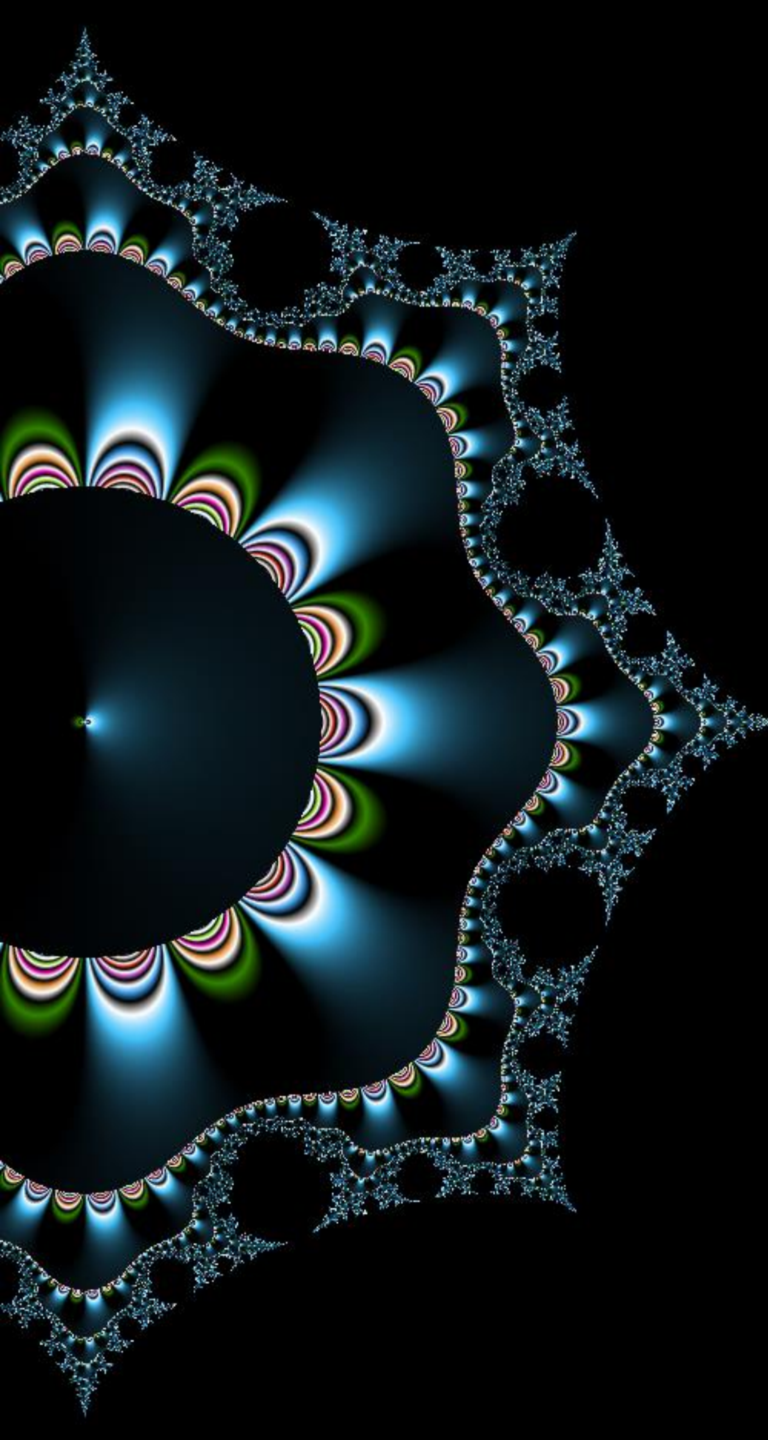
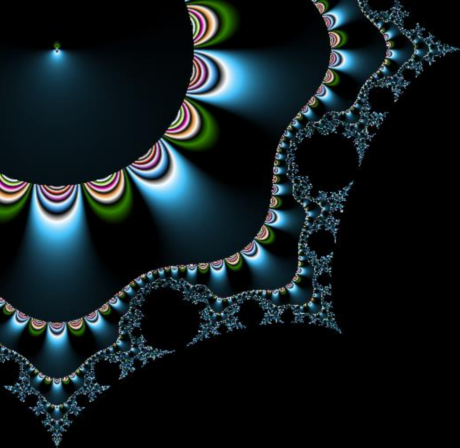


Memory Management

Software vulnerabilities

Corinne HENIN
www.arsouyes.org



Summary

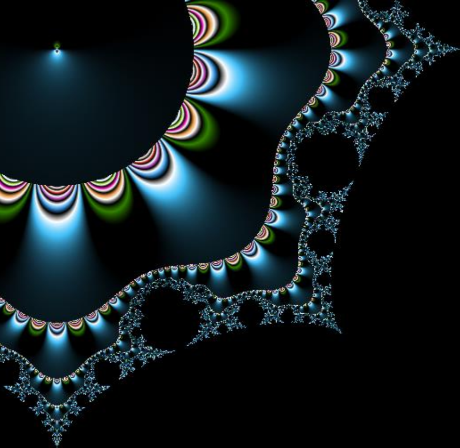
What we are going to see

Use after free

Double Free

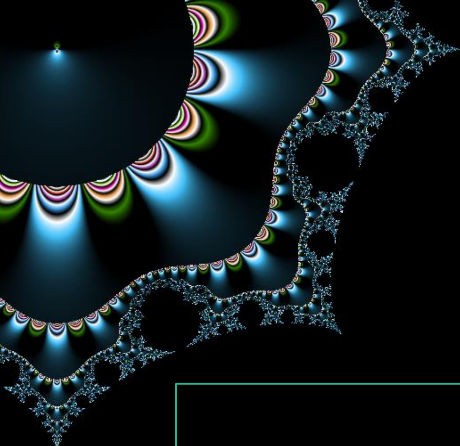
Null pointer dereference

Uninitilize local variable



Use after free

Use what shouldn’ t be

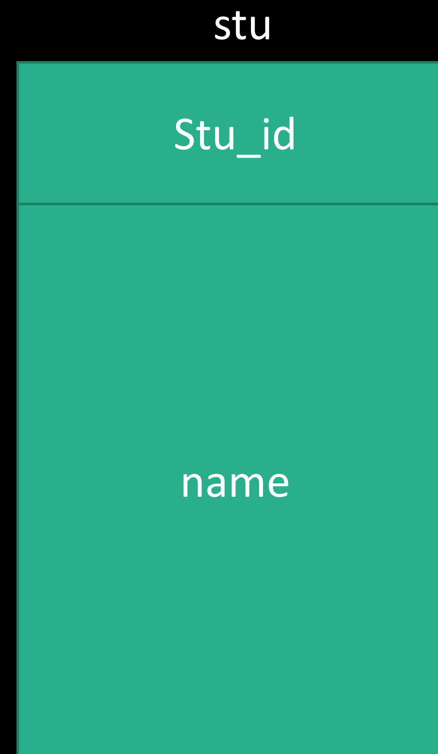


Before starting

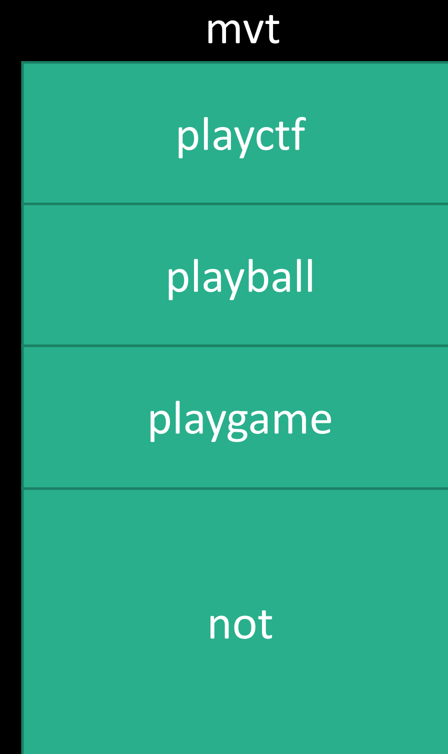
A small exemple

```
typedef struct stu {
        int stu_id ;
        char name[20] ;
} stu_t ;

typedef struct mvt {
        void (* playctf)() ;
        void (* playball)() ;
        void (* playgame)() ;
        char not[12] ;
} mvt_t ;
```

stu



mvt



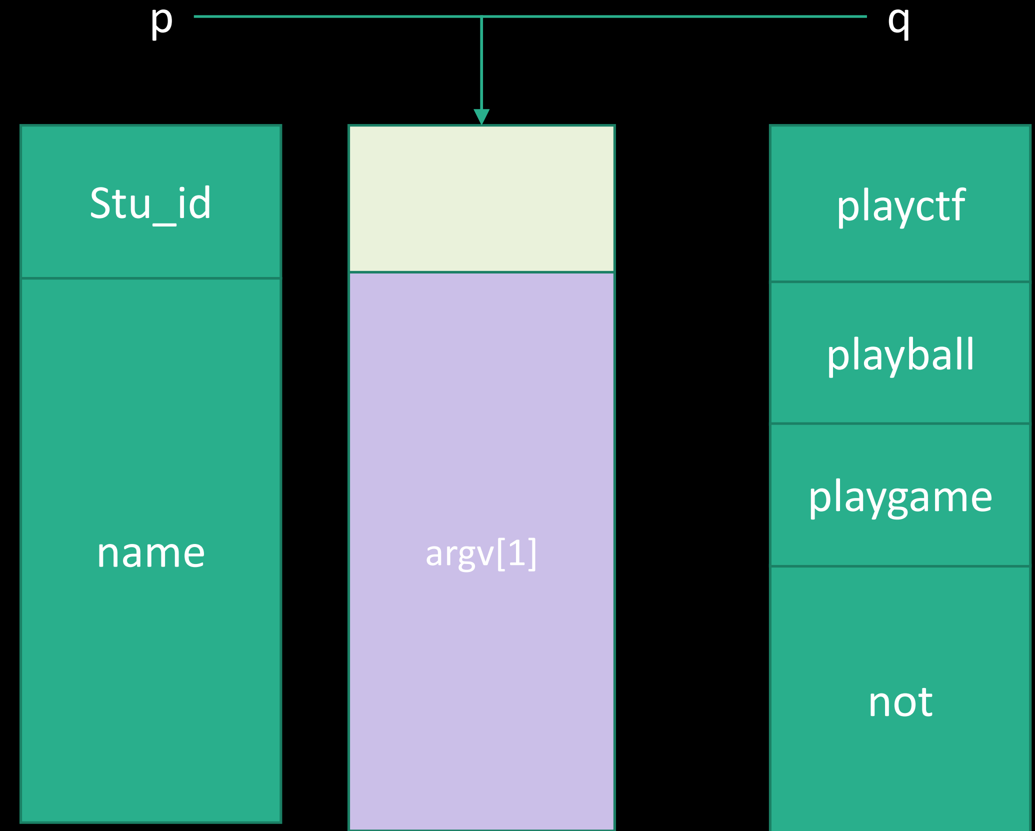
Problematic code

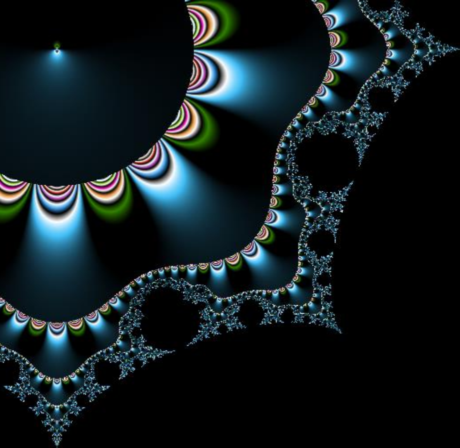
use a variable after freeing it

```
int main(int argc, char ** argv) {

  stu_t * p = (stu_t *) malloc(sizeof(stu_t)) ;
  free(p) ;
  mvt_q * q = (mvt_t *) malloc(sizeof(mvt_t)) ;
  initMvt(q) ;

  strncpy(p->name, argv[1], 20) ;

  q->playball() ;

  free(q) ;
}
```

Memory view

```
int main(int argc, char ** argv) {

  stu_t * p = (stu_t *) malloc(sizeof(stu_t)) ;
  free(p) ;
  mvt_q * q = (mvt_t *) malloc(sizeof(mvt_t)) ;
  initMvt(q) ;

  strncpy(p->name, argv[1], 20) ;

  q->playball() ;

  free(q) ;
}
```





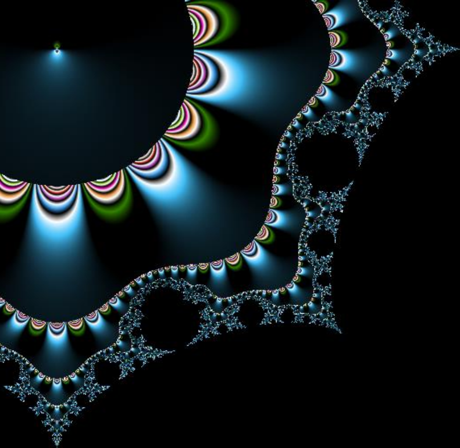
Clean code

How to avoir the problem

Use oriented object language

Good Design

RAII Pattern (init in constructor, free in destructor)

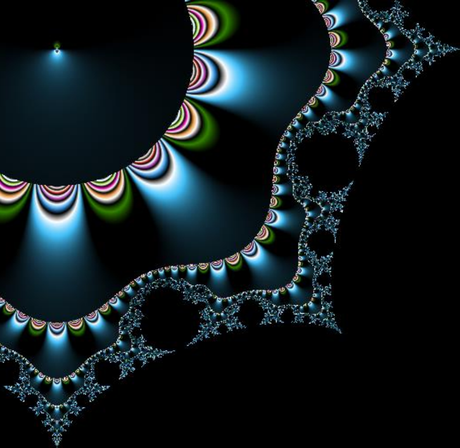


Defense in depth

a posteriori

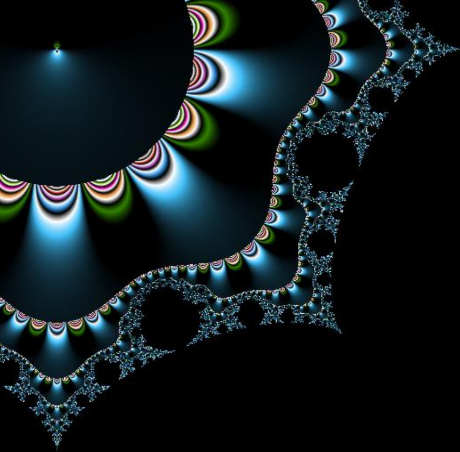
Dynamic code review

Valgrind



Double free

rather once than twice



FastBin Structure

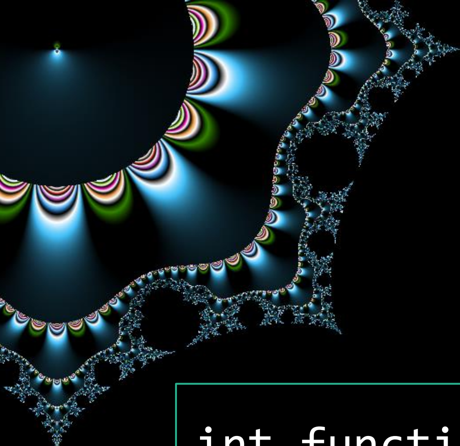
keep the chunks available

Linked list

chunks of a same size

Peculiarity

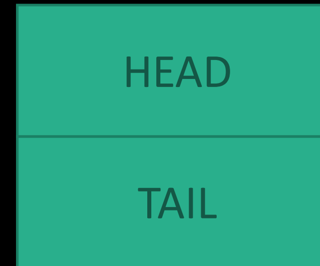
Chunks are not merged



FastBin Structure

On a small exemple

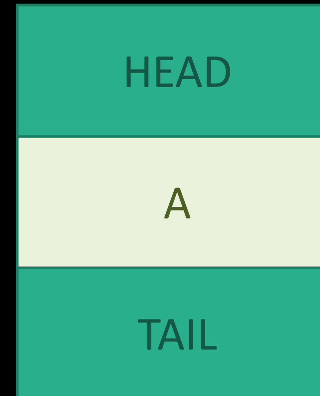
```
int function() {
  stu_t *a, *b, *c ;
  int res = 0 ;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  /*
   * Do stuffs
   */

  free(a) ;  free(b) ; free(c) ;
  return res ;
}
```

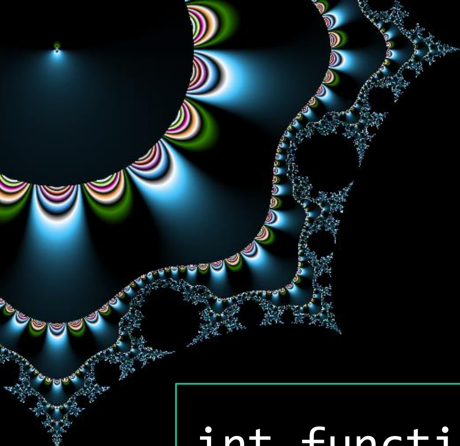


FastBin Structure

keep the chunks available

```
int function() {
  stu_t *a, *b, *c ;
  int res = 0 ;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  /*
   * Do stuffs
   */

  free(a) ;  free(b) ; free(c) ;
  return res ;
}
```

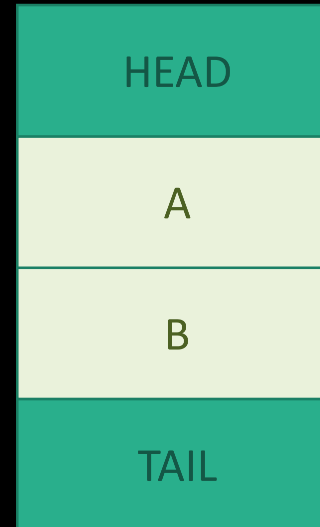


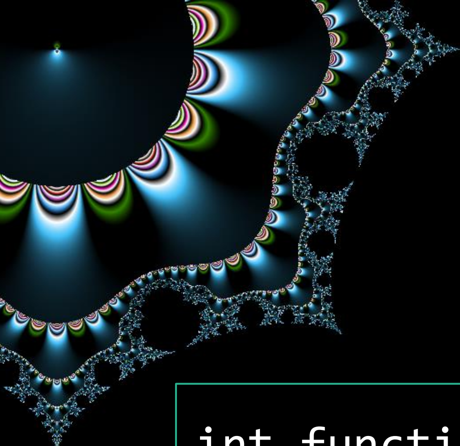


FastBin Structure

keep the chunks available

```
int function() {
  stu_t *a, *b, *c ;
  int res = 0 ;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  /*
   * Do stuffs
   */

  free(a) ;  free(b) ; free(c) ;
  return res ;
}
```

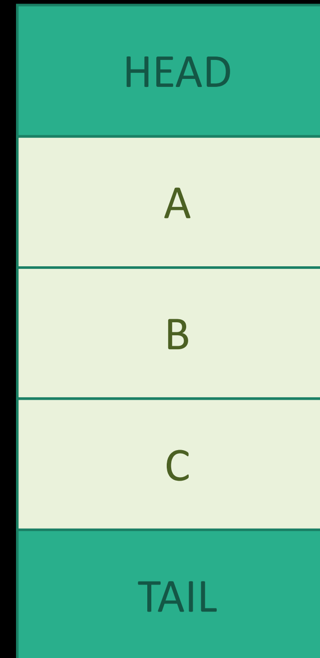


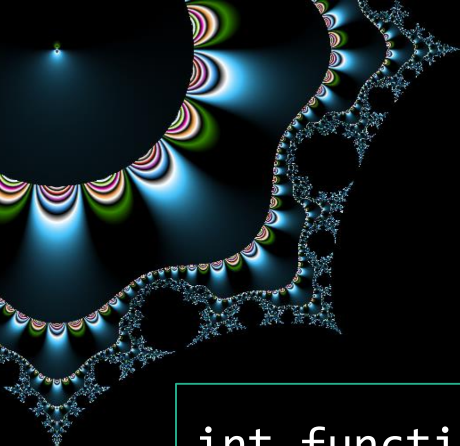


FastBin Structure

keep the chunks available

```
int function() {
  stu_t *a, *b, *c ;
  int res = 0 ;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  /*
   * Do stuffs
   */

  free(a) ;  free(b) ; free(c) ;
  return res ;
}
```

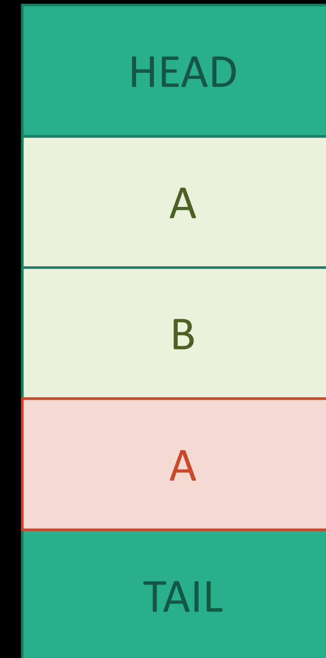


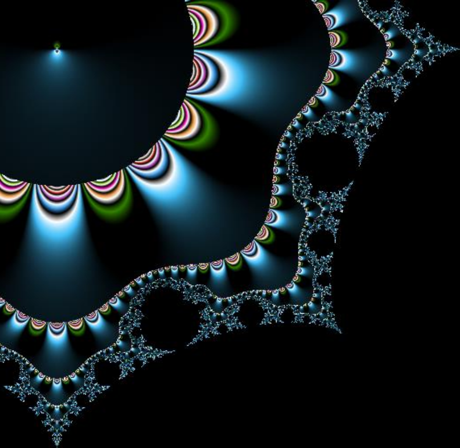


FastBin Structure

keep the chunks available

```
int function() {
  stu_t *a, *b, *c ;
  int res = 0 ;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  /*
   * Do stuffs
   */

  free(a) ;  free(b) ; free(a) ;
  return res ;
}
```





FastBin Structure

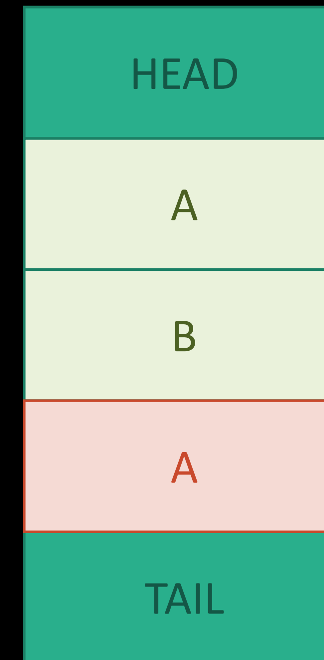
keep the chunks available

Then later…

Recycling

same size chunks are requested

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```



Recycling

same size chunks are requested

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```

A

HEAD

B

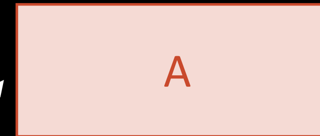
A

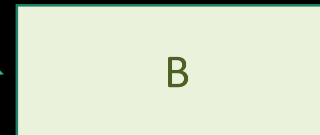
TAIL

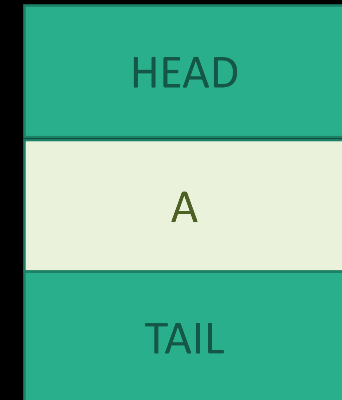
Recycling

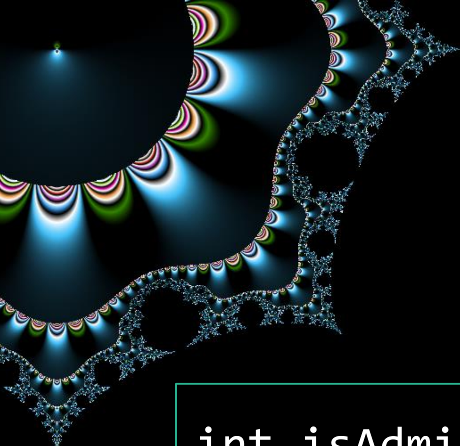
same size chunks are requested

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```





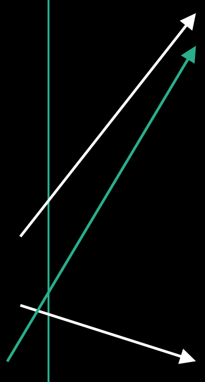


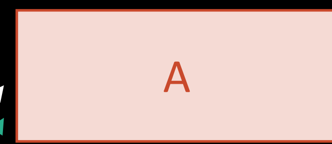


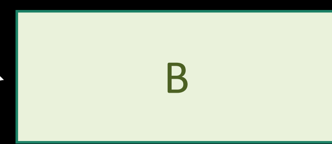
FastBin Structure

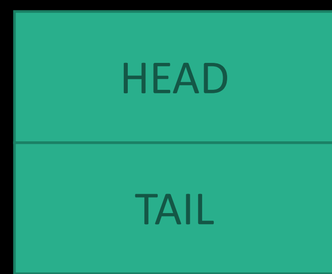
keep the chunks available

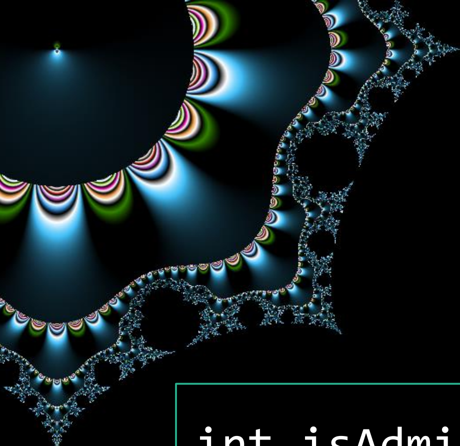
```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```







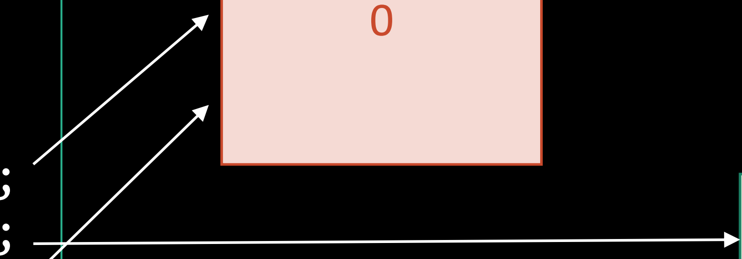


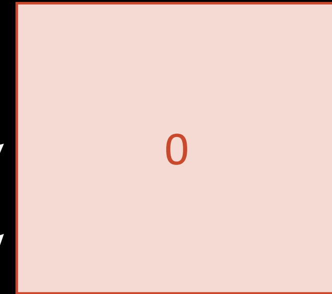


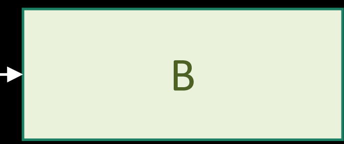
FastBin Structure

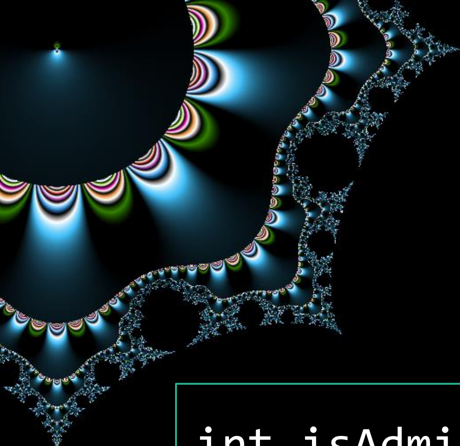
keep the chunks available

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```





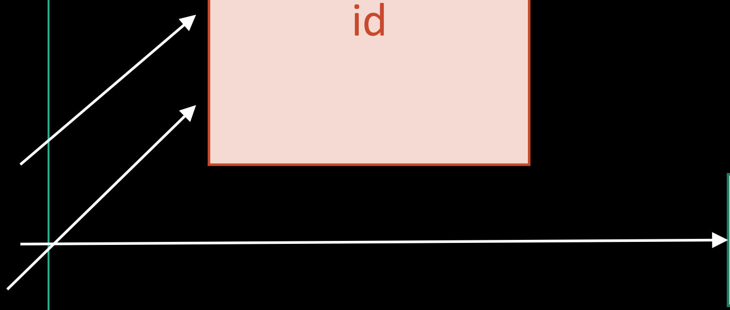


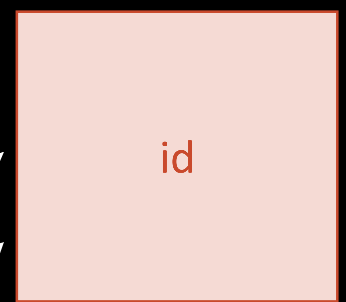


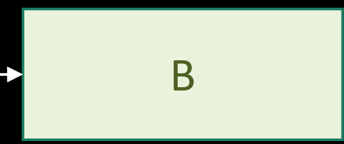
FastBin Structure

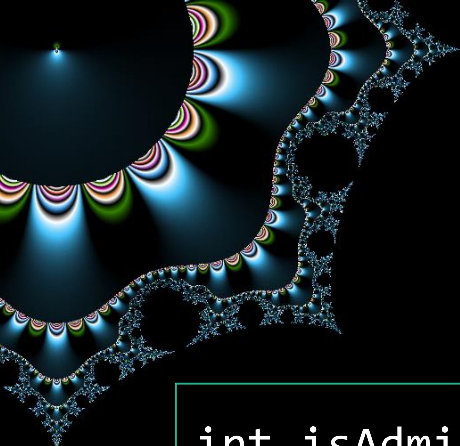
keep the chunks available

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```









FastBin Structure

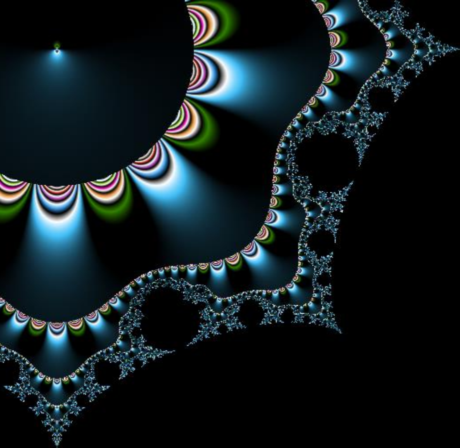
keep the chunks available

```
int isAdmin(int id) {
  stu_t *a, *b, *c ;
  int res = 0;

  a = (stu_t *) malloc(sizeof(stu_t)) ;
  b = (stu_t *) malloc(sizeof(stu_t)) ;
  c = (stu_t *) malloc(sizeof(stu_t)) ;

  a->id = 0 ;
  c->id = id ;
  res = a->id == c->id ;

  free(a) ; free(b) ; free(c) ;
  return res ;
}
```

Obviously

a->id IS c->id

Will always return True



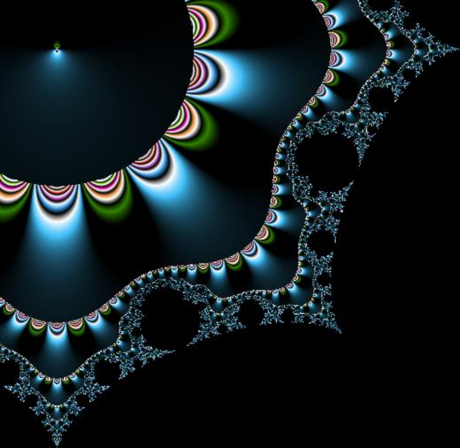
Clean code

Like use after free

Use oriented object language

Good Design

RAII Pattern (init in constructor, free in destructor)



Defense in depth

a posteriori

Dynamic code review

Valgrind

Null pointer dereferences

reading to null

Small exemple

What if cb is null ?

```
int tick(handler * h, void * data)
{
  cb_t * cb = h->cb ;
  return cb(data) ;
}
```

Small exemple

What if cb is null ?

```
int tick(handler * h, void * data)
{
  cb_t * cb = h->cb ;
  return cb(data) ;
}
```

```
$ ./null
Segmentation fault (core dumped)
```

Prerequisites

In which case is it a problem ?

Kernel process

To be allowed to use a pointer outside userland

Everytime a null pointer is used

Non-initilized, non catching error on allocation, exhausted memory, compiler optimization, …

What if ?

We succeed to allocate page 0

```
#include <sys/mman.h>

int main()
{
    mmap(0, 4096,
        PROT_READ    | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS
                    | MAP_FIXED,
        -1, 0);

        // …

  ticks(&h, &data);
}
```

What if ?

We succeed to allocate page 0

```
#include <sys/mman.h>

int main()
{
    mmap(0, 4096,
        PROT_READ   | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS
                    | MAP_FIXED,
        -1, 0);

  // All you can code since you can
  // write at 0

  ticks(&h, &data);
}
```

DOS

reboot(), panic(), …

Privilege escalation

prepare_creds() & commit_creds()

execve()

Clean code

Like for Stack overflow

Check given pointers

Good Design

RAII Pattern

Defense in depth

a posteriori

Disable address 0 mapping

Done since kernel 2.6.23 (2007)

But ?

Can be modify by admin

Use a suid program to modify the conf file

And set it to 0

Uninitialized local variable

what will we read ?

Exemple

From CVE 2016-8385

```c
int main(int argc, char ** argv) {
  if (argc < 3) {
    return 1 ;
  }

  function_a(argv[1]) ;
  function_b(argv[2]) ;
  return 0 ;
}
```

```c
void function_a(char * buf) {
  size_t len = strlen(buf) ;
  // do stuffs
}

void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```

Stack View

From CVE 2016-8385

```
int main(int argc, char ** argv) {
  if (argc < 3) {
    return 1 ;
  }
          // HERE
  function_a(argv[1]) ;
  function_b(argv[2]) ;
  return 0 ;
}
```

Stack View

From CVE 2016-8385

```
int main(int argc, char ** argv) {
  if (argc < 3) {
    return 1 ;
  }

  function_a(argv[1]) ;
  function_b(argv[2]) ;
  return 0 ;
}
```

Stack View

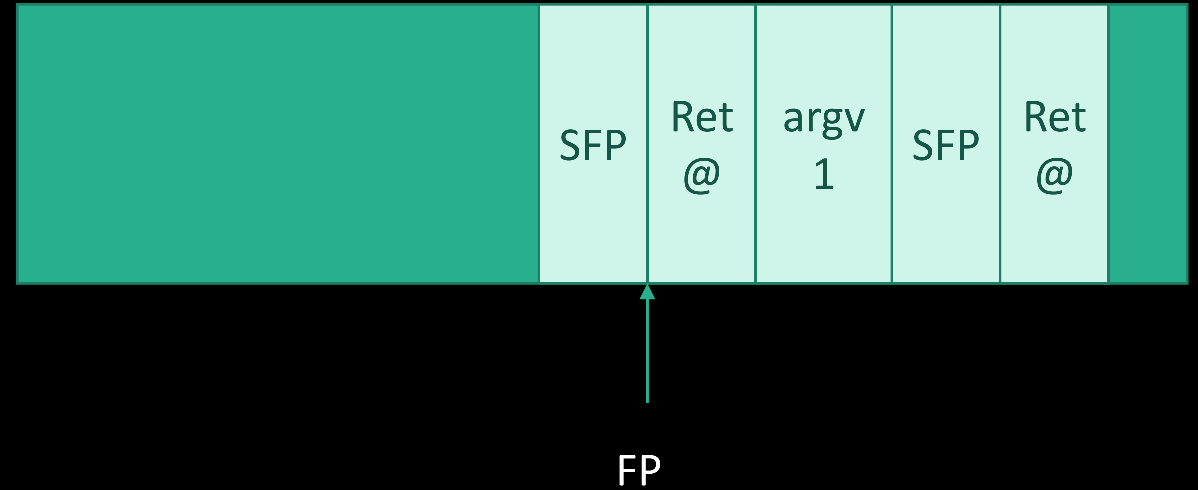
From CVE 2016-8385

```
void function_a(char * buf) {
  size_t len = strlen(buf) ;
  // do stuffs
}
```

Stack View

From CVE 2016-8385

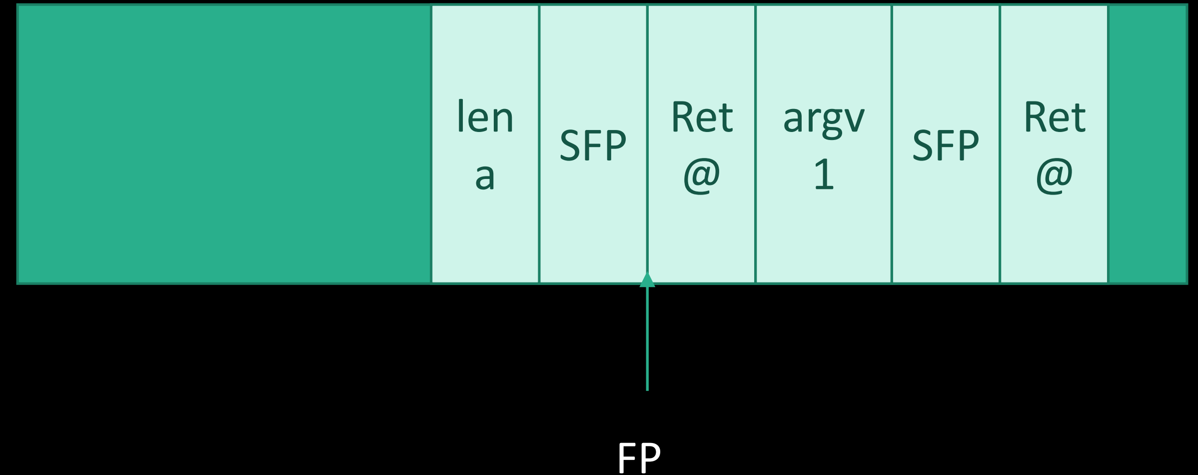
```
void function_a(char * buf) {
  size_t len = strlen(buf) ;
  // do stuffs
}
```



Stack View

From CVE 2016-8385

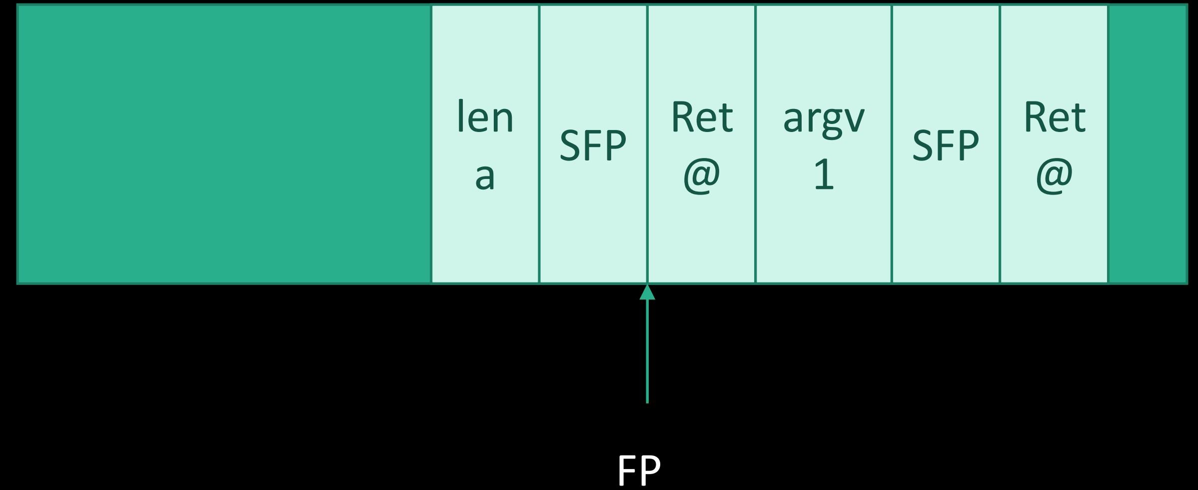
```
void function_a(char * buf) {
  size_t len = strlen(buf) ;
  // do stuffs
}
```



Stack View

From CVE 2016-8385

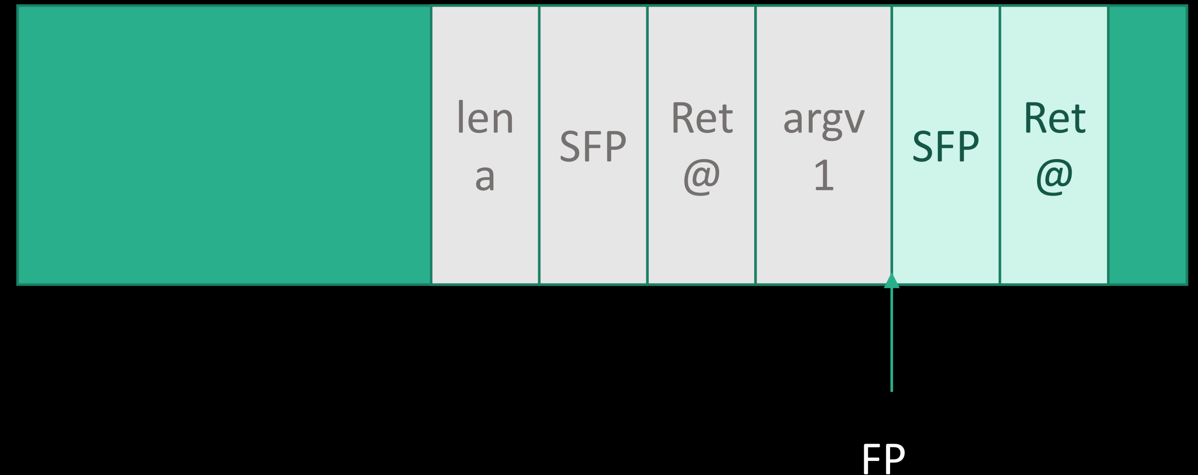
```
void function_a(char * buf) {
  size_t len = strlen(buf) ;
  // do stuffs
}
```



Stack View

From CVE 2016-8385

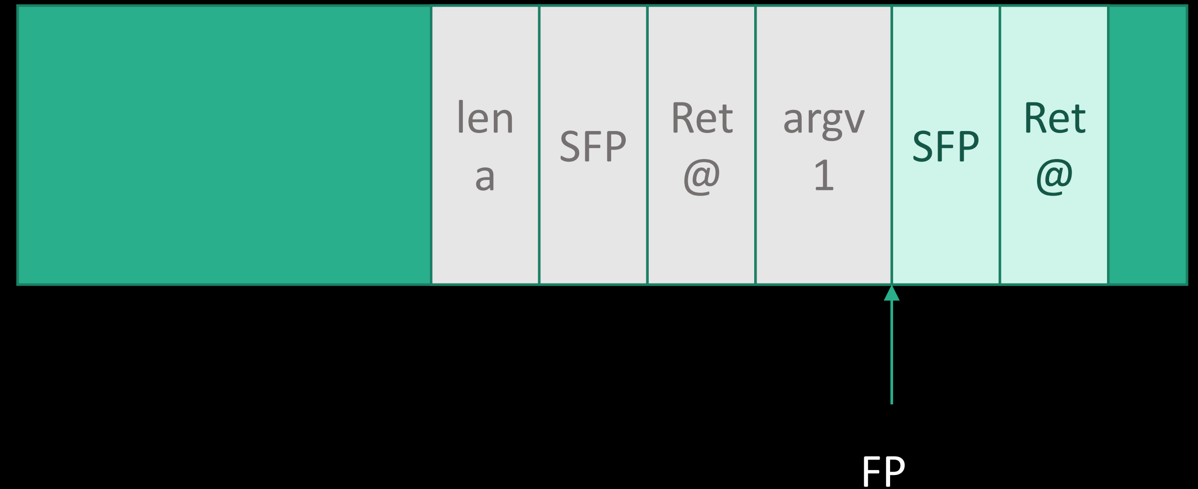
```
int main(int argc, char ** argv) {
  if (argc < 3) {
    return 1 ;
  }

  function_a(argv[1]) ; // HERE
  function_b(argv[2]) ;
  return 0 ;
}
```



Stack View

From CVE 2016-8385

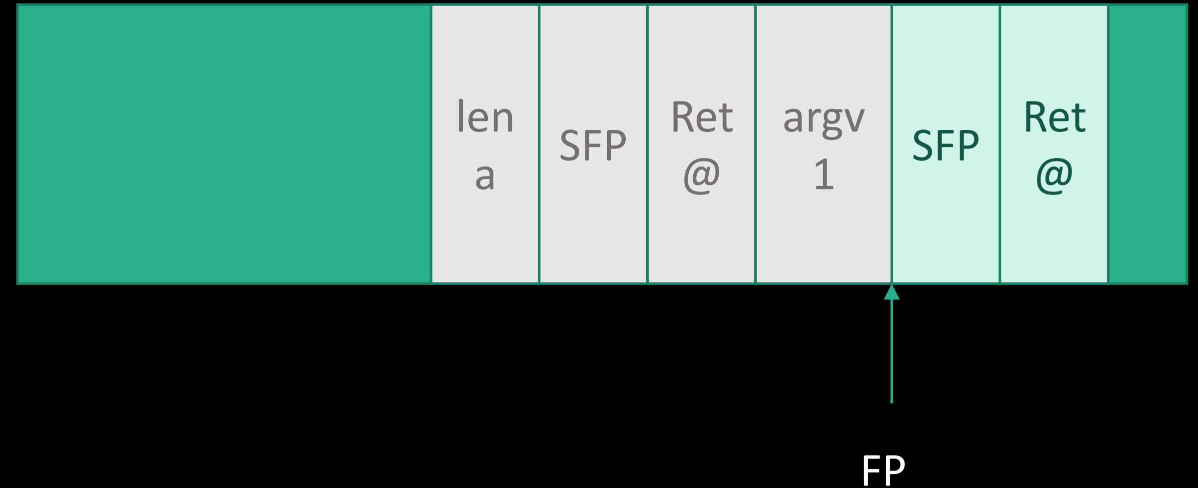
```
int main(int argc, char ** argv) {
  if (argc < 3) {
    return 1 ;
  }

  function_a(argv[1]) ;
  function_b(argv[2]) ;
  return 0 ;
}
```



Stack View

From CVE 2016-8385

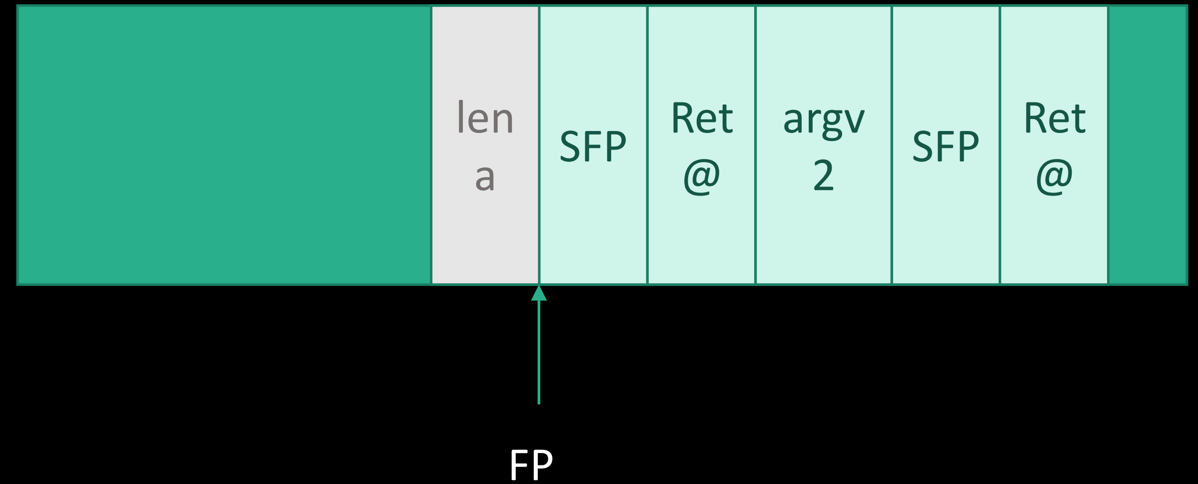
```
void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```



Stack View

From CVE 2016-8385

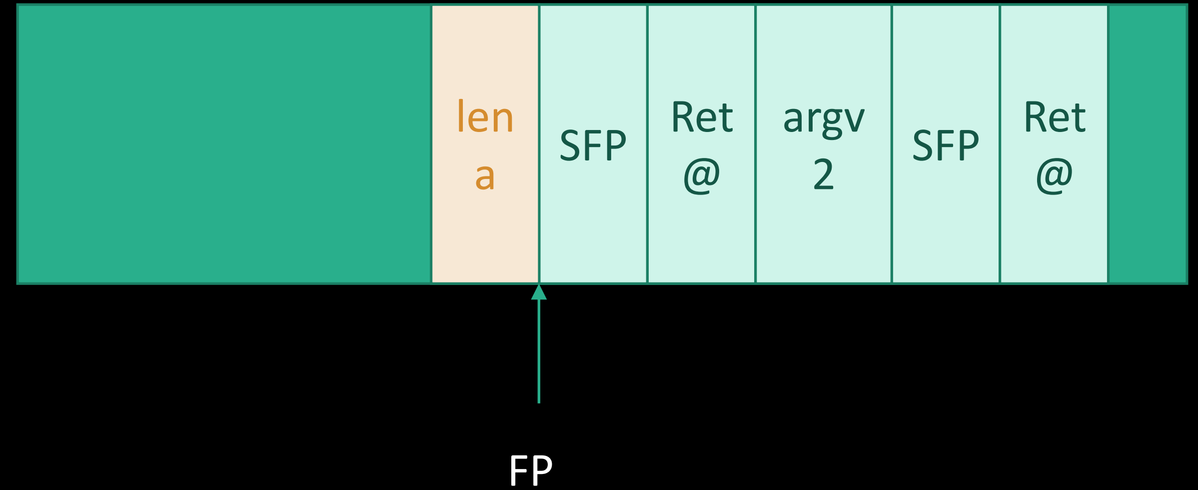
```
void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```



Stack View

From CVE 2016-8385

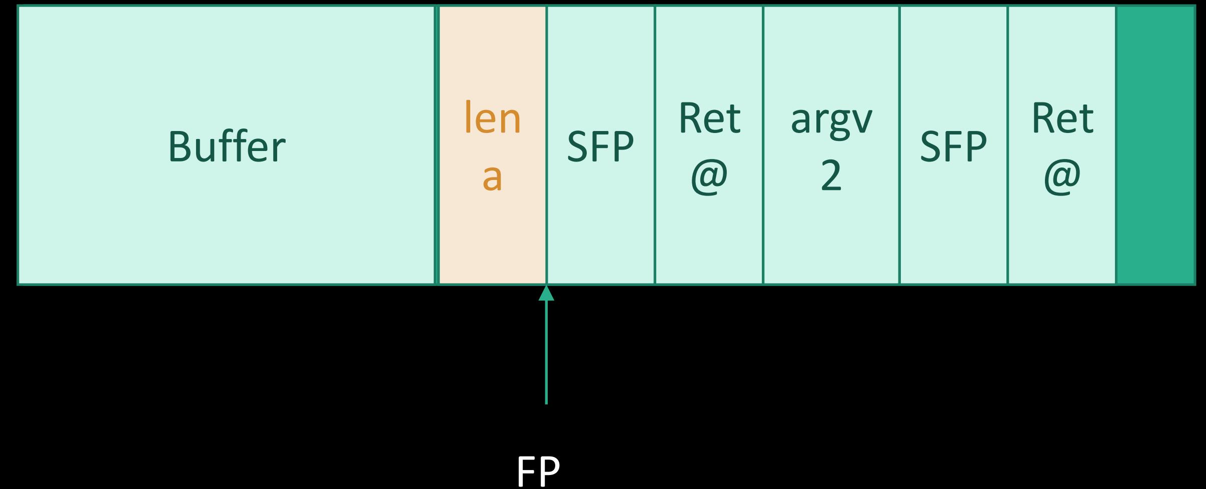
```
void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```



Stack View

From CVE 2016-8385

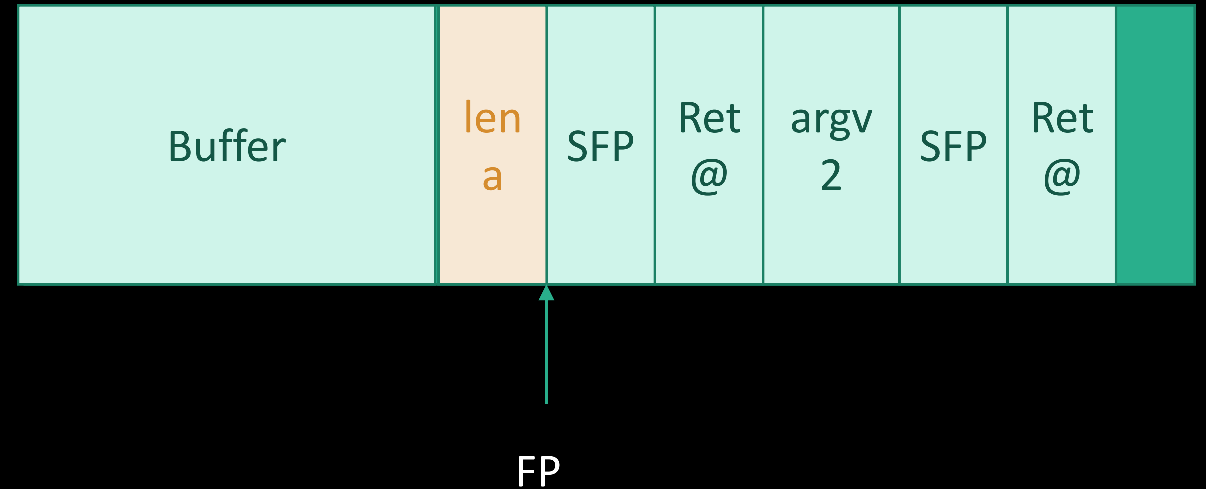
```
void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```



Stack View

From CVE 2016-8385

```
void function_b(char * buf) {
  size_t len ;
  char buffer[512] ;
  strncpy(buffer, buf, len) ;
}
```



Can lead to a Stack Buffer Overflow !!!

Clean code

Always

Always initialize variable

At declaration

Statical Review

Because gcc not always warns

Defense in depth

a posteriori

Compiler options

-ftrivial-auto-var-init=zero (since 2021)

-Wuninitialized

-Wmaybe-uninitilized

-Winit-self