Mérési útmutató a „Szoftver implementációs biztonsági rések vizsgálata (MEMC)” című méréshez

2023. március 6.

A mérést kidolgozta:

Gazdag András

BME, CrySyS Adat- és Rendszerbiztonság Laboratórium

1. Bevezetés

A mérés célja, hogy a hallgatók megismerkedjenek a szoftver implementációs hibák néhány fontosabb kategóriájával. A hibák felismerésén felül a mérés során a hallgatók néhány fontosabb segédeszköz használatát is elsajátíthatják, melyek segítségével implementációs hibák kihasználására is lehetőség nyílik. Ezután, utolsó lépésként, a hallgatók feladata, hogy különböző védekezéseket próbáljanak ki a korábban elkészített támadásokkal szemben.

Hozzáférés az infrastruktúrához

A mérés során virtuális gépeken van előkészítve a hallgatóknak a környezet. A gépeket a jelenléti oktatás során, a *vCenter* szolgáltatáson keresztül lehet elérni amelyhez a hozzáférést a mérésvezető adja majd meg. A virtuális gépbe a `crysys` / `labor` adatokkal lehet bejelentkezni.

2. Elméleti összefoglaló

A méréshez szükséges anyag elméleti részei a *Számítógép biztonság* kurzus Memory Corruption diáiban találhatóak.

Az ott található alapokon kívül a további részletek nyújthatnak segítséget a felkészülésben, valamint a mérés elvégzésében.

2.1. Fordítás

A mérés során sokszor szükség van az elemzett alkalmazást újrafordítani a megfelelő kapcsolókkal. Ezt hatékonyan `make` segítségével lehet megoldani. Minden feladathoz mellékelve található egy *makefile*, ami az összes szükséges konfigurációt tartalmazza a munka megkönnyítéséhez. A mérés elején javasolt a makefile átnézése!

2.2. Hatékony paraméterátadás

A mérés során elemzett alkalmazások a legtöbb esetben command line paramétereket dolgoznak fel. Egy támadás során gyakran előfordul, hogy nagy számú egyforma karakter átadására van szükség. Ezt hatékonyan meg lehet tenni, ha a bash képességeit valamelyik script nyelvvel kombináljuk. Egy egyszerű példa python használatával:

```
./app "$(python -c 'print "A"*4 + "\x01\x02\x03\x04"')"
```

2.3. Hasznos GDB parancsok

A mérés során az alkalmazások vizsgálatára gdb használata javasolt. Az ismertebb utasításokon túl a következő parancsok lehetnek még hasznosak:

- **b *<address>**: Breakpoint elhelyezése egy megadott címre.
- **print foo**: A foo függvény címe a memóriában.
- **print 'malloc@plt'**: A malloc függvény címe a plt táblában[1]. A külső library-kből hívott függvények indirekten kerülnek meghívásra a .plt táblán keresztül.
- **disas**: Egy adott függvény, cím disassembly-je.
- **x/[num]x <address>**: num*8 byte hosszan kiírja egy memóriaterület tartalmát.

2.4. Függvény hívás folyamata

A függvény hívás során a stack kezelés egy része a hívó függvényben, egy része pedig a hívott függvényben valósul meg.

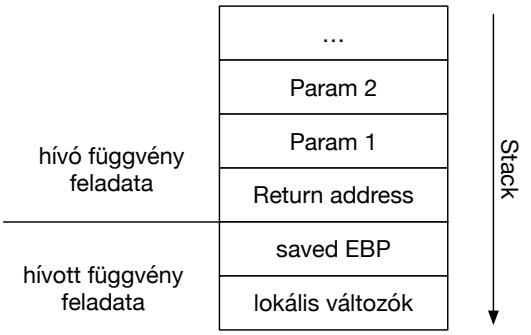
A hívó függvény feladata

Mielőtt a végrehajtás átugorhat egy függvényre, a hívó függvénynek két feladata van. Első lépésként a hívott függvény futásához szükséges paramétereket kell elhelyezni a stack-en, amiknek a sorrendje a hívási konvenciótól függhet. Második lépésként pedig egy visszatérési cím elhelyezése a feladat. Ez alapján tudja a hívott függvény, hogy melyik címen kell majd folytatni a végrehajtást a futás után.

A hívott függvény feladata

A hívott függvény folytatja tovább a stack frame felépítését. A hívó előkészítése után az EBP regiszter elmentése következik. Ezután utolsó lépésként már csak a lokális változóknak szükséges terület lefoglalása szükséges, majd indulhat a függvény tényleges működésének a végrehajtása.

Ezek alapján az előkészített stack a következő képen néz ki:



[1] https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

3. Feladatok

3.1. Buffer overflow bevezetés

Az első feladatban egy egyszerű buffer overflow megvalósítása a cél. A sérülékeny alkalmazás a ~`/developer/task-1` mappában található. A támadás célja, hogy az alkalmazás végrehajtását eltérítve a `not_called` függvény is lefusson.

Lépések:

1. Nézze át a `makefile`-t! Keresse meg, hogy a `make` parancs kiadása esetén melyik utasítás fog végrehajtódni!
2. A forráskódot megvizsgálva keresse meg a sérülékenységet az alkalmazásban!
3. Fordítsa le az alkalmazást: `make`
4. `gdb` segítségével elemezze az alkalmazás működését futásidőben!
5. Rajzolja fel a stack-et a támadás előtti állapotban, és követlenül utána!
6. Adja meg, hogy milyen támadó input esetén érhető el a kitűzött cél!
7. Javasoljon megoldási javaslatot a problémára a következő szinteken: forráskód módosítás, fordítás, operációs rendszer!
8. Vizsgálja meg, hogy ha a stack smashing protection engedélyezésével fordítja le az alkalmazást (`make withSSP`), akkor az véd-e a támadás ellen! Magyarázza meg az eredményt!

3.2. Buffer overflow paraméterekkel

Ebben a feladatban egy paraméter átadással kiegészített buffer overflow támadás megvalósítása a cél. A sérülékeny alkalmazás a `~/developer/task-2` mappában található.

A támadás célja, hogy a sérülékenységet kihasználva hívja meg a `now_called` függvényt, miközben egy valódi függvény hívást szimulálva elhelyezi a stacken a megfelelő paramétert is.

Lépések:

1. Fordítsa le az alkalmazást: `make`
2. `gdb` segítségével elemezze az alkalmazás működését futásidőben!
3. Rajzolja fel a stack-et a támadás előtti állapotban, és követlenül utána!
4. Adja meg, hogy milyen támadó input esetén érhető el a kitűzött cél!
5. Vizsgálja meg, hogy ha ASLR engedélyezésével fordítja le az alkalmazást (`make withASLR` és `make withASLRwithPIE`), akkor az véd-e a támadás ellen! Magyarázza meg az eredményt!

3.3. Return to LibC

A harmadik feladatban egy Return to LibC megvalósítása a cél. A sérülékeny alkalmazás a ~`/developer/task-3` mappában található.

Amennyiben egy alkalmazás valamilyen library-t használ, akkor nem csak azokra a függvényekre tud egy támadó ráugrani, amelyeket a programozók írnak, hanem a betöltött library-kben található összes függvényre. Ebből kifolyólag egy támadás során bármelyik függvény a támadó segítségére lehet, tipikus esetben a LibC library-ből.

A támadás célja, hogy az alkalmazás végrehajtását eltérítve a `system` függvény lefusson a megfelelő paraméterrel.

Lépések:

1. Fordítsa le az alkalmazást: `make`
2. `gdb` segítségével elemezze az alkalmazás működését futásidőben!
3. A LibC library dokumentációja alapján határozza meg, hogy milyen paraméter megadása szükséges a cél eléréséhez.
4. Rajzolja fel a stack-et a támadás előtti állapotban, és követlenül utána!
5. Adja meg, hogy milyen támadó input esetén érhető el a kitűzött cél!
6. Vizsgálja meg, hogy ha az NX bit engedélyezésével fordítja le az alkalmazást (`make withNX`), akkor az véd-e a támadás ellen! Magyarázza meg az eredményt!

3.4. ROP

A negyedik feladatban egy ROP támadás megvalósítása a cél. A sérülékeny alkalmazás a ~`/developer/task-4` mappában található.

Egy ROP támadás során a binárisban található kis kódrészletek (gadget) újrafelhasználásából áll össze egy támadás. Az első lépés, hogy a `ROPgadget` python script segítségével vizsgálja meg, hogy milyen gadgetek állnak rendelkezésre. Ezután ezek megfelelő kombinálásával érje el, hogy ismét egy bash terminál hozzáférést lehessen szerezni! Egy ROP esetén a támadáshoz használt bemenet jelentősen hosszabb tud lenni a korábbi esetekhez képest, így a mappában található `exploit.py` script használata javasolt. Ez a script tartalmaz több segítséget, hogy milyen gadget-ek megkeresése célravezető, valamint minta példaként is szolgál, hogy hogy lehet összeállítani a támadást.

Lépések:

1. A forráskódot megvizsgálva keresse meg a sérülékenységet és a lehetséges támadást az alkalmazásban!
2. Fordítsa le az alkalmazást: `make`
3. `gdb` segítségével elemezze az alkalmazás működését futásidőben!
4. Rajzolja fel a stack-et a támadás előtti állapotban, és követlenül utána!
5. ROPgadget segítségével keresse meg az elérhető gadgeteket!
6. Az exploit.py segítségével állítsa össze a támadást!
7. Vizsgálja meg, hogy ha a stack smashing protection engedélyezésével fordítja le az alkalmazást (`make withSSP`), akkor az véd-e a támadás ellen! Magyarázza meg az eredményt!