



# Pamatove zranitelnosti




Bezpecnost informacnych systemov z pohladu praxe

Peter Svec

>reverse vysledky

Rank		Team	Score
#1		Breaking Bad Bits	9
#2		Team	9
#3		キコ_team	9
#4		Yonathan	9
#5		' DROP Database BISPP; -	9

celkovy stav ->

				
1. Breaking Bad Bits	1	0	0	(3)
2. Team	0	1	0	(2)
3. キコ_team	0	0	1	(1)

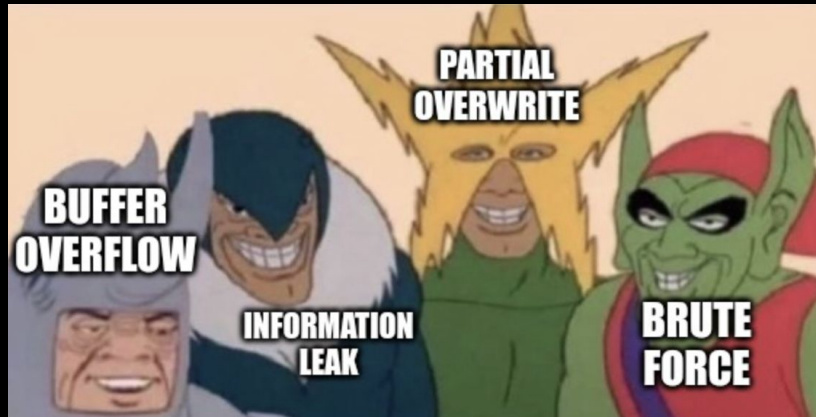
# >uvod

>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:

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

IndexError: list index out of range



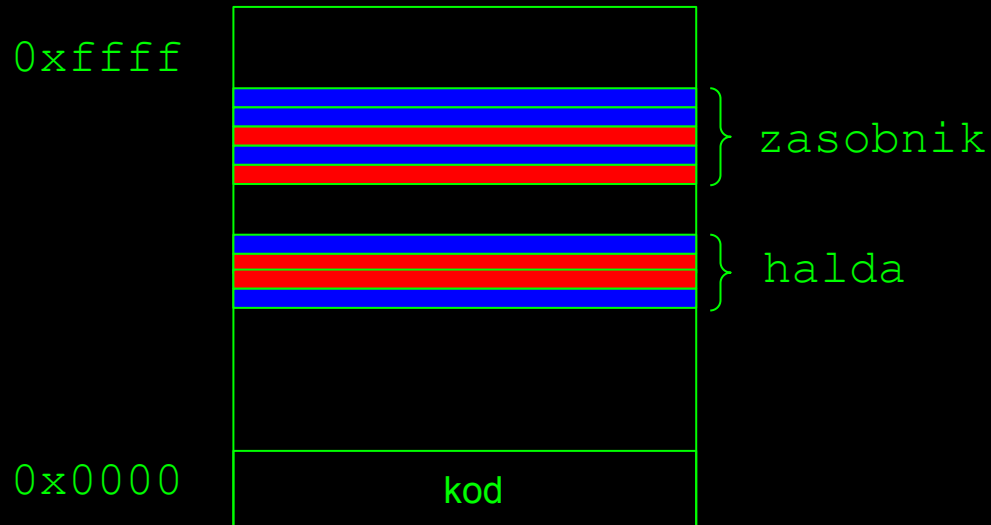
C:

```
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)



# >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

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

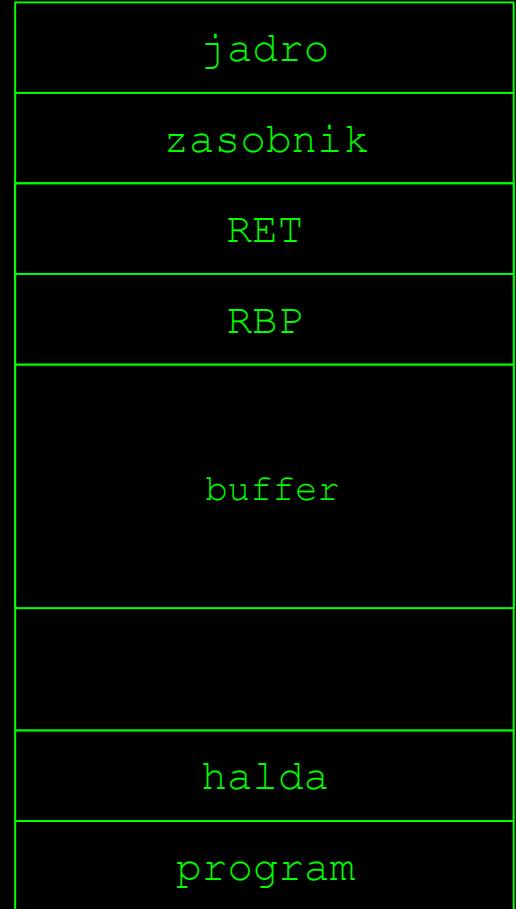


# >stack smashing

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

**Vstup:** 40 \* "A"

0xffff



0x0000

0x08

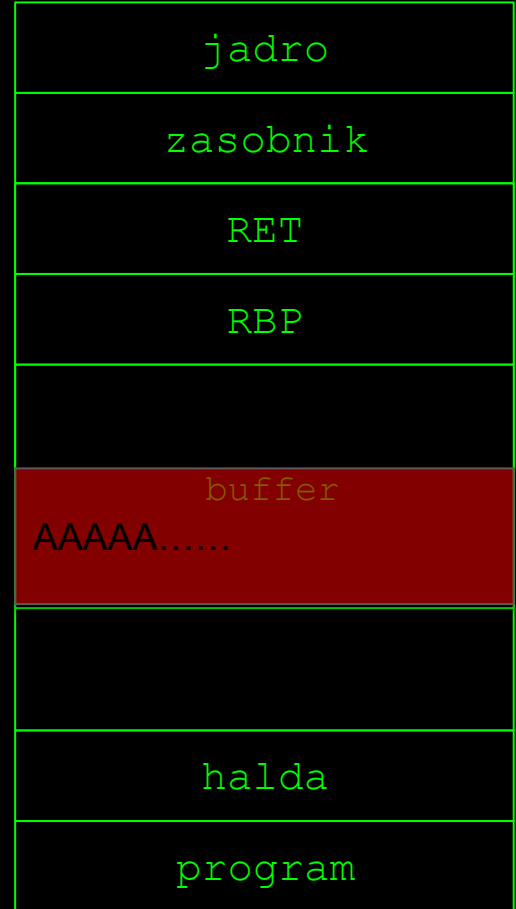


# >stack smashing

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

**Vstup:** 40 \* "A"

0xffff



0x0000

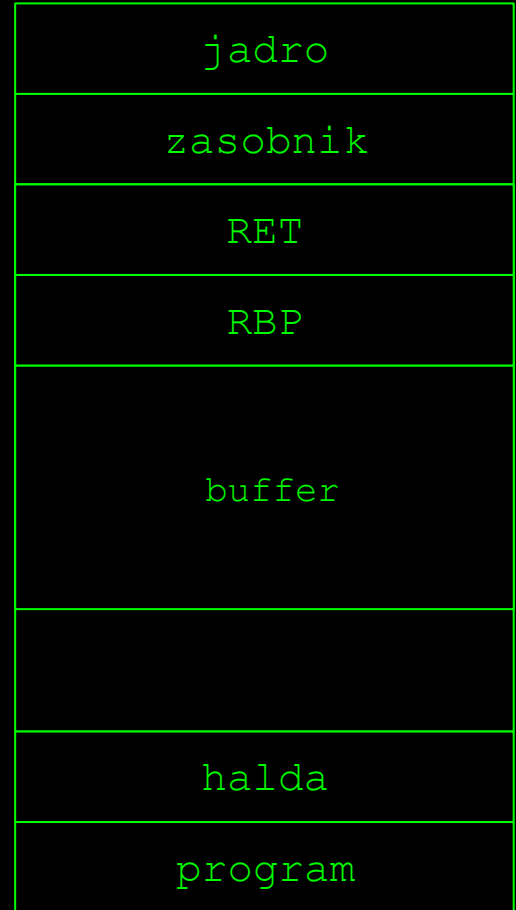
0x09

# >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



0x0000

0x0B

# >co mozeme prepisat?

>navratovu adresu

>lokálne premenne

>hodnota v pamäti, ktorá ovplyvňuje mat. operácie

>smerníky

NOOOOOOO!! YOU CAN'T JUST CREATE A PROGRAM THAT WRITES MORE DATA THAN CAN BE ALLOCATED IN A BLOCK OF MEMORY!!!! ATTACKERS CAN USE THIS AS AN EXPLOIT TO EXECUTE ARBITRARY CODE ON THE USER'S MACHINE

haha buffer go brrrrrrrrrr

name-parameter
return address
base pointer
buffer

↑ buffer fill direction

>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 opravnienia

>**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 (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

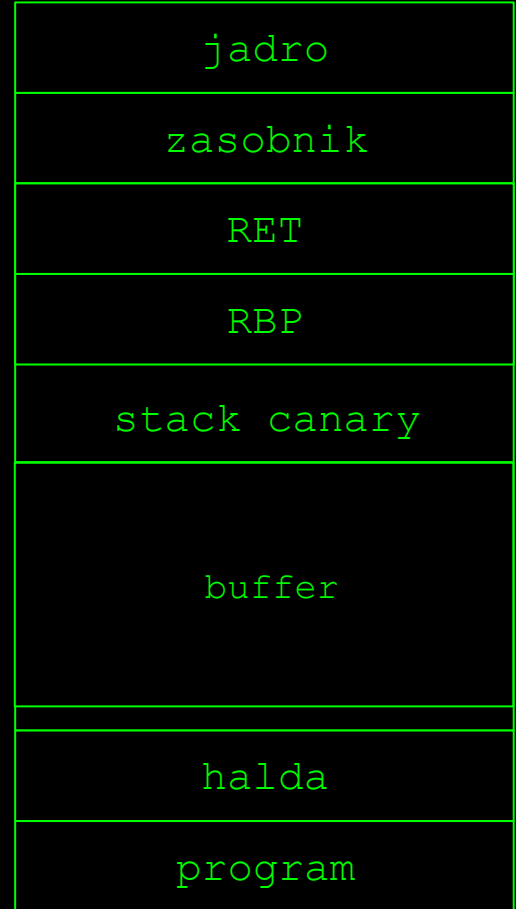


# >stack canary

```
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



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);
```



# >ako pisat exploity?

```
>PWNTOOLS1 2 (ipython)
```

```
import pwn
p = pwn.process('challenge')
addr = pwn.p64(0x40000)
p.sendline('366')
p.send(b'A'*360 + addr)
p.clean()

...
p.recv(7)
p.readuntil('test')
```

<sup>1</sup><https://docs.pwntools.com/en/stable/>

<sup>2</sup><https://github.com/Gallopsled/pwntools-tutorial#readme>

# >ako ladit exploity?

```
>p = pwn.gdb.debug('challenge')
```

```
>pri ladeni je potrebny terminalovy multiplexer
```

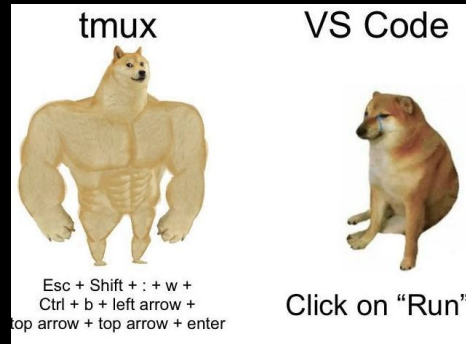
```
>tmux
```

```
>CTRL+b % (rozdelenie obrazovky horizontalne)
```

```
>CTRL+b ` (rozdelenie obrazovky vertikálne)
```

```
>CTRL+b x (zatvorenie okna)
```

```
>CTRL+b sipka (prepínanie medzi oknami)
```



## >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!)