8. Puffer túlcsorduláson alapuló támadások: Memória-kezelés, a stack működése, Assembly alapok, fuzzing, memóriavédelmek (DEP, ASLR), Immunity Debugger, mona.py

Tartalom

---

1. Alapok

1.1. Puffer túlcsordulás

- A szoftversérülékenységek egyik leggyakoribb fajtája.
- A puffer túlcsordulás egy fajtája a verem puffer túlcsordulás (nem keverendő a verem túlcsordulással).

1.2. Új sérülékenységek találása

- A forráskód elemzése (az rendelkezésre áll),
- "reverse engineering",
- "fuzzing"-gal találhatunk hibákat.

1.3. Memóriavédelmek (DEP, ASLR)

Az *SLMail* alkalmazást Data Execution Prevention (DEP) és Address Space Layout Randomization (ASLR) védelmek nélkül fordították le.

- A DEP: megakadályozza a program végrehajtása adat memória lapra (data page) ugorjon. Ha ezt mégis megpróbálná valami, akkor kivétel (exception) keletkezik.
- ASLR használatával az alkalmazás betöltött moduljainak (exe, dll) a kezdőcímei véletlenszerű helyekre kerülnek, amik minden bootoláskor megváltoznak.

1.4. Assembly alapok

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Regisztek

Általános: EAX, EBX, ECX, EDX

Mutató: ESI, EDI, EIP, ESP

A - Accumulator

B - Base

C - Counter

D - Data

Mutató

ESI - source index

EDI - destination index

ESP - stack pointer

EBP - base pointer

2. Immunity debugger

- Olvasnivaló: Help (Menü: Help > Contents)
- A "CPU" ablak részei:
  - instructions, registers,
  - Stack: stack memory, verem
  - Dump: dump
- Legfontosabb regiszterek: ESP: a stack teteje, EIP: a következő utasítás címe.

2.1. Billentyű-gyorsparancsok

parancs	jelentése
Az Insructions ablakban: Ctrl + G	Goto: ugrás egy adott címre.
CPU ablakban: F2	Int3 breakpoint elhelyezése
F9	Run (debugged application)
F7	Step into
Ctrl + F9	Execute till return
Ctrl + F2	Restart

---

3. A verem (stack) működése

3.1. Mire jó?

- Függvényhívások visszatérési címeinek tárolására,
- helyi változók tárolására.
- LIFO (last-in, first-out) szerekezet.
- A verem egy memóriaterület, amit a CPU kezel az ESP (stack pointer) regiszter segítségével.
- Az ESP-t nagyon ritkán változtatjuk közvetlenül, csak közvetetten a CALL, RET, PUSH és POP utasításokkal. Az ESP mindig az utoljára a veremre helyezett (push) értékre mutat. 32-bites üzemmódban ezek az értékek 32 bit hosszúak.

3.2. Használata

- LIFO (last-in, first-out) szerkezet.
- A verem egy memóriaterület, amit a CPU kezel az ESP (stack pointer) regiszter segítségével.
- Az ESP mutatja a tetejét, az utoljára a veremre helyezett (push) értéket. 32-bites üzemmódban ezek az értékek 32 bit hosszúak.
  - Az ESP-t nagyon ritkán változtatjuk közvetlenül, csak közvetetten a CALL, RET, PUSH és POP utasításokkal. Az ESP mindig az utoljára a veremre helyezett (push) értékre mutat.
- Call utasítás után a stack tetején lesz a call utáni utasítás címe.

3.3. A verem keret (stack frame)

stack frame tartalom	kód példa	stack frame creation step
1. passed arguments		Passed arguments, if any, are pushed on the stack.
2. subroutine return address	`CALL 00463CA4`	The subroutine is called, causing the subroutine return address to be pushed on the stack.
3. EBP	`PUSH EBP` `MOV EBP,ESP`	• As the subroutine begins to execute, EBP is pushed on the stack. • EBP is set equal to ESP. From this point on, EBP acts as a base reference for all of the subroutine parameters. • Ennek eredményeként az EBP tartalma = az előző EBP tartalom memóriában tárolt címe (rekurzívan). • Egyéb: stack frame pointer (sfp), saved frame pointer
stack cookie		Más néven: canary.
4. local variables	`ADD ESP,-10`	If there are local variables, ESP is decremented to reserve space for the variables on the stack.

1. hívó fél

2.

3. hívott

4

4. Példaprogram

```
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[]) {
    char buffer[64];

if (argc < 2) {
    printf("legalabb egy argument kene\r\n");
    return(1);
}

strcpy(buffer, argv[1]);
return(0);
}
```

A csatolt példában

- Strcpy hívásának előkészítése: 0040 148F

függvény hívás

00401053