Jegyzőkönyv

Biztonság és védelem az informatikában c. tárgy

6. feladat

Készítette:

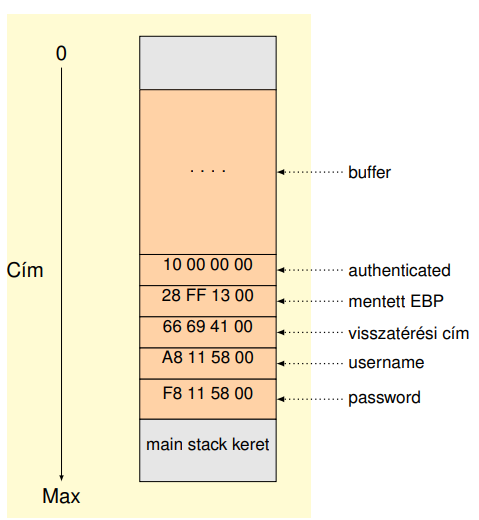
Csomor Bence, TVIK4I

2021.03.24.

**Puffer túlcsordulásos támadási módok**

A 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.

A stack elhelyezkedése:



túlcsordulás fajtái:

stack alapú (stack buffer overflow)

heap alapú (heap buffer overflow)

Fontos, hogy a stack buffer overflow nem egyenlő a stack overflow-al!

Egyik esetben csak a stack keretet vagy azon belül egy területet írunk túl úgy, hogy megváltoztatunk más változók értékét is. A másik esetben a stack hely fogy el, például végtelen rekurzió esetén.

**Klasszikus verem túlcsordulás:**

A verem túlcsordulás (stack overflow) 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.

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)

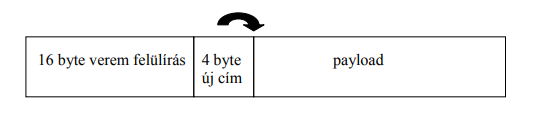
használaton kívüli hely (2 byte)

ar2 tömb lokális változó (10 byte)

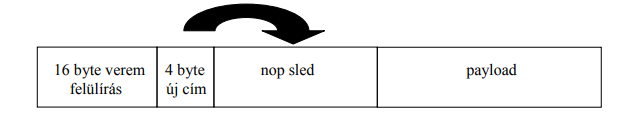
A metódus meghívása előtt a hívó metódus leteszi a verembe a metódus paramétereket, jelen esetben az argv[1]-re mutató pointert. Ezután leteszi a metódus visszatérési címét (a hívó metódus hívás utáni utasításának a címe), majd lementi a bázis pointert (push ebp). Ezután átállítja a bázis pointer értékét 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 15 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 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 az alábbi 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. Az 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.



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:



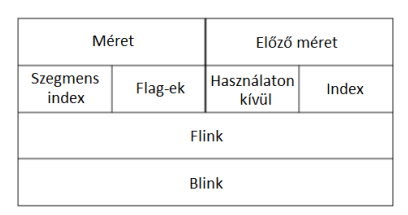
A támadó kód legfontosabb része a payload. A payload különböző támadó feladatokat láthat el, mint pl. parancssort nyithat, kinyithat egy portot, vagy akár egy tetszőleges másik programot is elindíthat. A "proof of concept" jellegű támadó kódoknál mindezek miatt a támadás sikerességét általában úgy szemléltetik, 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 shellstorm gyűjteménye.



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, illetve a stack overflow valamely továbbfejlesztett változatával foglalkoznak inkább.

**Halom túlcsordulás:**

A halom túlcsordulás (heap overflow) a túlcsordulásos memóriakorrupciós hibák egy gyakori példája, amely a heap szegmensben jön létre. 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.





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 fentebbi ábra egy darab chunk szerkezetét mutatja a szabad listában, a lentebbi 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.

A Windows néhány verziójában előfordulnak olyan hibák is, ami puffer túlcsorduláshoz vezet, erre példa egy Windows LRPC client hiba, amelyet kihasználva verem alapú puffer túlcsordulást lehet előidézni, egy speciálisan formázott LPC (Local Procedure Call) port üzenet küldésével egy testreszabott LRPC szerveren keresztül.

A sérülékenység sikeres kihasználása tetszőleges kód futtatását teszi lehetővé kiterjesztett jogosultságokkal.

Ezeket általában verziófrissítésekkel állítják helyre.

A Linux rendszerében is előfordulnak sérülékeny részek, egyik ilyen a B.A.T.M.A.N. néven futott:

A sérülékenységet egy határhiba okozza a “bat\_socket\_read()” függvényen belül (net/batman/icmp\_socket.c). Ez kihasználható a felhasználói memóriaterületen puffer túlcsordulás előidézésére speciálisan erre a célra elkészített csomagok küldésével a sérülékeny rendszernek.

A sérülékenységet a 2.6.39.4. verzióban jelentették, de egyéb verziók is érintettek lehetnek.

Ezt végül egy fejlesztői verzióban kijavították.

Manapság szinte nem is fordul elő se a Windows rendszerében sem a Linuxéban ilyes fajta sérülékeny kód, viszont vannak kisebb csoportok, akik kevesebb figyelmet fordítanak a biztonságra és általában ezeknél fordul elő puffer túlcsordulás. Ezek közül van, amit az operációs rendszer sem tud enyhíteni.

források: [https://lib.uni-obuda.hu/sites/lib.uni obuda.hu/files/Erdodi\_Laszlo\_ertekezes.pdf](https://lib.uni-obuda.hu/sites/lib.uni%20obuda.hu/files/Erdodi_Laszlo_ertekezes.pdf)

<http://www.hit.bme.hu/~izso/bizprog1/sp04.pdf>

<https://nki.gov.hu/figyelmeztetesek/serulekenysegek/microsoft-windows-lrpc-client-puffer-tulcsordulas-serulekenyseg/>

<https://nki.gov.hu/en/figyelmeztetesek/serulekenysegek/linux-kernel-b-a-t-m-a-n-bat_socket_read-puffer-tulcsordulas-serulekenyseg/>