Pamatove zranitelnosti

Bezpecnost informacnych systemov z pohladu praxe

Peter Svec

>reverse vysledky

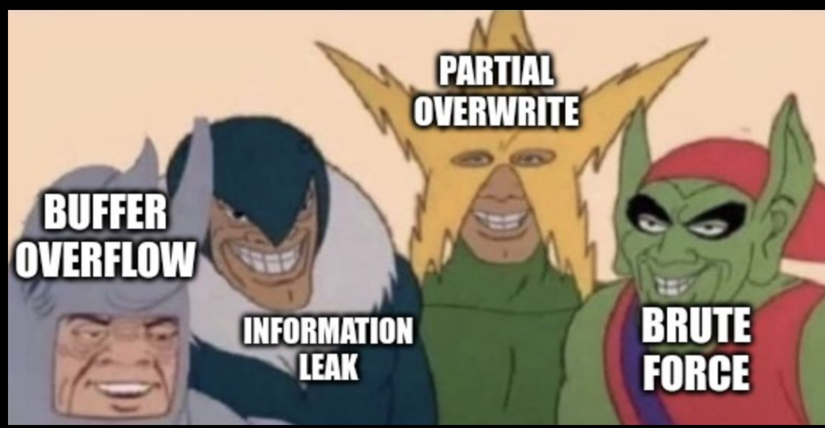
Rank		Team	Score
#1		impostor	10
#2		Exploit Sigmas	10
#3		EmYjoyers	10
#4		DePrEsSiOn_OvErFlOw	10
#5		mSUS	10

celkovy stav ->

	🥇	🥈	🥉
1. Exploit Sigmas	1	1	0
2. impostor	1	0	1
3. mSUS	0	1	0
4. EmYjoers	0	0	1

>uvod

>C je nizko urovnovy jazyk (veri vyvojarovi)
>co ak vieme pomocou chyby v programe zapisovat data mimo alokovany priestor?
>typy pamatovych chyb v ramci zasobnika
>moderne mitigacie (a techniky ako ich obist)



>problem 1

>dovera vyvojarovi

python:

```python
a = [1, 2, 3, 4, 5 ]
a[10] = 0x41
```

IndexError: list index out of range

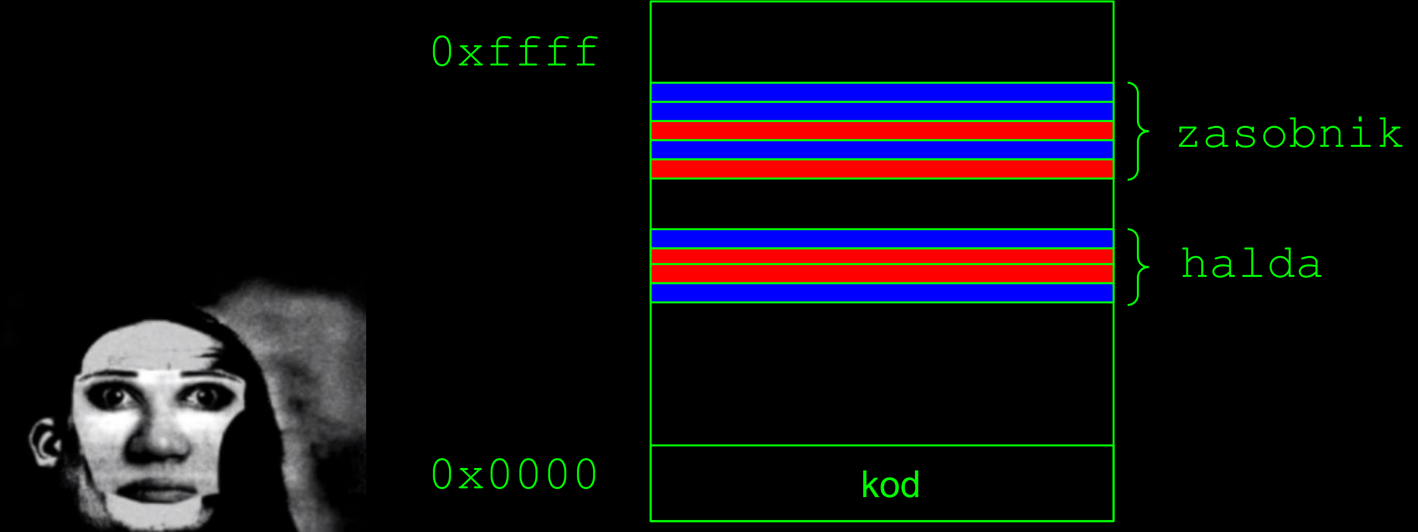
C:

```c
int a[5] = { 1, 2, 3, 4, 5 }
a[10] = 0x41;
```

Vsetko ok!

>problem 2

>mixovanie dat (aj pouzivatelsky vstup) a control flow udajov (navratove adresy, ptrs na funkcie)



>problem 3

>mixovanie dat a metadat

char data[10] = “ctf”;

c	t	f	\0	\0	\0	\0	\0	\0	\0

strlen(data) = 3

read(0, data, sizeof(data));

c	\0	f	_	b	i	s	p	p	!

strlen(data) = 1

c	t	f	_	b	i	s	p	p	!

strlen(data) = ?

>problem 4

>inicializacia a dealokacia zasobnika
>obe operacie su iba posun registrov

```c
void foo()
{
    char data[20];
    // co sa nachadza v data?
}
```

>stack smashing

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 40 * “A”

0xffff
jadro
zasobnik
RET
RBP
buffer
halda
program
0x0000

0x08

>stack smashing

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 40 * “A”

0xffff
jadro
zasobnik
RET
RBP
buffer AAAAA……
halda
program
0x0000

>stack smashing

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 96 * “A”

0xffff	
	jadro
	zasobnik
	RET
	RBP
	buffer
	halda
0x0000	program

0x0A

>stack smashing

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 96 * “A”

Segmentation fault
Invalid address 0x414141414141

0xffff

jadro
zasobnik
RET
RBP
AAAAA...... buffer
halda
program

0x0000

0x0B

>co mozeme prepisat?

>navratovu adresu
>lokalne premenne
>hodnota v pamati, ktora ovplyvnuje mat. operacie
>smerniky

>buffer overflow

>chyby pri pocitani velkosti buffra
>nebezpecne funkcie (gets, strcpy,...)

```
void vuln()
{
    char buffer[32];
    read(0, buffer, 128);
}
```

>signed/unsigned mismatch

>standardna kniznica (libc) pouziva na definovanie velkosti typ **unsigned**

```
void vuln()
{
    char buffer[80];
    int size;
    scanf("%i", &size)
    if(size > 80) exit(1);
    read(0, buffer, size);
}
```

*znamienka sa reprezentuju pomocou dvojkoveho doplnku: 0xffffffff = -1, 0xfffffffe = -2, …

>integer overflow

>co sa stane ak k max. hodnote 32 bitoveho integeru (0xffffffff) pripocitam 1?

```
void vuln()
{
    unsigned int size;
    scanf("%i", &size);
    char *buff = alloca(size + 1);
    int n = read(0, buff, size);
    buff[n] = '\0';
}
```

>mitigacie

>**ASLR** (**A**ddress **S**pace **L**ayout **R**andomization)
>**DEP** (**D**ata **E**xecution **P**revention) alebo **NX**
>**Stack Canaries**

>ASLR

- >znahodnovanie adries (offsety vsak ostavaju rovnake!)
- >plna sila **ASLR** je iba v kombinacii so zapnutym **PIE**
    - >**PIE** (**P**osition **I**ndependent **E**xecutable)

no ASLR, no PIE	**ASLR, no PIE**	**ASLR, PIE**
1.spustenie: 0x4012d4	1.spustenie: 0x4012d4	1.spustenie: 0x5571aef0b1d4
2.spustenie: 0x4012d4	2.spustenie: 0x4012d4	2.spustenie: 0x55b4a56b61d4
3.spustenie: 0x4012d4	3.spustenie: 0x4012d4	3.spustenie: 0x562b131e31d4

iba pre sekciu s kodom!

>co s tym?

>**information leak**

>ak vieme z programu precitat lubovolnu adresu, tak sme vyhrali. Offsety ostavaju rovnake a tym padom vieme vypocitat base adresu

>**partial overwrite**

>program ma stranky zarovnane na 4096 bajtov (0x1000)

>spodnych 12 bitov ostava -> 2^4 brute force

1. spustenie: 0x5571aef0b**1d4**
2. spustenie: 0x55b4a56b6**1d4**
3. spustenie: 0x562b131e3**1d4**

>**brute force**

>na 64 bitovych systemoch nepouzitelne (2^{32} BF)

>DEP

>stranky v pamati mozu mat rozne opravnenia

>**R**ead (**R**)

>**W**rite (**W**)

>**E**xecute (**X**)

>pri zapnutom NX, nemoze mat heap/stack **X** flag, (problematicke, ak chceme spustit shellcode)

>zapnute DEP je v dnesnej dobe absolutny standard

>princip ako to obist je relativne komplexny a bude nato vyhradeny cely blok (ROP)

>stack canary

>nahodna 8 bajtova hodnota, ktora chrani navratovu adresu
>pred navratom z funkcie sa skontroluje jej integrita
>56 nahodnych bitov (jedna hodnota pre vsetky funkcie)
>meni sa pri kazdom spusteni programu

priklady hodnot:

0x**3740b6e89010db**00
0x**d23db590b34909**00
0x**b22e811c4f7a2c**00

>stack canary

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 96 * “A”

0xffff

jadro
zasobnik
RET
RBP
stack canary
buffer
halda
program

0x0000

0x15

>stack canary

```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 96 * “A”

*** stack smashing detected ***
terminated

0xffff

jadro
zasobnik
RET
RBP
stack canary
AAAAA......
buffer
halda
program

0x0000

0x16

>co s tym?

- >**information leak**
  - >podobne ako pri ASLR, ak vieme precitat hodnotu kanarika, tak ju vieme pripojit do payloadu
- >**brute force**
  - >robit 2^{56} nema zmysel
  - >pri tzv. fork-servis programoch (napr. spracovanie sietovych spojeni je robene cez novy proces)
  - >pri forknuti procesu sa hodnota kanarika nemeni
  - >v takom pripade vieme aplikovat inteligentny brute force
  - >postupne prepisovanie kanarika po bajtoch
  - >max. 7 * 255 pokusov

>information leak

>**buffer overread:**

```
char buffer[16] = {};
write(1, buffer, 128);
```

>**zabudnuty NULL byte**

```
char name[10] = {0};
char flag[10] =  "feictf{...";
read(0, name, 10);
printf("Hi %s!\n", name);
```

>format string

>zranitelnost patriaca do minulosti (obcas sa vsak vyskytne, najma ako information leak)
>zvykne sa objavit aj na CTFkach
>je mozne dosiahnut aj arbitrary write (nielen read)

```c
char buffer[80];
printf(buffer);
```

vstup: ‘%p %p %p %p’
vystup: 0x7fd449f65723 (nil) 0x7fd449e860a7 0x1b

>format string

>ked chceme priamo pristupit k n-temu parametru (napr. 4ty)

```
printf("%p %p %p %p"); vypise vsetky 4 hodnoty
printf("%4$p");        vypise iba 4tu hodnotu
```

>ked chceme zapisat do adresy na 4 pozici hodnotu 100
>formatovaci retazec %n -> na adresu, ktoru poskytneme, zapise pocet doposial zapisanych bajtov

```
// na 4 adresu zo stacku zapise hodnotu 100
printf("A"*100 + "%4$n");
```

>postup

>spustim binarku, vyskusam ako funguje
>skontrolujem mitigacie (checksec)
>zreverzujem binarku (toto uz viete)
>identifikujem ako vyuzit zranitelnost (samotna zranitelnost je vacsinou nami kontrolovany overflow)
>napisem exploit (pwntools!)
>ak nieco nejde tak debugujem (gdb!)

Poznamka:
-argumenty do funkcii na 32 bit architekture sa vkladaju na zasobnik, nie do registrov