Pamäťové zraniteľnosti

Bezpečnosť informačných systémov z pohľadu praxe

Peter Švec

0x01

>shellcode výsledky

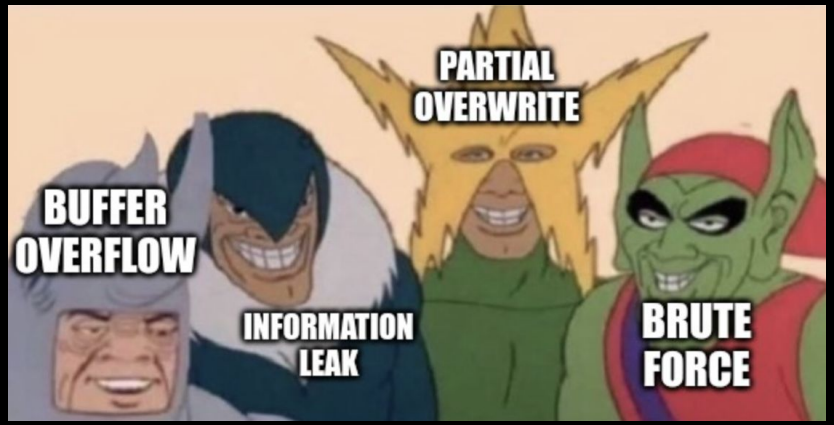
		_tím	_skóre
#1		skl	8
#2		Stormwind	8
#3		Lock-in	8
#4		NBU	8
#5		Colin Robinson Fanclub	8

celkový stav ->

	🥇	🥈	🥉
1. skl	1	1	0 (5)
2. MilujemBISPP	1	0	0 (3)
3. TvojTatkoRecords	1	0	0 (3)
4. Stormwind	0	1	1 (3)
5. Maet	0	1	0 (2)
6. Lock-in	0	0	1 (1)
7. Kruzidlo	0	0	1 (1)

>Úvod

>C je nízko úrovňový jazyk (verí vývojárovi)
>Čo ak vieme pomocou chyby v programe zapisovať dáta mimo alokovaný priestor?
>Typy pamäťových chýb v rámci zásobníka
>Moderné mitigácie (a techniky ako ich obísť)



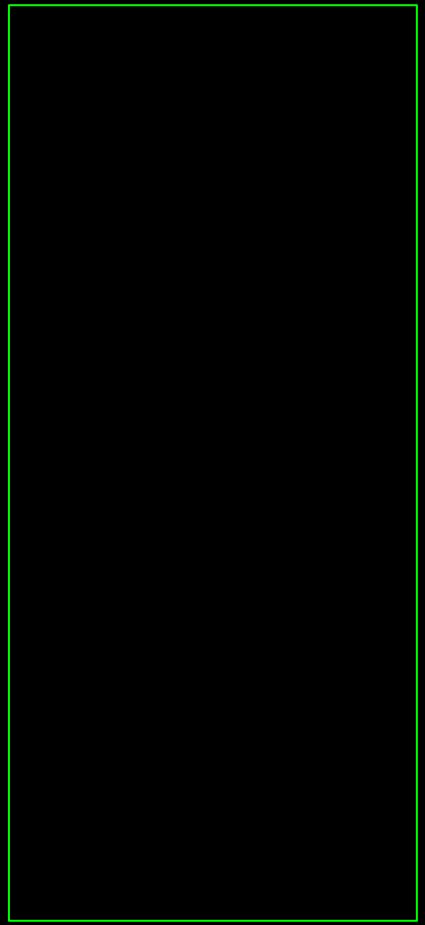
>Zásobník

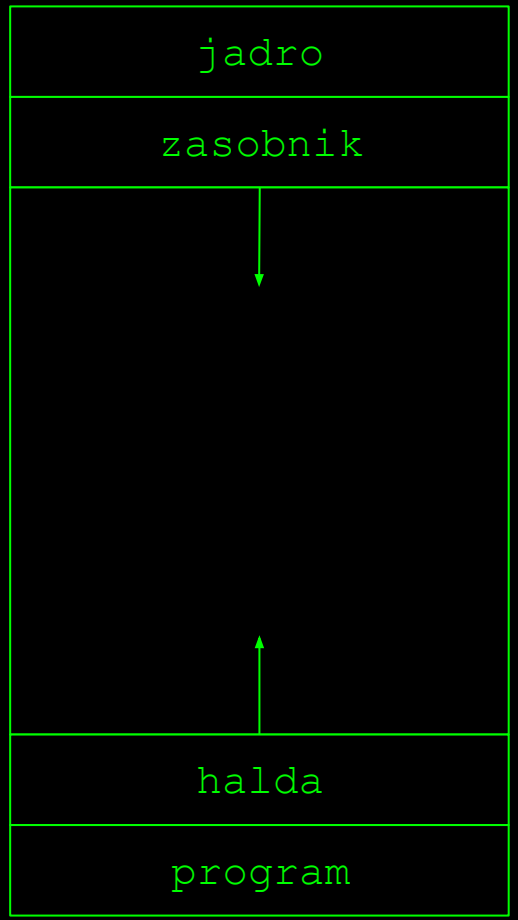
>LIFO dátová štruktúra používaná pri volaní funkcií (call stack)
>Na zásobnik sa ukladajú:
- >lokálne premenné
- >návratová adresa 
- >zásobníkový rámec z predchádzajúceho volania
- >pri vysokom množstve argumentov aj argumenty

ZÁSOBNÍK RASTIE OPAČNÝM SMEROM
OD VYSOKÝCH ADRIES(0xFFFF…) PO NIŽŠIE (0x000…)

>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```





0xffff

0x0000

0x05

>Zásobník

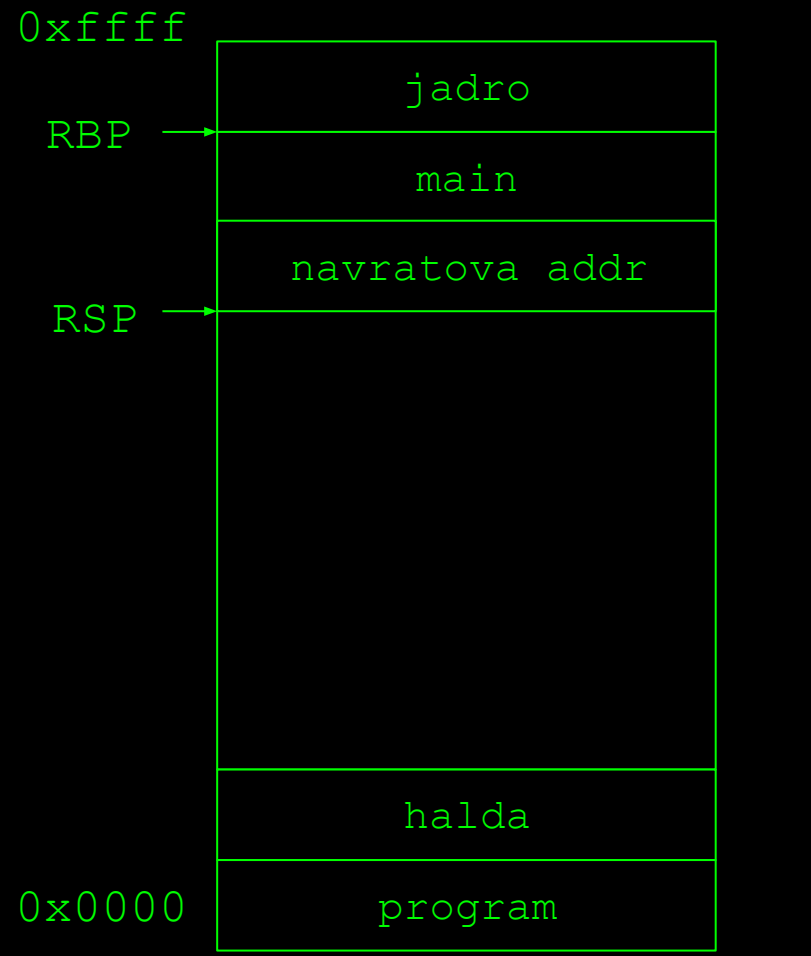
```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

```
call foo
```

>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

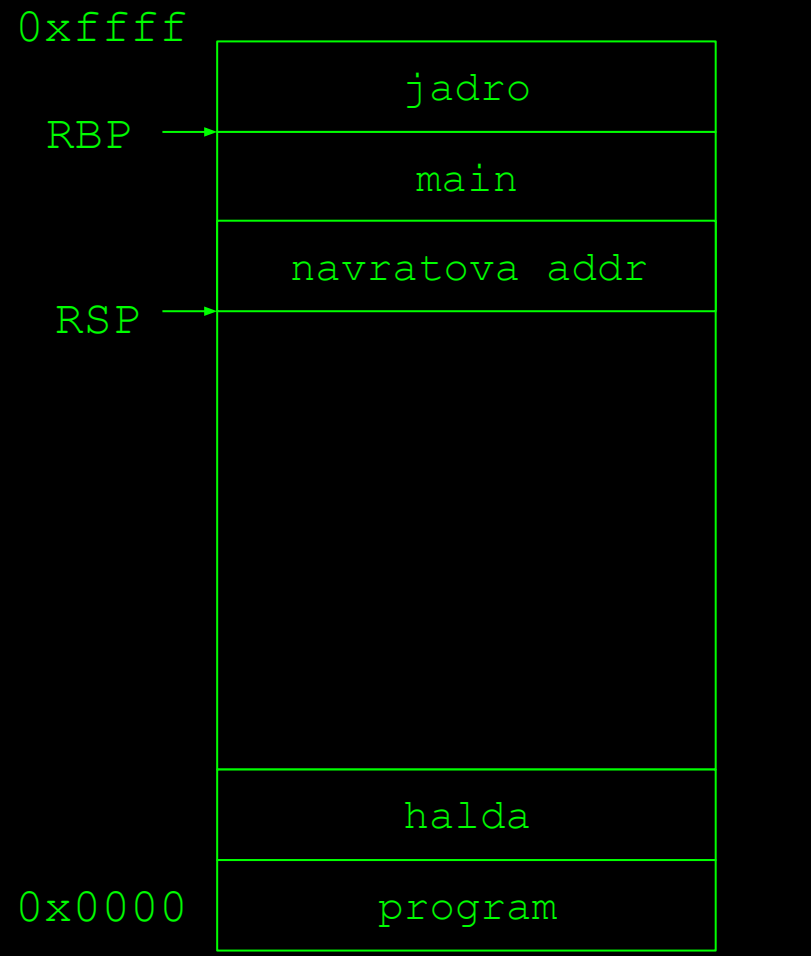
```
call foo
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

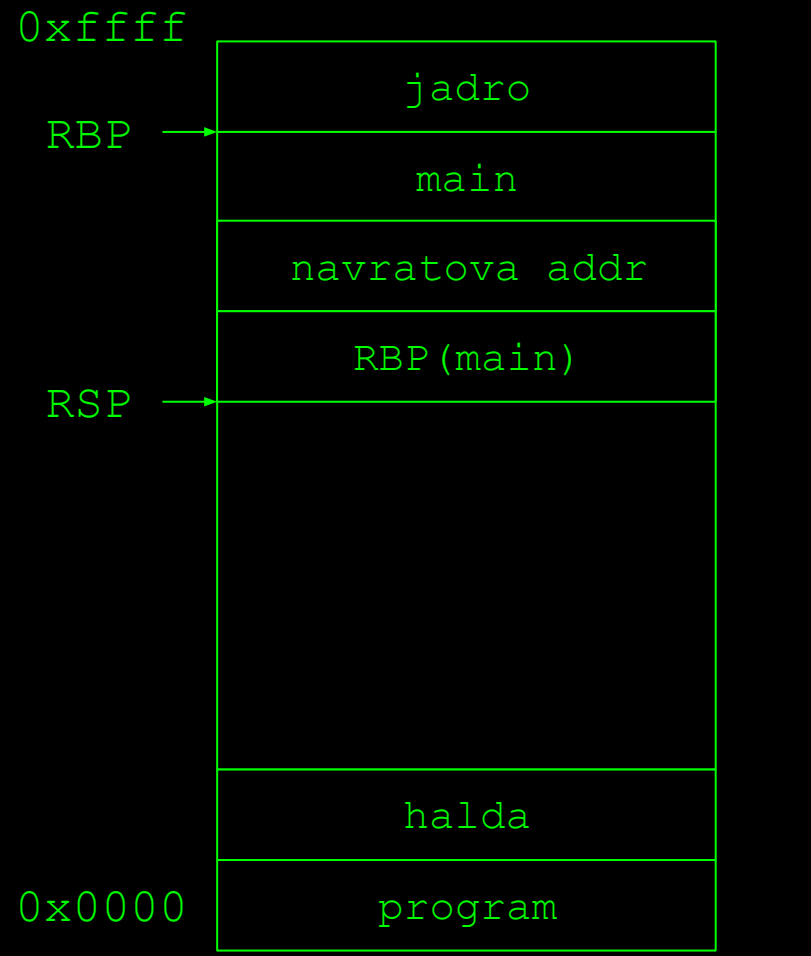
```
call foo

push rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

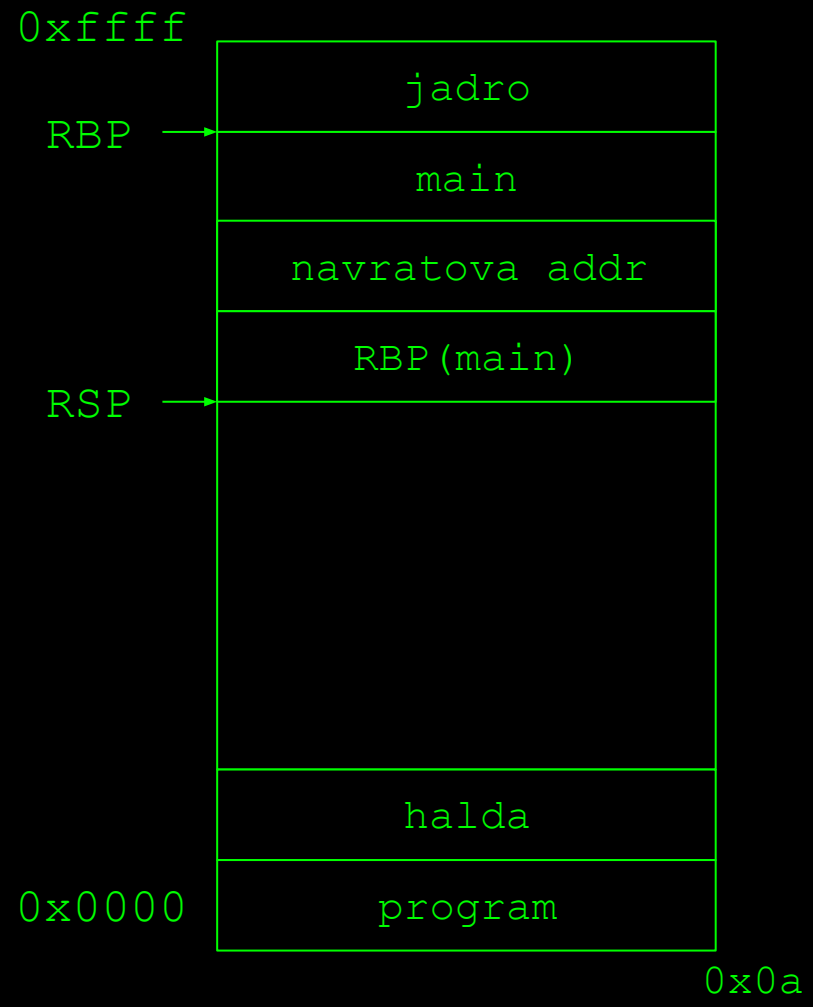
```
call foo

push rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

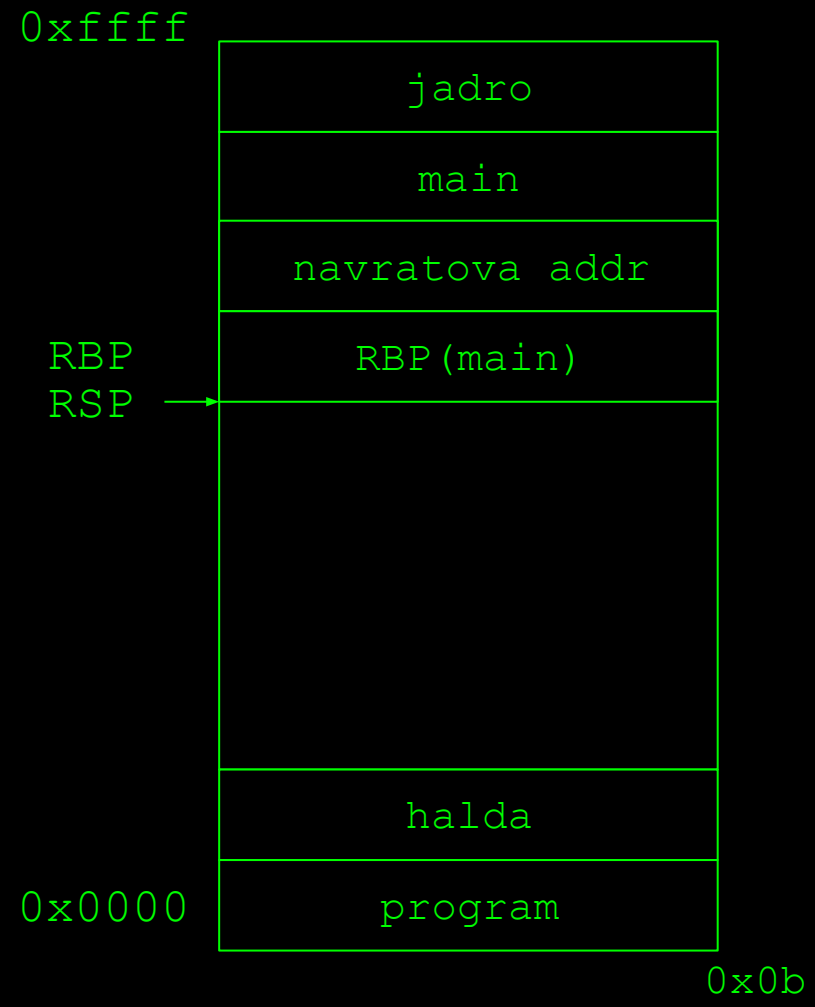
```
call foo

push rbp
mov rbp, rsp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

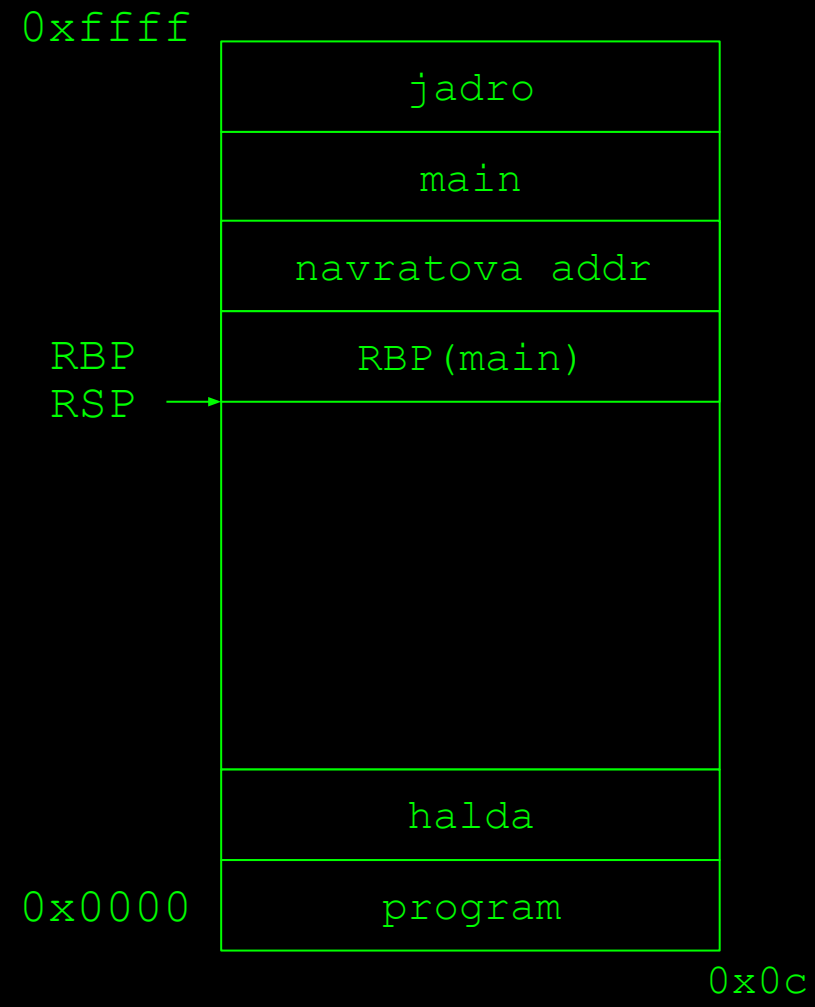
```
call foo

push rbp
mov rbp, rsp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

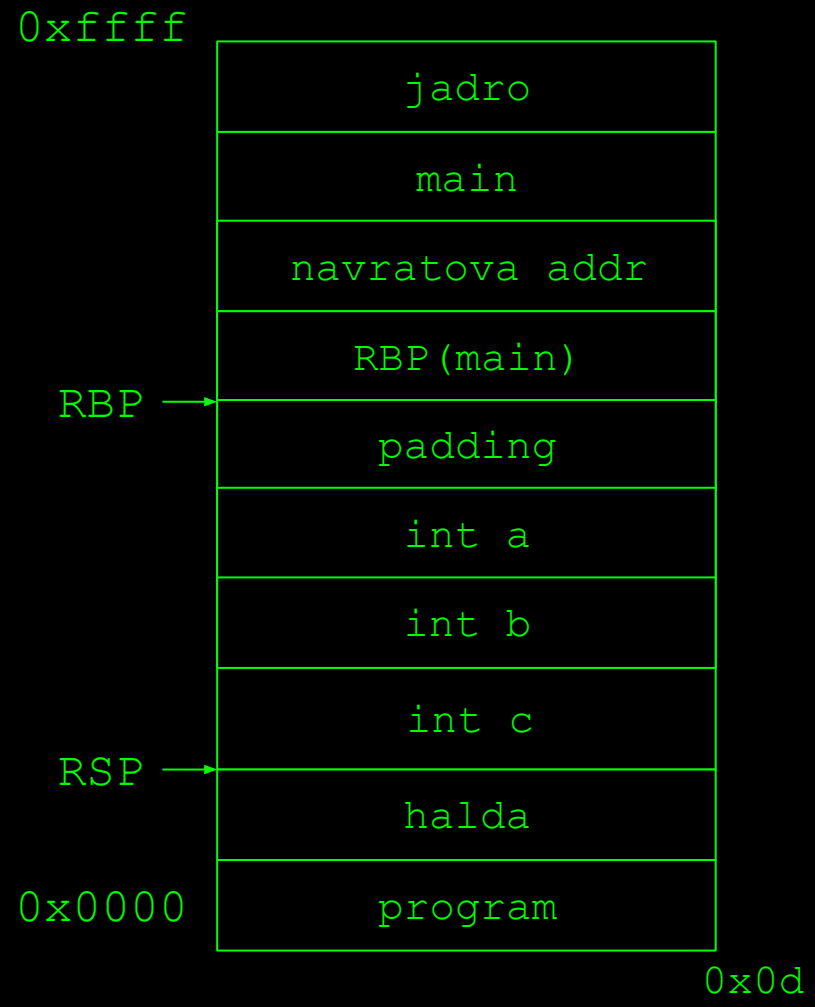
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

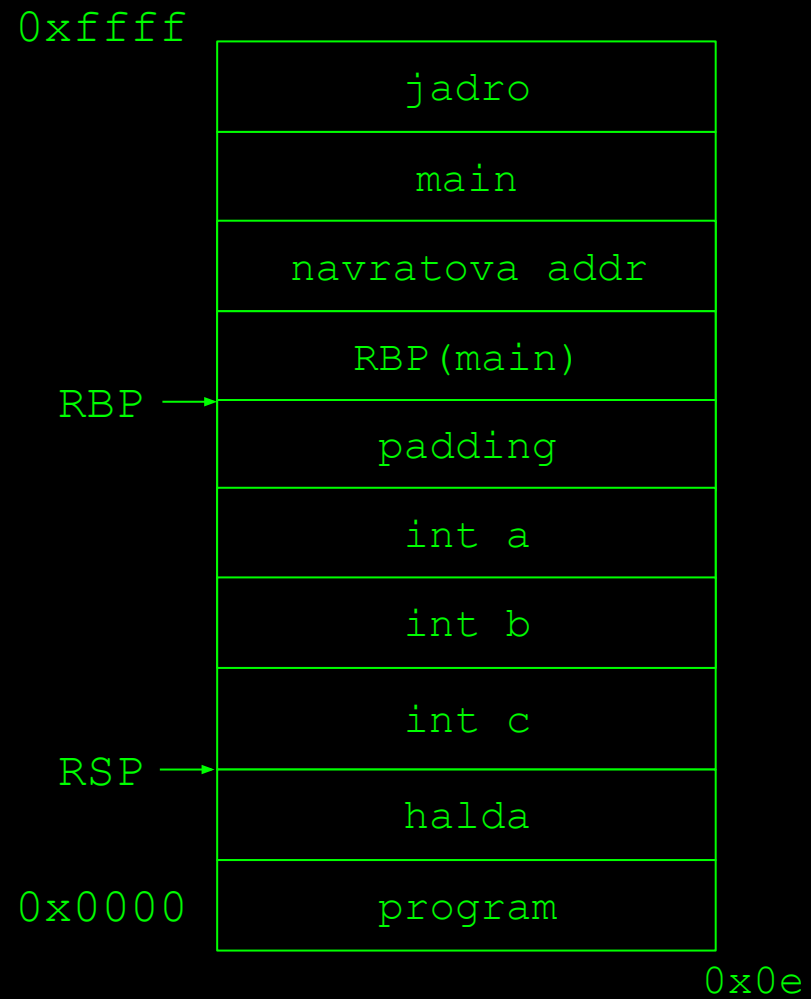
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

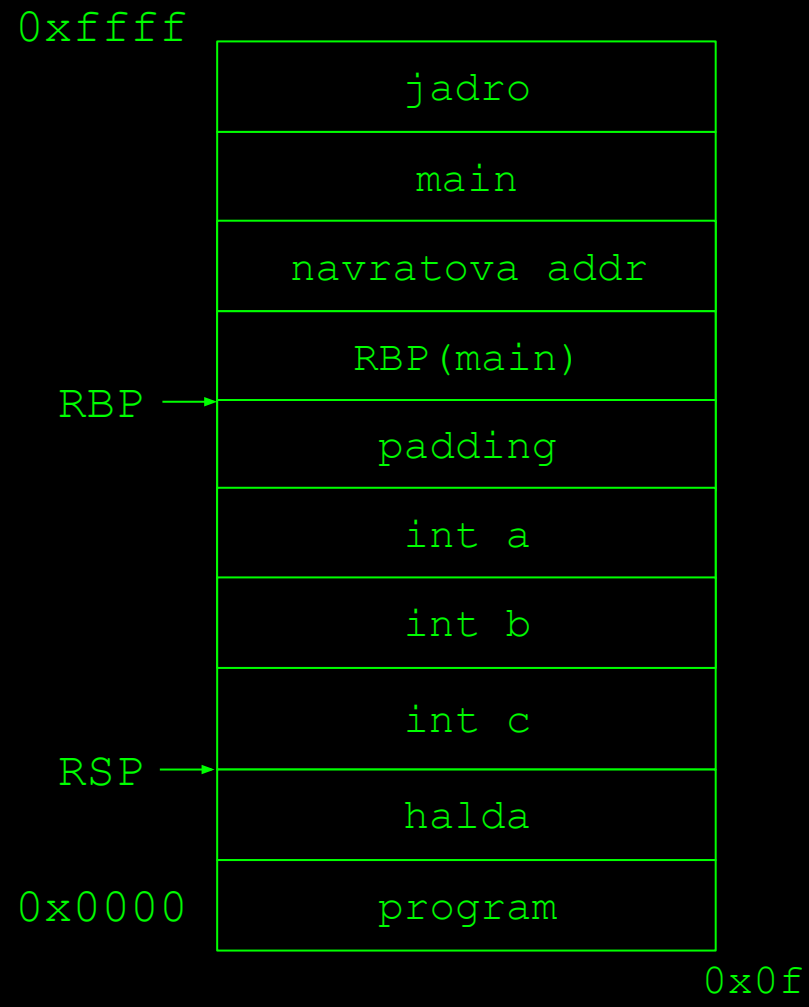
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

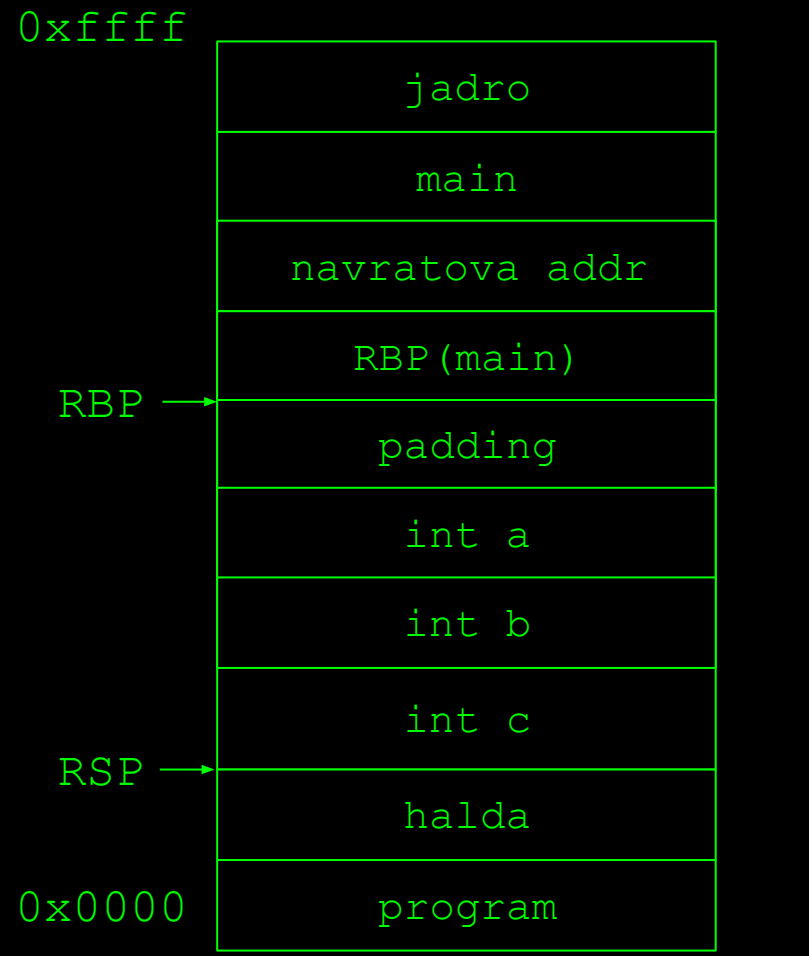
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

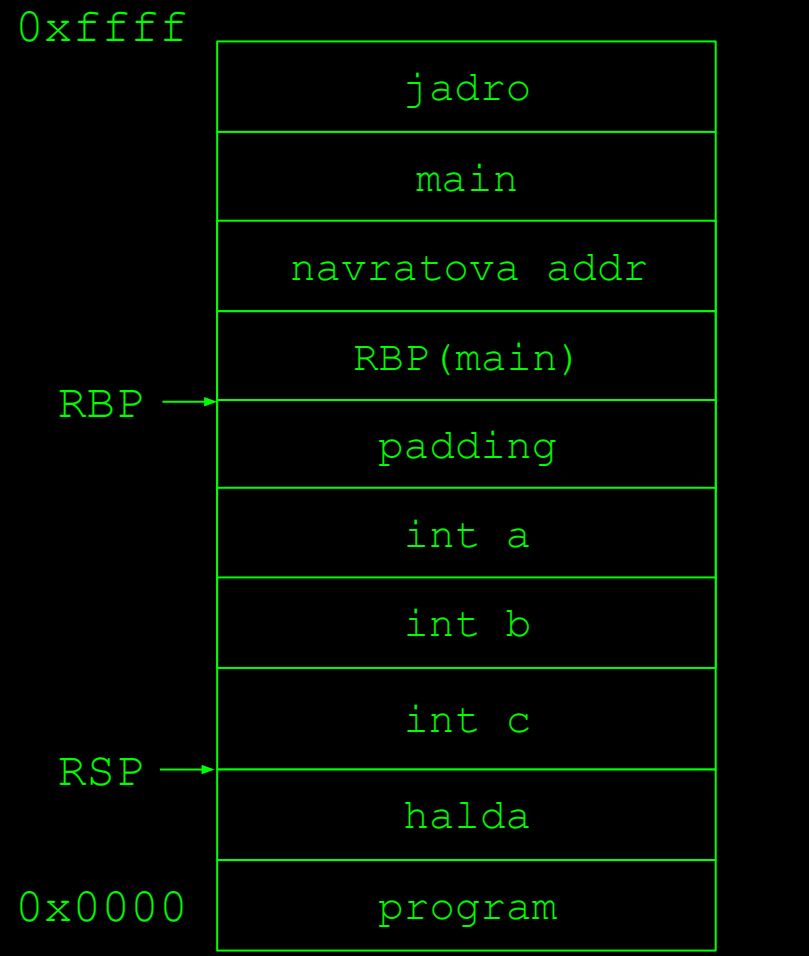
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

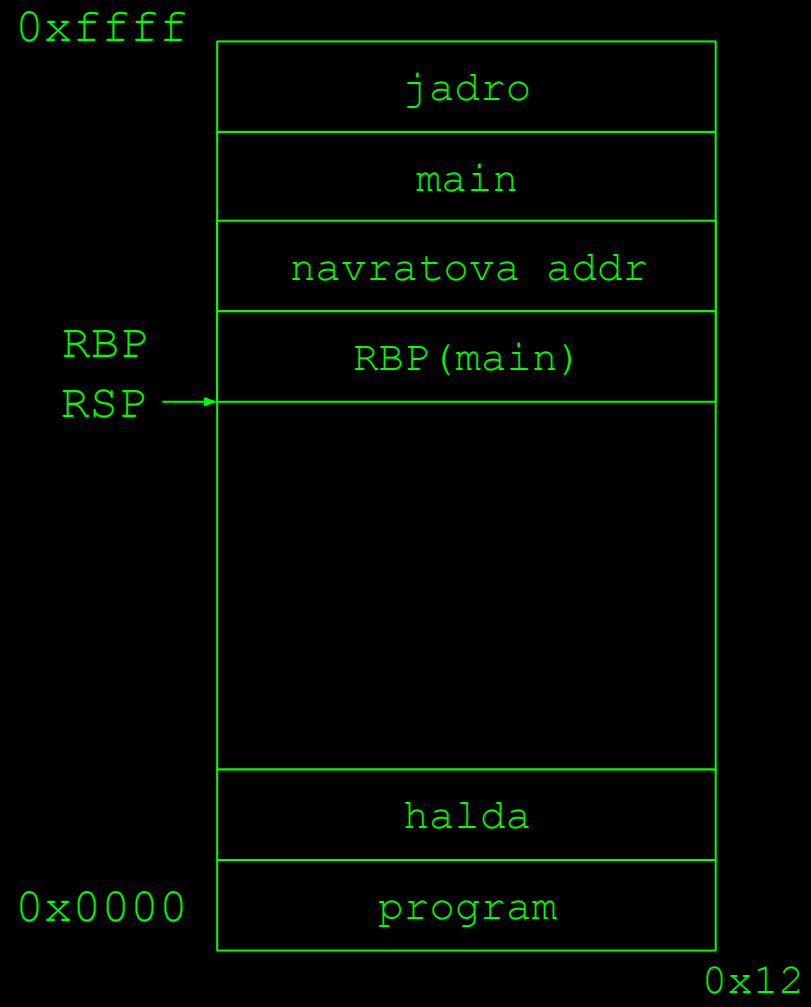
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

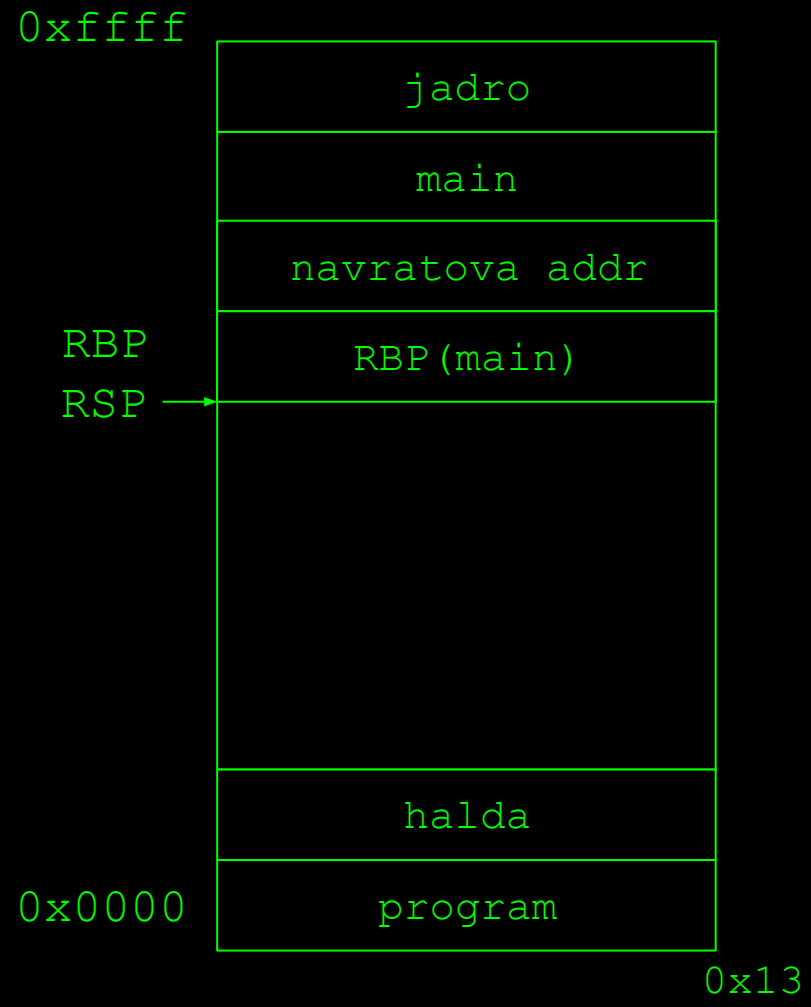
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

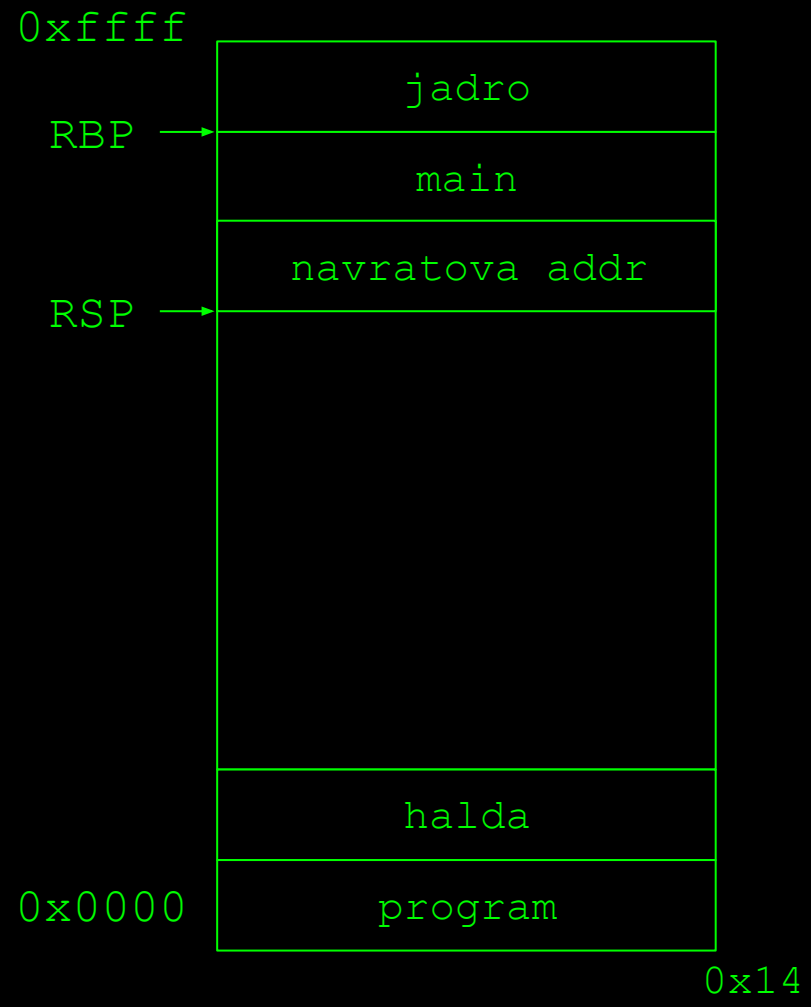
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
pop rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

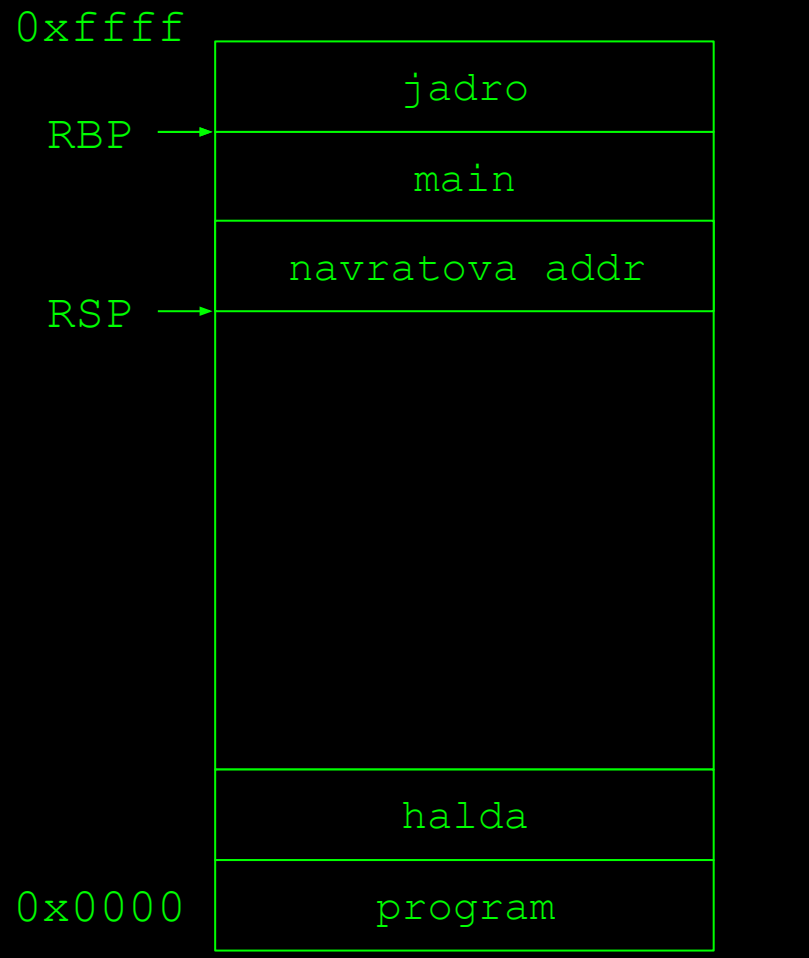
```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
pop rbp
```



>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
pop rbp
ret
```

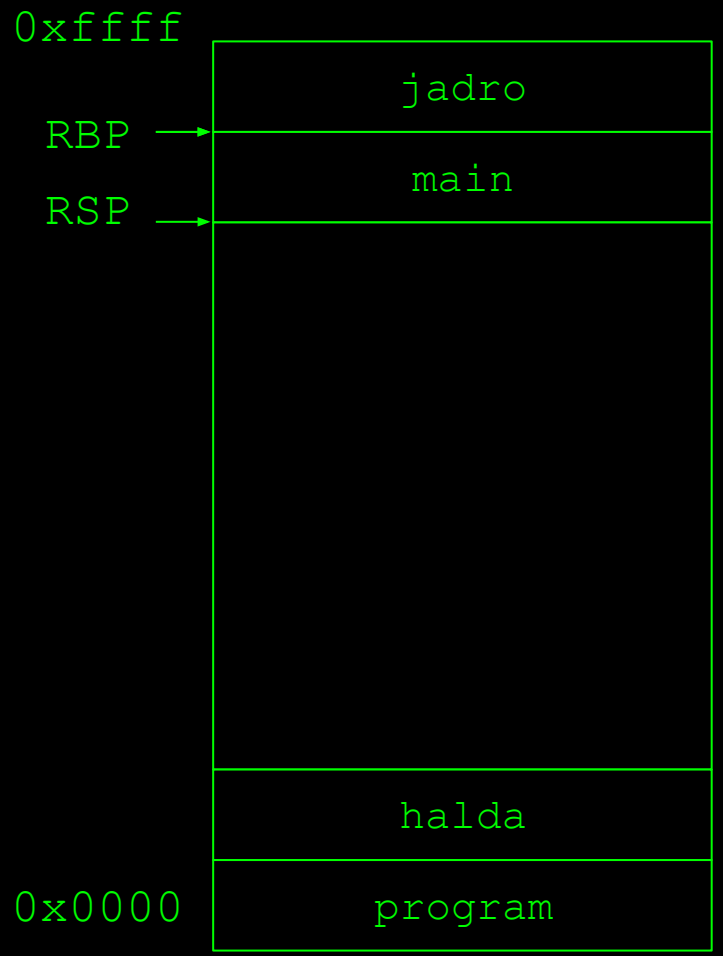


ret = načíta do RIP vrch zásobníka

>Zásobník

```
int foo()
{
    int a,b,c;
    b=1;c=2;
    a=b+c;
    return a;
}

int main()
{
    foo();
    return 0;
}
```

```
call foo

push rbp
mov rbp, rsp
sub rsp,0x10
mov[rbp-8],1
mov[rbp-12],2

vypocty…

mov rax,[rbp-4]
mov rsp, rbp
pop rbp
ret
```



ret = načíta do RIP vrch zásobníka

>Problém 1

>Dôvera vývojárovi

python:

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

```
IndexError: list index out of range
```

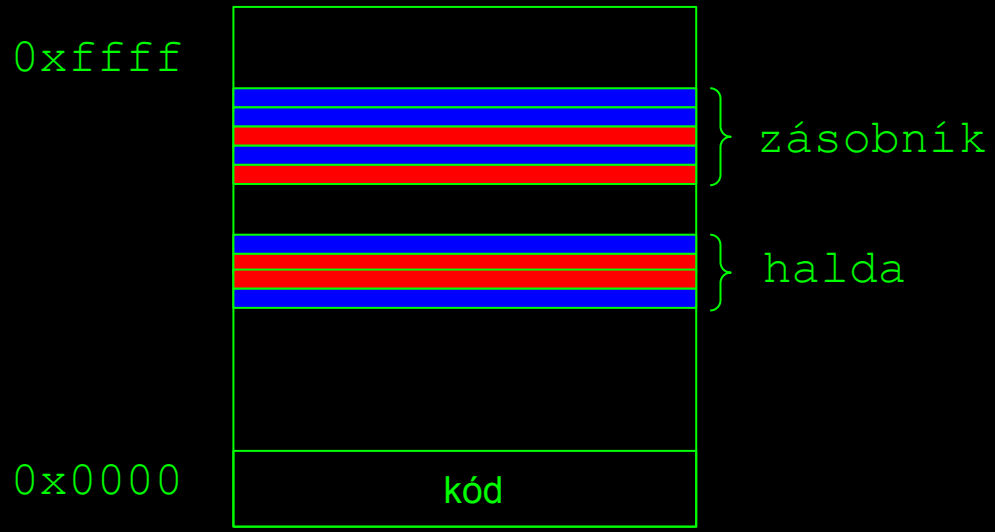
C:

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

Všetko ok!

>Problém 2

>Mixovanie dát (aj používateľský vstup) a control flow údajov (návratové adresy, ptrs na funkcie)



>Problém 3

>Mixovanie dát a metadát

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) = ?

>Problém 4

>Inicializácia a dealokácia zásobníka
>Obe operácie sú 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”

>stack smashing

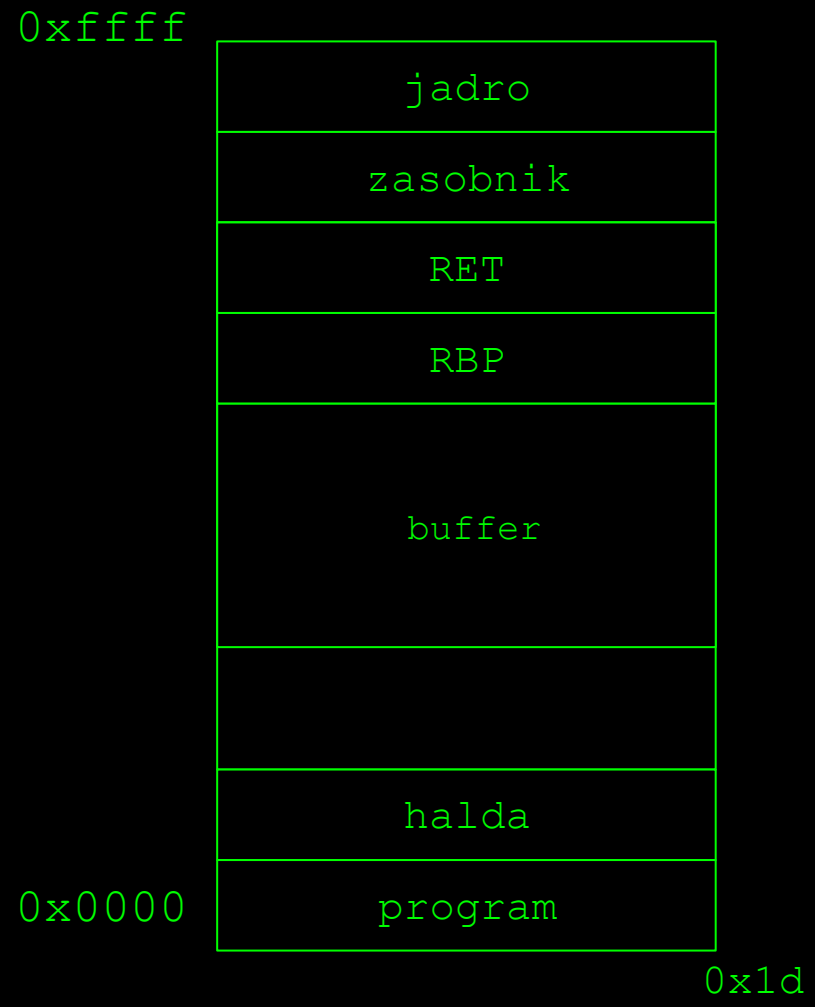
```
void vuln()
{
    char buffer[80];
    gets(buffer)
}
```

Vstup: 40 * “A”

>stack smashing

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

Vstup: 96 * “A”

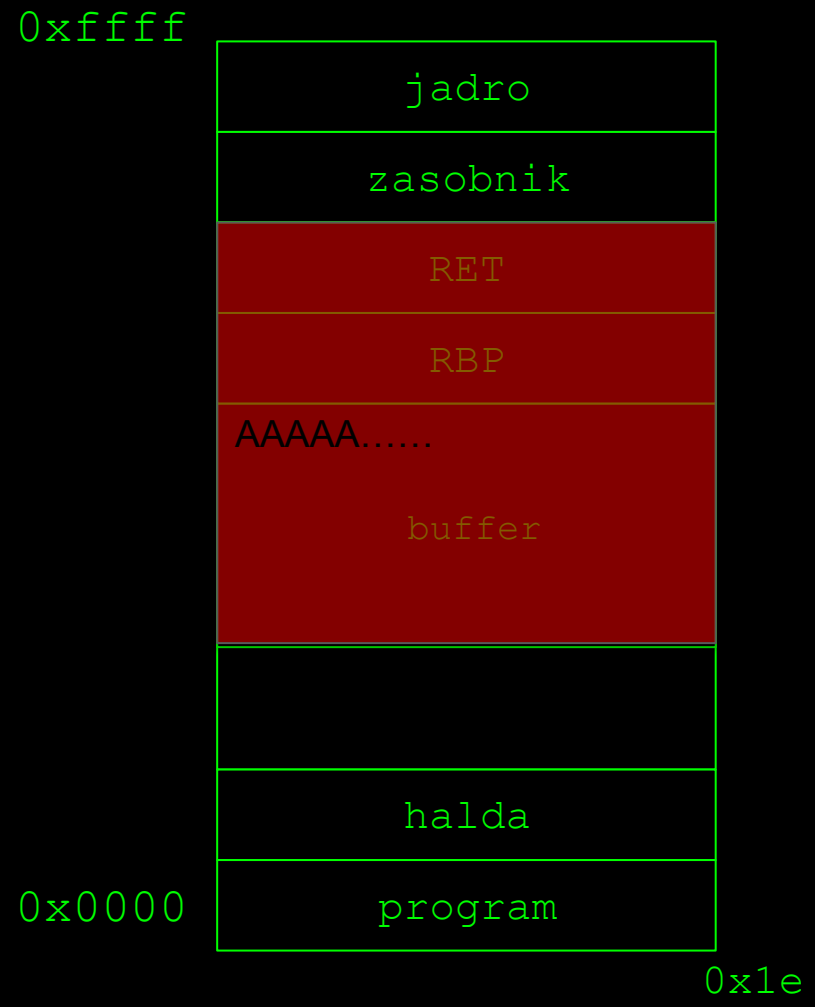


>stack smashing

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

Vstup: 96 * “A”

Segmentation fault
Invalid address 0x414141414141



>Čo môžeme prepísať?

>návratovú adresu
>lokálne premenné
>hodnota v pamäti, ktorá ovplyvňuje mat. operácie
>smerníky

>Buffer overflow

>chyby pri počítaní veľkosti buffra
>nebezpečné funkcie (gets, strcpy,...)

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

>Signed/unsigned mismatch

>štandardná knižnica (libc) používa na definovanie veľkosti typ **unsigned**

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

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

>Integer overflow

>čo sa stane ak k max. hodnote 32 bitového integeru (0xffffffff) pripočítam 1?

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

>Mitigácie

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

>ASLR

>Znáhodňovanie adries (offsety však ostávajú rovnaké!)
>Plná sila **ASLR** je iba v kombinácii so zapnutým **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!

>Čo s tým?

>**information leak**

>ak vieme z programu prečítať ľubovoľnú adresu, tak sme vyhrali. Offsety ostavajú rovnaké a tym pádom vieme vypočítať base adresu

>**partial overwrite**

>program má stránky zarovnané na 4096 bajtov (0x1000)

>spodných 12 bitov ostáva -> 2^4 brute force

1.spustenie: 0x5571aef0b**1d4**

2.spustenie: 0x55b4a56b6**1d4**

3.spustenie: 0x562b131e3**1d4**

>**brute force**

>na 64 bitových systémoch nepoužiteľné (2^{32} BF)

>DEP

>Stránky v pamäti môžu mať rôzne oprávnenia
- >**R**ead (**R**)
- >**W**rite (**W**)
- >**E**xecute (**X**)

>Pri zapnutom NX, nemôže mať heap/stack **X** flag, (problematické, ak chceme spustiť shellcode)
>Zapnuté DEP je v dnešnej dobe absolútny štandard
>Princíp ako to obísť je relatívne komplexný (ROP)

>Stack canary

>Náhodná 8 bajtová hodnota, ktorá chráni návratovú adresu
>Pred návratom z funkcie sa skontroluje jej integrita
>56 náhodnych bitov (jedna hodnota pre všetky funkcie)
>Mení sa pri každom spustení programu

Príklady hodnôt:

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

>Stack canary

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

Vstup: 96 * “A”

>Stack canary

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

Vstup: 96 * “A”

***** stack smashing detected *****
terminated

>Čo s tým?

>**information leak**
  >podobne ako pri ASLR, ak vieme prečítať hodnotu kanárika, tak ju vieme pripojiť k payloadu
>**brute force**
  >robiť 2^{56} nemá zmysel
  >pri tzv. fork-servis programoch (napr. spracovanie sieťových spojení je robené cez nový proces)
  >pri forknutí procesu sa hodnota kanárika nemení
  >v takom prípade vieme aplikovať inteligentný brute force
  >postupné prepisovanie kanárika po bajtoch
  >max. 7 * 255 pokusov

>Information leak

>buffer overread:

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

>zabudnutý NULL byte

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

>Ako písať exploity?

>PWNTOOLS[1] [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')
```

[1] https://docs.pwntools.com/en/stable/

[2] https://github.com/Gallopsled/pwntools-tutorial#readme

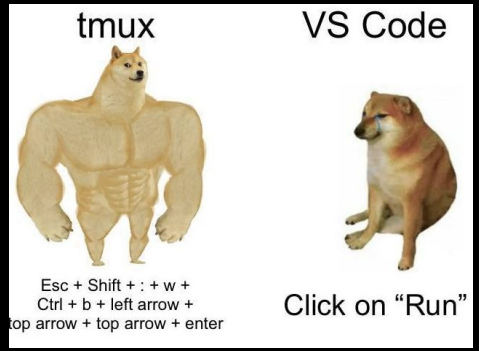
>Ako ladiť exploity?

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

>echo source /opt/pip_pwndbg/gdbinit.py > ~/.gdbinit

>pri ladení je potrebný terminálový multiplexer

- >tmux
- >CTRL+b % (rozdelenie obrazovky horizontálne)
- >CTRL+b " (rozdelenie obrazovky vertikálne)
- >CTRL+b x (zatvorenie okna)
- >CTRL+b sipka (prepínanie medzi oknami)



>Postup

>Spustím binárku, vyskúšam ako funguje
>Skontrolujem mitigácie (checksec)
>Zreverzujem binárku (toto uz viete)
>Identifikujem ako využiť zraniteľnosť (samotná zraniteľnosť je väčšinou nami kontrolovaný overflow)
>Napíšem exploit (pwntools!)
>Ak niečo nejde tak debugujem (gdb!)