mechanizmusokkal határt szabhatunk az egyes támadások lehetőségeinek, de eliminálni őket nem tudjuk. Szigorúbb hozzáférésvédelmi politikát valósíthatunk meg továbbá olyan megoldásokkal, mint például a SELinux [7].

Memóriavédelem

A puffertúlcsordulásnál láttunk, hogy a támadó általában a vermen vagy a heapen futtatja az általa a memóriába juttatott kódot. Ezeket a memóriaterületeket „nem futtatható” területekként kell megjelölni. Ezt biztosítja például Windows rendszerekben a Data Execution Prevention (DEP), vagy Linuxon a PaX vagy az Exec Shield. Ezek a megoldások természetesen nem védenek minden támadás ellen, például a „Return-to-libc” [8] támadással megkerülhetőek.

Nagy segítség a támadó számára a kihasználáskor, hogy a virtuális memóriában, azonos architektúrán a

memória címek állandóak. Így tudhatja, hogy mit mire kell átírni, és hogy bizonyos „hasznos” függvények hol találhatók.

Egy másik lehetőség, hogy megnehezítsük ilyenkor a támadó dolgát, ha ezt a memória elrendezést véletlenszerűvé tesszük úgy, hogy mindig változtatunk a kódszegmens, a programkönyvtárak, a stack és a heap báziscímén. Ezt a technikát ASLR-nek (Address Space Layout Randomization) hívjuk. Természetesen ezt a védelmet is ki lehet játszani, de a sikeres támadások számát csökkenteni képes.

Védekezés ismert rosszindulatú kódok ellen

A vírusirtók és behatolás detektáló rendszerek olyan rosszindulatú programok és támadások ellen képesek védeni elsősorban, melyek a múltból már ismertek. Ahogyan a 2. szakaszban arra már utaltunk, ezek a kiegészítő szoftverek ugyanúgy növelik a rendszer komplexitását, mint bármilyen más alkalmazás, így a sebezhetőség potenciált is. A veszélyt fokozza, hogy különösen az antivírus és IDS szoftverek mélyen beépülnek az operációs rendszerekbe, ezért egy esetleges biztonsági lyuk teljes körű hozzáférést képes biztosítani egy potenciális támadó számára.

Hálózati határvédelem

A hálózati rétegben is védekezhetünk a támadások ellen. Tűzfalak segítségével kikényszeríthetjük a hálózathozzáférési politikánkat. Ez gyakorlatilag ugyanazt jelenti, mint az operációs rendszer hozzáférésvédelme, csak a hálózati erőforrásokról van szó. Segítségükkel leszűkíthetjük a támadási felületet, de természetesen az elérhetően maradt szolgáltatásokban lévő sebezhetőségek ellen nem véd.

Foltozás

Mivel nem hibamentesen kerülnek ki a szoftverek a fejlesztőktől, egy hiba felbukkanása után azt utólag kell kijavítani. Nagyon fontos, hogy ezek a hibajavítások minél gyorsabban jussanak el a felhasználókhoz, és fel is települjenek. Ahogy azt már említettük, a reakcióidő miatt az idő nagy részében a sebezhetőségek kihasználásra várnak.

Látható, hogy az eddig felsorolt védelmi mechanizmusok mind a számítógépre telepített szoftverekben már meglévő sérülékenységek *kihasználását* nehezítik, vagy a támadási felületet szűkítik. Ezek a technikák képesek a kockázatok bizonyos mértékű enyhítésére, azonban mivel ezek természetüket tekintve reaktív jellegűek, nem előzik meg a sebezhető pontok kialakulását.

6. Megelőzés a fejlesztés során

A sebezhetőségek kialakulásának elkerülése, megakadályozása, vagy még időben való detektálása a szoftverfejlesztők feladata. Effajta preventív technikák alkalmazására a szoftverfejlesztés életciklusának minden állomásán szükség van.

Védekezés a szoftverfejlesztés folyamán

Biztonsági szempontokat is figyelembe vevő szoftverfejlesztésnél már a követelmények meghatározásánál számolni kell a lehetséges fenyegetettségekkel. Praktikus gyakorlat a használati esetek (use case) mellé a potenciális visszaélési eseteket is meggondolni és felsorolni. Az architektúrális és részletes tervezés folyamataiba is kockázatelemzés bevonása szükséges, számba véve a lehetséges hibákat, sebezhetőségeket. Az implementáció folyamán érdemes a manuális kódszemlézést automatikus statikus kódelemző eszközökkel segíteni. A tesztelés fázisában pedig egyrészt a biztonsági funkciókat a hagyományos tesztelési módszerekkel kell ellenőrizni, majd a teljes rendszeren egy veszélyalapú biztonsági tesztelést kell végrehajtani.

Típusbiztos programozási nyelvek használata

A használt programozási nyelv nagymértékben befolyásolja egy szoftver támadhatóságát. A mai operációs rendszerek, eszközmeghajtók és rengeteg felhasználói szoftver is C illetve C++ nyelven íródik. Ezek a nyelvek túl sok szabadságot adnak a programozónak, hogy elég biztonságosak lehessenek. Más programozási nyelvek (pl. Java, C#, Python) szigorú típusossággal (típus-biztonság), mutatók kiküszöbölésével és egyéb biztonsági megfontolásból bevezetett szabályokkal és megkötésekkel eleve kiküszöbölnek olyan tipikus hibákat, mint például a C programokban sokszor előforduló puffer túlcsordulás. Természetesen vannak területek, ahol a C/C++ használata elkerülhetetlen, például teljesítménykritikus szoftvereknél. Megjegyezzük, hogy léteznek, még ha csak kutatási célból is, olyan C nyelvre alapuló módosított nyelvek illetve fordítók is, amelyeket úgy terveztek, hogy ne lehessen elkövetni használatuk során a legtipikusabb hibákat (pl. Cyclone [9] vagy Ccured [10]).

Statikus kódelemzők

Az elmúlt években a statikus kódelemzők használata nagymértékben elterjedtté vált a manuális kódszemlézés kiegészítőjeként, illetve automatizálásaként. Ezen kereskedelmi forgalomban kapható programanalizáló eszközök használata ma már sok szoftverfejlesztő cég fejlesztési folyamatának része. Több ilyen automatikus statikus analízist végrehajtó szoftver is létezik, amely képes kimutatni bizonyos tipikus biztonsági szempontból veszélyes programozói hibát, több száz hibadefiníció illetve szabály alapján, a forrást elemezve [11]. Ilyenek például a FindBugs, a Coverity vagy a Fortify Source Code Analyze. E szoftvereknek az előnye az, hogy elvben teljes kód-lefedettséget tudnak biztosítani (a gyakorlatban ez nem mindig kivitelezhető). Hátrányuk pedig, hogy általában túl nagy a hibás riasztások (false positive) aránya. Ez nagyban meg tudja nehezíteni használójuk munkáját.

Dinamikus feketedoboz-alapú biztonsági tesztelés

A Washingtoni Egyetem kutatói 1990-ben úgy teszteltek szabványos UNIX alkalmazásokat, hogy hosszú, véletlenül generált inputot küldtek a programok beme-

netére [12]. A tesztelt programoknak körülbelül 30%-a elszállt, vagy kiakadt. A módszert „fuzzing”-nak nevezték el. Hogy a technika a biztonsági tesztelésre még alkalmasabbá váljon, párhuzamosan többen is, két szempontból fejlesztették azt tovább.

Ez egyik oldala a fejlődésnek, hogy figyelembe veszszük a szoftver által elvárt bemeneti struktúráját is (pl. egy Word dokumentum) és egy struktúra leírás alapján generálunk véletlen módon helytelen, de strukturálisan helyes bemenet. Ezzel nagyban növelhető a kódlefedettség.

A másik, hogy a tipikus hibákra általában meghatározhatók olyan bemeneti minták, melyek az adott típushibát relatíve nagy valószínűséggel felszínre képesek hozni. Ilyen minták például a puffer túlcsordulásnál a nagyon hosszú, vagy éppen üres bemenet, vagy a lezáró `0x00` karakter hiánya. Az egészekkel kapcsolatos hibákat, a nulla, a negatív, a nagyon kicsi, illetve nagyon nagy értékek vagy a 2 hatványai idézhetik elő könnyen.

Folytathatnánk a sort a `printf()` hibával („`%s%s%s`” vagy „`%n%n%n`”), SQL injection-nel (`' OR username IS NOT NULL OR username = '`, vagy `1' OR '1'='1`, `...`) és így tovább. Ha a véletlen tesztvektor generálás során olyan heurisztikákat alkalmazunk, melyek ilyen mintákat hoznak létre, még tovább növelhetjük a hibák megtalálásának valószínűségét.

Tehát ennél a biztonsági tesztelési módszernél egy olyan feketedoboz-alapú tesztelést hajtunk végre, amelynél (ellentétben a hagyományos teszteléssel) nem a funkciók specifikációi alapján határozzuk meg a teszteseteket, hanem a potenciális programozói hibák által kialakulható sebezhetőségek alapján. Ilyen fejlett fuzzing eszközök a Peach [13] vagy a Flinder [14], amely olyan összetett protokoll-implementációkat is képes tesztelni, mint például az SSL könyvtárak [15].

Ezen eszközök előnye, hogy minden talált hiba valós hiba (csak true positive), ellentétben a statikus elemzőkkel, viszont az általuk elérhető kódlefedettség jóval kisebb. Említést kell tennünk arról is, hogy természetesen ezeket az eszközöket a támadók is használják a kész szoftvereken, ezért fontos, hogy még a fejlesztői oldalon megtörténjenek ezek a vizsgálatok.

7. Mit tartogat a jövő?

Ebben a szakaszban néhány reményteli kutatási irányzatot mutatunk be, melyek nagyban hozzájárulhatnak a jövőben a biztonságosabb szoftverfejlesztéshez és megbízhatóbb szoftverekhez.

A jövő megoldásai elsősorban a formális módszerek (tételbizonyítók, modellellenőrzők) fejlesztésére, használatuk megkönnyítésére, a szoftverfejlesztési folyamatba való teljes körű beépítésére alapulnak. Pár éve Tony Hoare meg is hirdette az „Informatika nagy kihívása” projektet [16], melynek végcélja egy olyan eszközkészlet megalkotása, mely teljes és automatikus programverifikációra képes.

Mi most két területet szeretnénk kiemelni: az egyik az automatikus kódalapú teszteset-generálás, a másik a programozási nyelv alapú biztonsági megoldások.

Automatikus szoftvertesztelés

A kódszemlézés automatizálása a statikus kódelemzők fejlődésének köszönhetően ma már viszonylag jól használható technológia. Ami viszont a következő évek egyik nagy kutatási feladata az a szoftvertesztelés – minél kiterjedtebb – automatizálásának megoldása.

A fuzzing roppant népszerű módszer lett az elmúlt években, hiszen viszonylag egyszerűen, gyorsan és olcsón lehetett vele akár nagyon komoly biztonsági hibákat is találni bármilyen szoftverben. Annak ellenére, hogy ez valóban hatásos módszer, nagyon komoly korlátai is vannak.

Képzeljük csak el, hogy egy kihasználható programozó hiba például egy `if (x==12)` utasítás igaz ágán helyezkedik el, ahol `x` az egyik bemeneti változó. Annak a valószínűsége, hogy egyáltalán egy teszteset során futni fog a hibás kódrészlet, egy 32 bites `x` változó esetén $1/2^{32}$, hiszen az x bemenetet véletlen módon generáljuk. Éppen ezért a fuzzing általában nagyon kicsiny kódlefedettséget biztosít.

A Microsoft Research kutatói olyan megközelítéssel álltak elő a probléma kezelésére, amit „whitebox fuzzing”-nak neveztek el. A megoldás a dinamikus tesztgenerálás és a szimbolikus futtatás [17] meglehetősen régi ötletére alapszik. Egy véletlenszerűen választott kezdeti bementettel szimbolikusan futtatják a vizsgált programot, miközben bemeneti kényszereket gyűjtenek az érintett feltételes elágazások alapján. Az összegyűjtött kényszereket a bemeneti adatokra vonatkozóan azután szisztematikusan negálják, majd kényszermegoldó eljárásokkal (constraint solver) újabb bemeneteket, teszteseteket generálnak, melyek már a program más részeit hozzák működésbe.

Például az előbbi példát tekintve, a szimbolikus futtatás során az `if (x==12)` elágazásnál, egy `x=0` kezdeti változó az `x≠12` kényszert hozza létre. Ha ezt a kényszert negáljuk és megoldjuk, akkor az `x=12` értéket kapjuk, mint következő tesztesetet, ami már lefedi az elágazás igaz ágát, ahol a feltételezett hibánk el van rejtve [18].

Nyelvalapú biztonság

Ahhoz, hogy a jövőben eleve kizárjunk bizonyos biztonsági szempontból veszélyes programozási hibákat, sokkal mélyebb szinten, alapjaiban kellene megváltoztatni a programozási nyelveket, fordítóprogramokat, valamint a futtatókörnyezeteket. Az ilyen típusú megoldások területe az úgynevezett nyelvalapú biztonság (language-based security), melyet a terület egyik éllovasa Schneider [19] a következőképpen definiált, meglehetősen tág értelmezésben: „azon módszerek halmaza, melyek a programozási nyelvek elméletére és implementációjára alapozva, beleértve ide a szemantikákat, típusokat és az optimalizálást, a biztonság kérdésére próbálnak megoldást nyújtani”.

Az egyik ötlet amit „Proof-Carrying Code”-nak [20] nevez az irodalom az, hogy a fordító generáljon formális bizonyítást arra vonatkozóan, hogy a lefordított program megfelel a különböző biztonsági feltételeknek, majd ezt a bizonyítást mellékelje a lefordított állományhoz. A programot futtató hoszt ezt a bizonyítást ellenőrizheti a futtatás előtt, hogy megbizonyosodjon róla, hogy a szoftver megfelel a követelményeknek.

Ide tartozik még egy másik javaslat is, amely a „Typed Assembly Language” [21] nevet kapta. Ez egy olyan kibővített assembly nyelv, amely a változók típusinformációit is magában képes foglalni. Így futtatás előtt meg lehet bizonyosodni a lefordított állomány típusbiztonságáról anélkül, hogy olyan köztes bájtkód-reprezentációkat használnánk, mint amilyet a Java vagy .NET platformok.

8. Összefoglalás

Megállapítottuk, hogy az informatikai rendszerek biztonságának kérdése a legnagyobb részben valójában szoftverminőségi kérdés. Amíg a szoftverek minőségében nem történik áttörő fejlődés, addig a szoftverbiztonság, így az egész IT biztonság terén sem fogunk lényeges javulást tapasztalni. Az is egyértelművé vált, hogy nem létezik egy minden problémára orvosságot nyújtó megoldás. Egyszerre kell a szoftverfejlesztési metodológiákat, a programozási nyelveket, fordítókat és operációs rendszereket, futtatókörnyezeteket alapjaiban megváltoztatni.

A formális módszerek fejlődése sokat ígér, de ne feledjük, hogy programozói hibák mindig voltak, vannak és lesznek is, így várhatóan az azokból adódó sebezhetőségek sem fognak teljes mértékben megszűnni.

A szerzőkről

SZEKERES LÁSZLÓ 1983-ban született Szegeden. 2007-ben diplomázott mérnök informatikusként a BME Méréstechnika és Információs Rendszerek Tanszékén, Infokommunikációs Rendszerek Biztonsága Szakirányon. Jelenleg biztonsági kutató-fejlesztőként dolgozik a SEARCH-LAB Biztonsági Értékelő Elemző és Kutató Laboratóriumban. Érdeklődési területe a számítógépes biztonságon belül a szoftverbiztonság, biztonsági tesztelés, biztonságos programozási nyelvek és típuselmélet. CISA és CISSP vizsgákkal rendelkezik.

TÓTH GERGELY (CISA) a SEARCH-LAB Kft. értékelő és tesztelő csoportjának vezetője. A Budapesti Műszaki és Gazdasátudományi Egyetemen végzett mérnök-informatikusként és 10 éve foglalkozik IT biztonsággal. Számos magyar és EU K+F projektben vett részt és több mint 20 ipari biztonsági értékelési projektet vezetett. Emellett a SEARCH-LAB által fejlesztet automatikus biztonsági tesztelő keretrendszer, a Flinder vezető fejlesztője.

Irodalom

[1] C. Collberg, C. Thomborson, „Watermarking, tamper-proofing and obfuscation – tools for software protection”, IEEE Transactions on Software Engineering, Vol. 28, 2002, pp.735–746.

[2] R. Giobbi, „Avast! antivirus buffer overflow vulnerability”, US-CERT 2007, http://www.kb.cert.org/vuls/id/125868

[3] „National Vulnerability Database”, http://nvd.nist.gov/

[4] Matt Conover and w00w00 Security Team, „w00w00 on Heap Overflows”, 1999.

[5] Blexim: „Basic Integer Overflows,” Phrack, Vol. 11, 2002.

[6] A. Thuemmel: „Analysis of format string bugs,” Manuscript, 2001.

[7] B. McCarty: SELinux, O&Reilly, 2004.

[8] J. Pincus, B. Baker, „Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns,” IEEE Security & Privacy, 2004, pp.20–27.

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

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

[11] „Static code analysis,” Software, IEEE, Vol. 23, 2006, pp.58–61.

[12] B.P. Miller, L. Fredriksen, B. So, „An empirical study of the reliability of UNIX utilities,” Communications of the ACM, Vol. 33, 1990, pp.32–44.

[13] M. Eddington: „Peach fuzzer framework”, http://peachfuzzer.com/

[14] SEARCH-Lab Ltd.: „Flinder”, http://www.flinder.hu/

[15] L. Szekeres, „Biztonsági szempontból veszélyes programozói hibák felderítése automatizált módszerekkel”, Tudományos Diákköri Konferencia, BME, 2006.

[16] C. Hoare, „The Verifying Compiler: A Grand Challenge for Computing Research”, Modular Programming Languages, 2003, pp.25–35. http://www.springerlink.com/content/t96b79tanjm9d4tc

[17] J.C. King, „Symbolic execution and program testing”, Commun. ACM, Vol. 19, 1976, pp.385–394.

[18] P. Godefroid et al., „Automating software testing using program analysis” Software, IEEE, Vol. 25, 2008, pp.30–37.

[19] F.B. Schneider, G. Morrisett, R. Harper, „A Language-Based Approach to Security,” Lecture Notes in Computer Science, Vol. 2000, 2001, pp.86–101.

[20] G.C. Necula, „Proof-carrying code,” Proc. of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Paris, France, ACM, 1997, pp.106–119. http://portal.acm.org/citation.cfm?doid=263699.263712

[21] G. Morrisett et al., „TALx86: A realistic typed assembly language,” In Second Workshop on Compiler Support for System Software, 1999, pp.25–35.