

Pamatove zranitelnosti

Bezpecnost informacnych systemov z pohladu praxe

Peter Svec

```
>reverse vysledky
```

All Time Rankings

Rank		Hacker	Score
#1		ALŠEKU	20
#2		Power_Puffers	20
#3		Slava_Ukrajine	20
#4		FIIT_Dropouts	20
#5		tichoslapos_divocakos	20

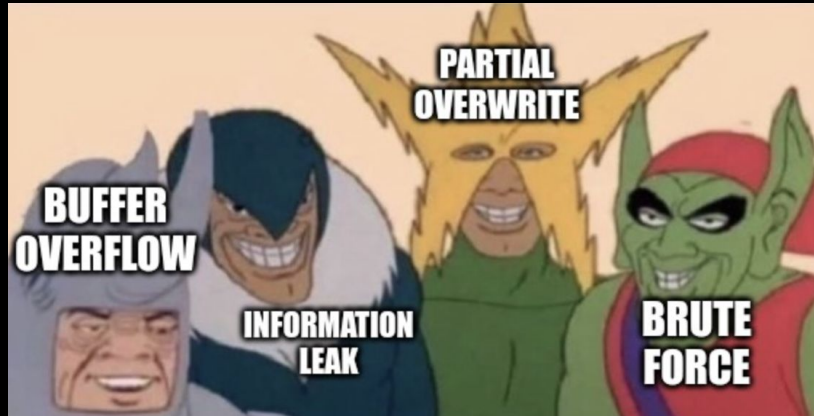
>uvod

>C je nízko úrovňový jazyk (verí vyvojárovi)

>čo ak vieme pomocou chyby v programe zapisovať data mimo alokovaný priestor?

>typy pamäťových chýb v rámci zásobníka

>moderne mitigácie (a techniky ako ich obísť)



>problem 1

>dovera vyvojarovi

python:

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

IndexError: list index out of range



C:

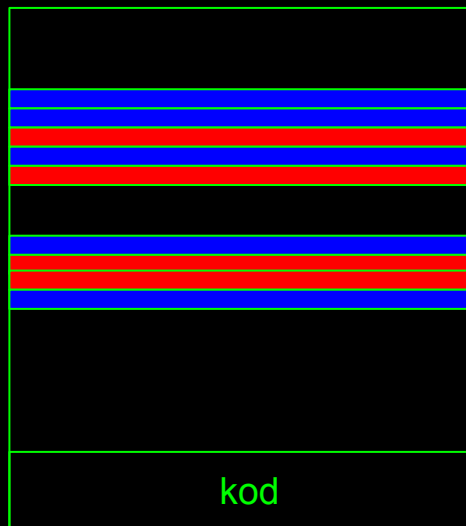
```
int a[5] = { 1, 2, 3, 4, 5 }  
a[10] = 0x41;
```

Vsetko ok!

>problem 2

>mixovanie **dat** (aj pouzivatel'sky vstup) a **control flow** udajov (navratove adresy, ptrs na funkcie)

0xffff



zasobnik

halda

0x0000

kod

0x05



>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) = ?

0x06

>problem 4

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

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



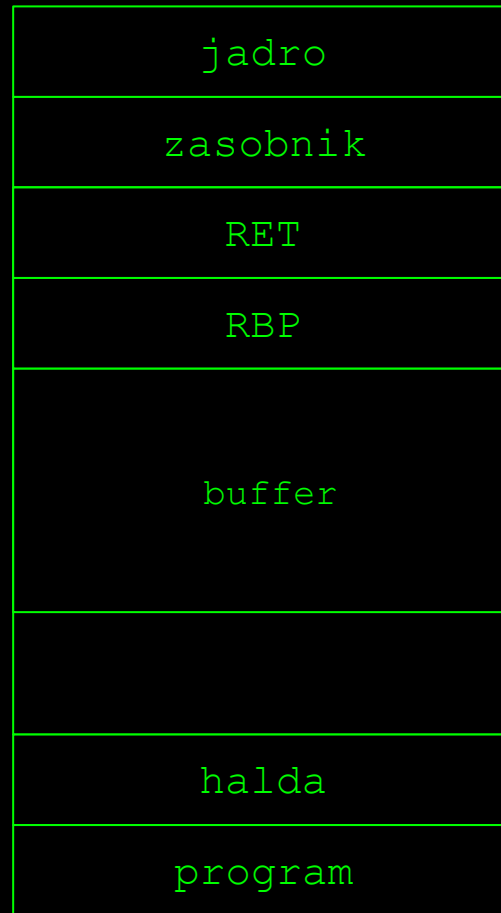
>stack smashing

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

Vstup: 40 * "A"

0xffff

0x0000



0x08

>stack smashing

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

Vstup: 40 * "A"

0xffff



0x0000

0x09

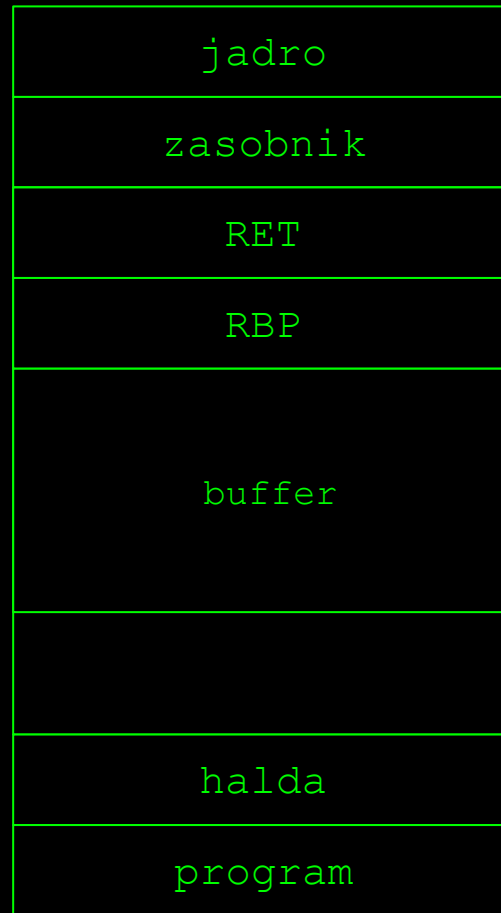
>stack smashing

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

Vstup: 96 * "A"

0xffff

0x0000



0x0A

>stack smashing

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

Vstup: 96 * "A"

Segmentation fault
Invalid address 0x414141414141

0xffff



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);
}
```

>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

1.spustenie: 0x4012d4

2.spustenie: 0x4012d4

3.spustenie: 0x4012d4

ASLR, no PIE

1.spustenie: 0x4012d4

2.spustenie: 0x4012d4

3.spustenie: 0x4012d4

ASLR, PIE

1.spustenie: 0x5571aef0b1d4

2.spustenie: 0x55b4a56b61d4

3.spustenie: 0x562b131e31d4

iba pre sekciu s kodom!

0x11

>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: 0x5571aef0 **b1d4**

2.spustenie: 0x55b4a56b **61d4**

3.spustenie: 0x562b131e **31d4**

>**brute force**

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

>DEP

>stranky v pamati mozu mat rozne opravenia

>**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

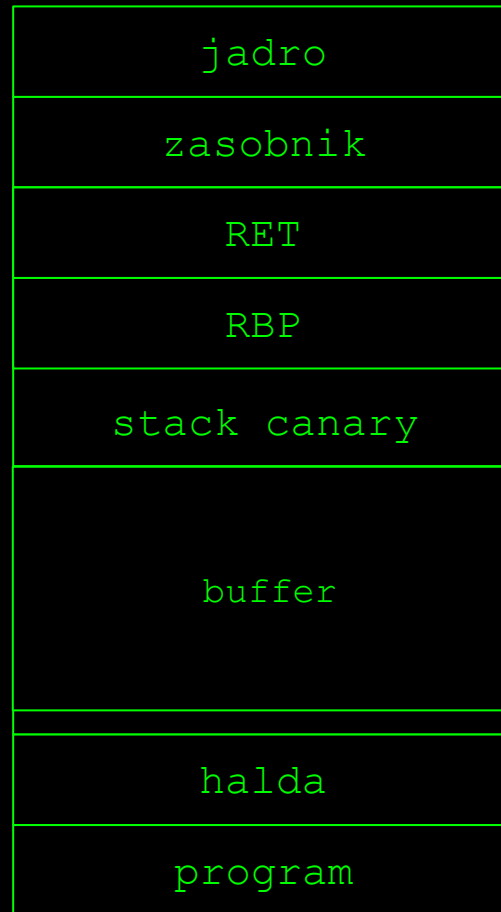


>stack canary

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

Vstup: 96 * "A"

0xffff



0x0000

0x15

>stack canary

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

Vstup: 96 * "A"

*** stack smashing detected ***
terminated

0xffff



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 forknutí 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);
```

>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);
```

>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!)