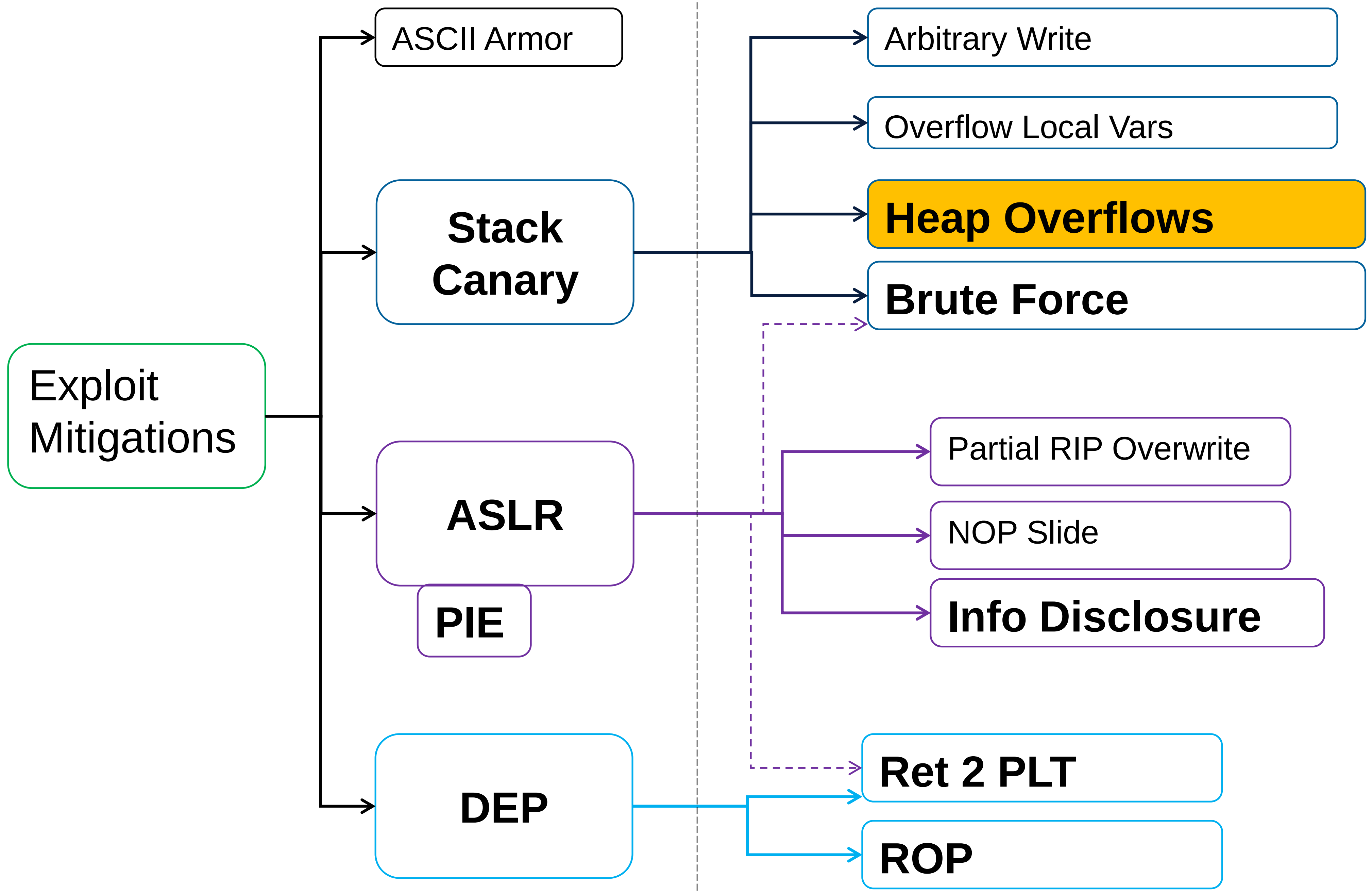
Defeat Exploit Mitigation Heap Attacks



ASCII Armor
Stack Canary
Exploit Mitigations
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

Content

Content:

- About vulnerability counting
- UAF Explained
- UAF Example
- What is Object Orientation
- Vtables
- Garbage collection
- Stack pivoting
- ~~Other heap attacks~~
- Heap massage

Heap Attacks

Heap Attacks:

Alternative for stack based buffer overflow to perform memory corruption

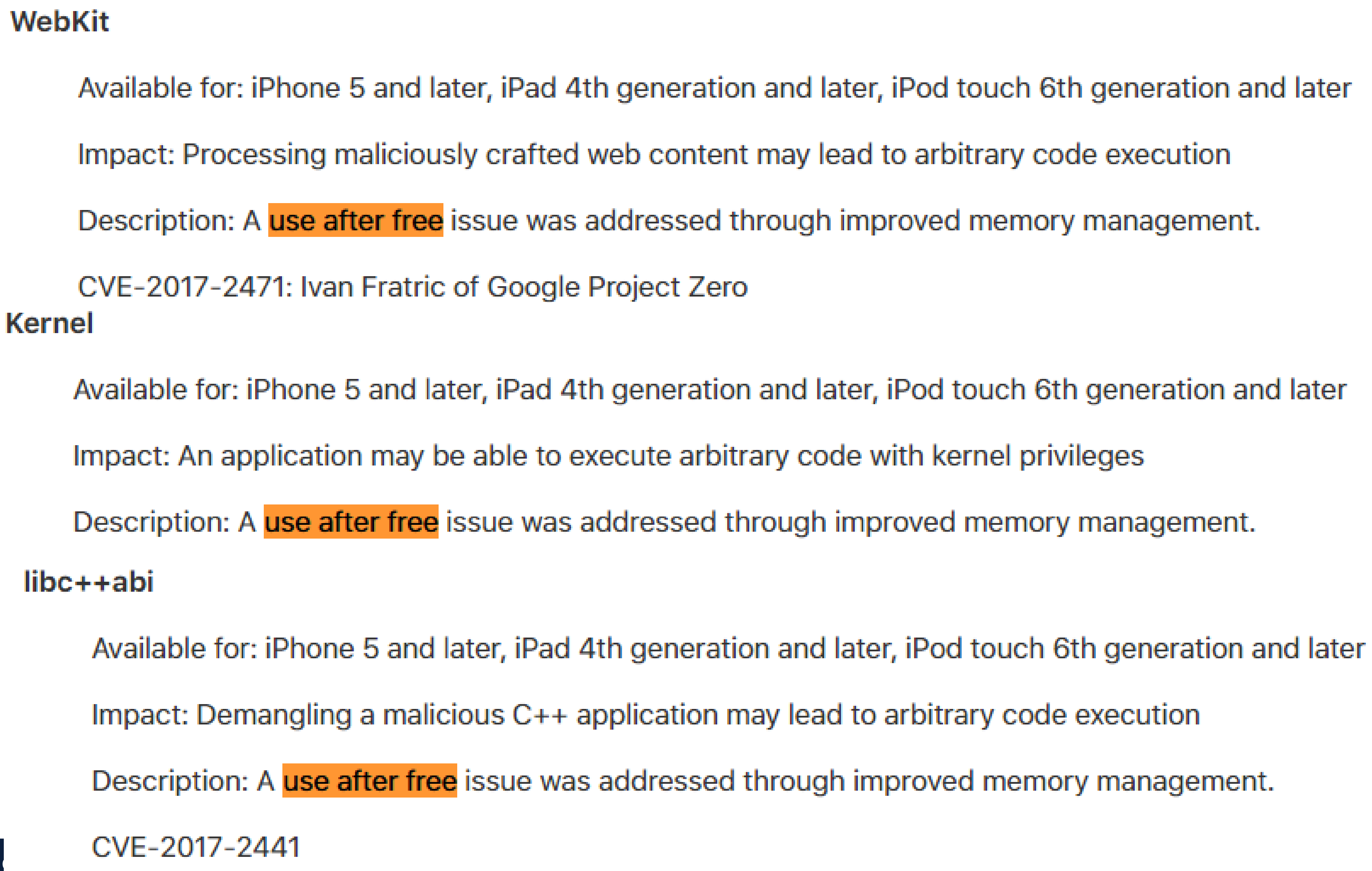
Heap Attack Types:

- Use after free
- Double Free
- Intra-chunk heap overflow
- Inter-chunk heap overflow
- Type confusion

Heap Attacks:
Use After Free (UAF)

Intermezzo

Use After Free



WebKit

Available for: iPhone 5 and later, iPad 4th generation and later, iPod touch 6th generation and later

Impact: Processing maliciously crafted web content may lead to arbitrary code execution

Description: A use after free issue was addressed through improved memory management.

CVE-2017-2471: Ivan Fratric of Google Project Zero

Kernel

Available for: iPhone 5 and later, iPad 4th generation and later, iPod touch 6th generation and later

Impact: An application may be able to execute arbitrary code with kernel privileges

Description: A use after free issue was addressed through improved memory management.

libc++abi

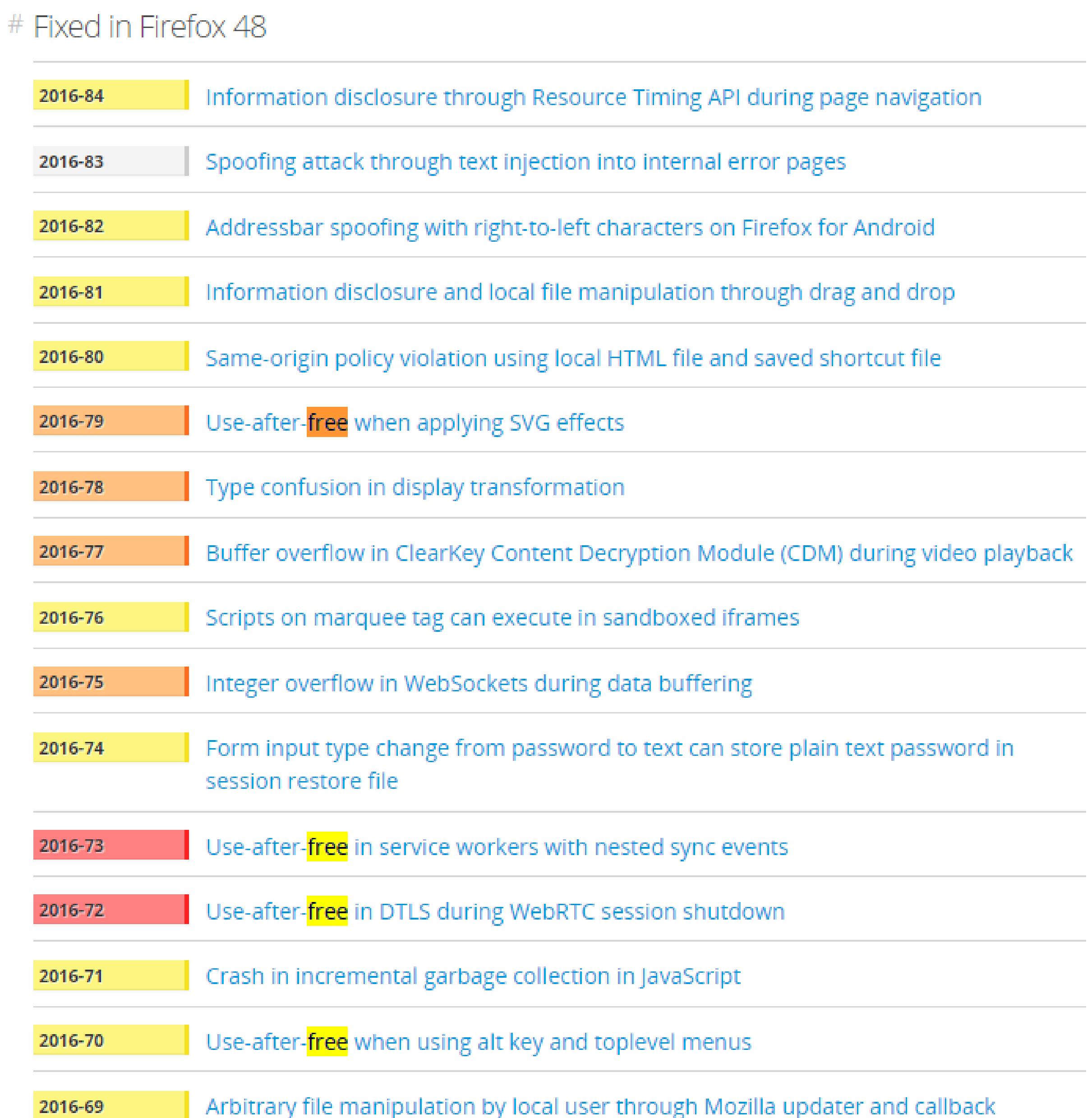
Available for: iPhone 5 and later, iPad 4th generation and later, iPod touch 6th generation and later

Impact: Demangling a malicious C++ application may lead to arbitrary code execution

Description: A use after free issue was addressed through improved memory management.

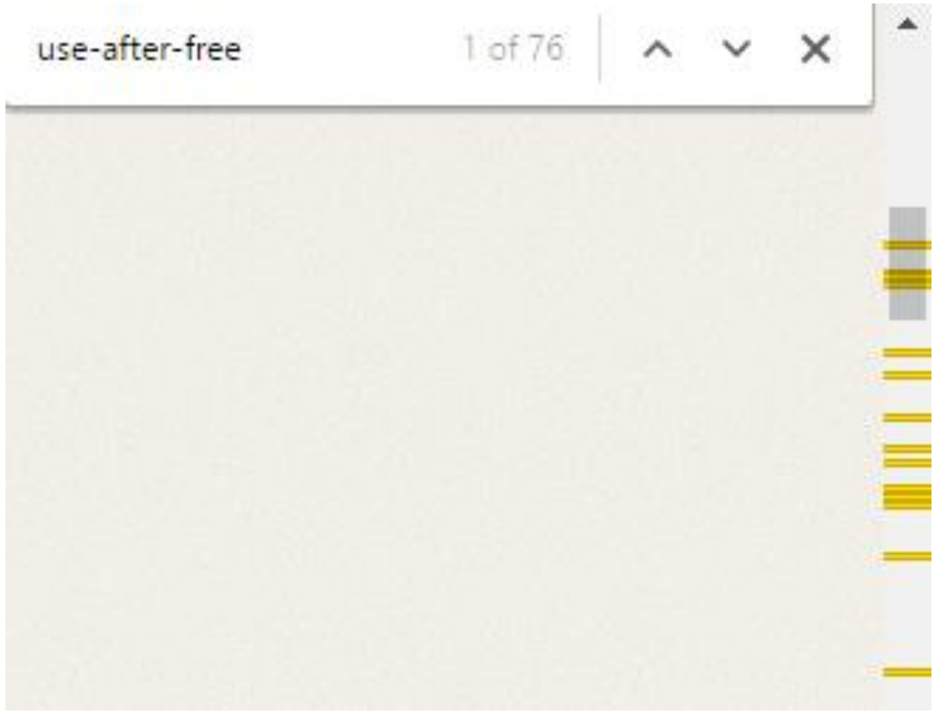
CVE-2017-2441

Use After Free

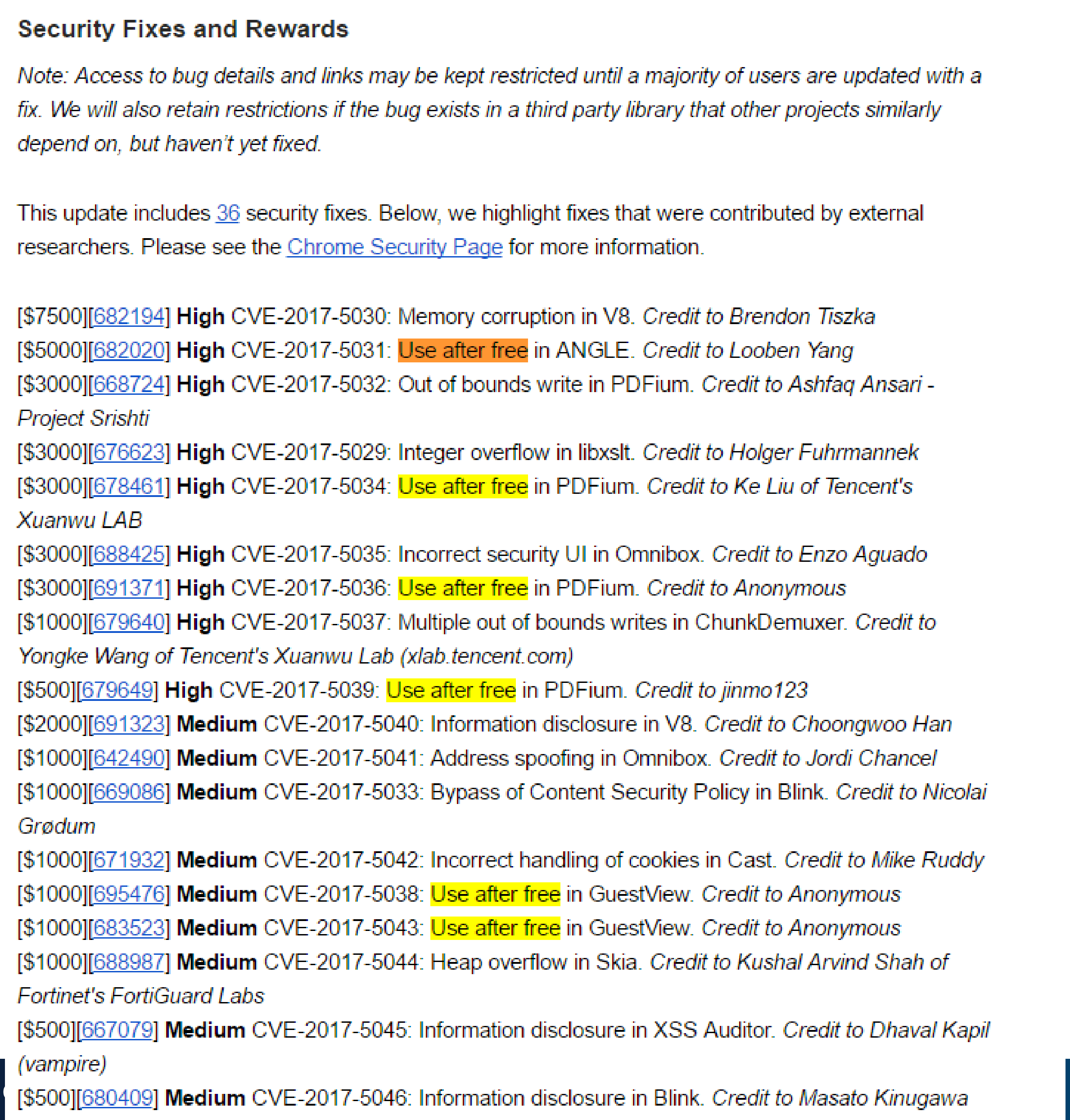


# Fixed in Firefox 48

2016-84	Information disclosure through Resource Timing API during page navigation
2016-83	Spoofing attack through text injection into internal error pages
2016-82	Addressbar spoofing with right-to-left characters on Firefox for Android
2016-81	Information disclosure and local file manipulation through drag and drop
2016-80	Same-origin policy violation using local HTML file and saved shortcut file
2016-79	Use-after-free when applying SVG effects
2016-78	Type confusion in display transformation
2016-77	Buffer overflow in ClearKey Content Decryption Module (CDM) during video playback
2016-76	Scripts on marquee tag can execute in sandboxed iframes
2016-75	Integer overflow in WebSockets during data buffering
2016-74	Form input type change from password to text can store plain text password in session restore file
2016-73	Use-after-free in service workers with nested sync events
2016-72	Use-after-free in DTLS during WebRTC session shutdown
2016-71	Crash in incremental garbage collection in JavaScript
2016-70	Use-after-free when using alt key and toplevel menus
2016-69	Arbitrary file manipulation by local user through Mozilla updater and callback



Use after free



Security Fixes and Rewards

Note: Access to bug details and links may be kept restricted until a majority of users are updated with a fix. We will also retain restrictions if the bug exists in a third party library that other projects similarly depend on, but haven't yet fixed.

This update includes 36 security fixes. Below, we highlight fixes that were contributed by external researchers. Please see the Chrome Security Page for more information.

[$7500][682194] **High** CVE-2017-5030: Memory corruption in V8. *Credit to Brendon Tiszka*
[$5000][682020] **High** CVE-2017-5031: Use after free in ANGLE. *Credit to Looben Yang*
[$3000][668724] **High** CVE-2017-5032: Out of bounds write in PDFium. *Credit to Ashfaq Ansari - Project Srishti*
[$3000][676623] **High** CVE-2017-5029: Integer overflow in libxslt. *Credit to Holger Fuhrmannek*
[$3000][678461] **High** CVE-2017-5034: Use after free in PDFium. *Credit to Ke Liu of Tencent's Xuanwu LAB*
[$3000][688425] **High** CVE-2017-5035: Incorrect security UI in Omnibox. *Credit to Enzo Aguado*
[$3000][691371] **High** CVE-2017-5036: Use after free in PDFium. *Credit to Anonymous*
[$1000][679640] **High** CVE-2017-5037: Multiple out of bounds writes in ChunkDemuxer. *Credit to Yongke Wang of Tencent's Xuanwu Lab (xlab.tencent.com)*
[$500][679649] **High** CVE-2017-5039: Use after free in PDFium. *Credit to jinmo123*
[$2000][691323] **Medium** CVE-2017-5040: Information disclosure in V8. *Credit to Choongwoo Han*
[$1000][642490] **Medium** CVE-2017-5041: Address spoofing in Omnibox. *Credit to Jordi Chancel*
[$1000][669086] **Medium** CVE-2017-5033: Bypass of Content Security Policy in Blink. *Credit to Nicolai Grødum*
[$1000][671932] **Medium** CVE-2017-5042: Incorrect handling of cookies in Cast. *Credit to Mike Ruddy*
[$1000][695476] **Medium** CVE-2017-5038: Use after free in GuestView. *Credit to Anonymous*
[$1000][683523] **Medium** CVE-2017-5043: Use after free in GuestView. *Credit to Anonymous*
[$1000][688987] **Medium** CVE-2017-5044: Heap overflow in Skia. *Credit to Kushal Arvind Shah of Fortinet's FortiGuard Labs*
[$500][667079] **Medium** CVE-2017-5045: Information disclosure in XSS Auditor. *Credit to Dhaval Kapil (vampire)*
[$500][680409] **Medium** CVE-2017-5046: Information disclosure in Blink. *Credit to Masato Kinugawa*

Security: Vulnerability lists

Intermezzo:

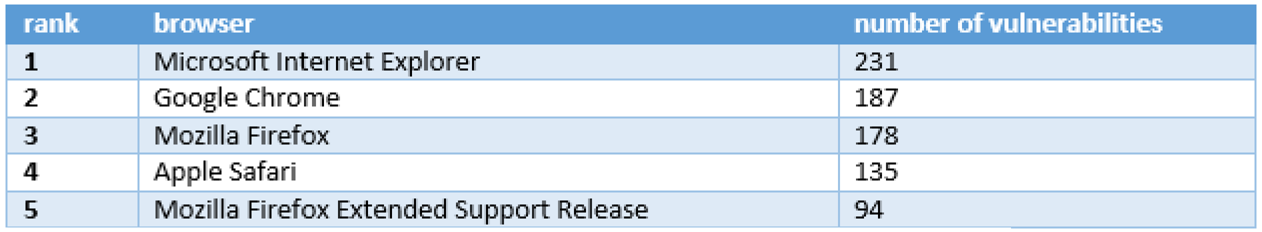
- Secure products:
  - Mention security fixes (don’t hide it)
  - Have a website with all fixed security vulnerabilities
  - As pentest: Can see which vulnerabilities are in which versions
  - Vendor is open, up to date and ready for security issues
- Bad products:
  - Don’t have a page with vulnerabilities
  - Don’t mention security fixes in changelogs
  - Vendor hides, doesn’t handle, obfuscate security issues

Security: CVE

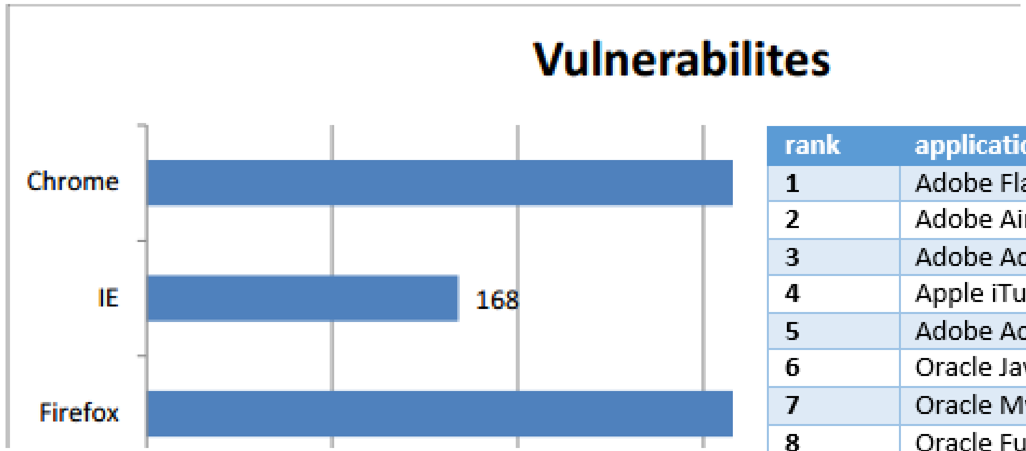
CVE:

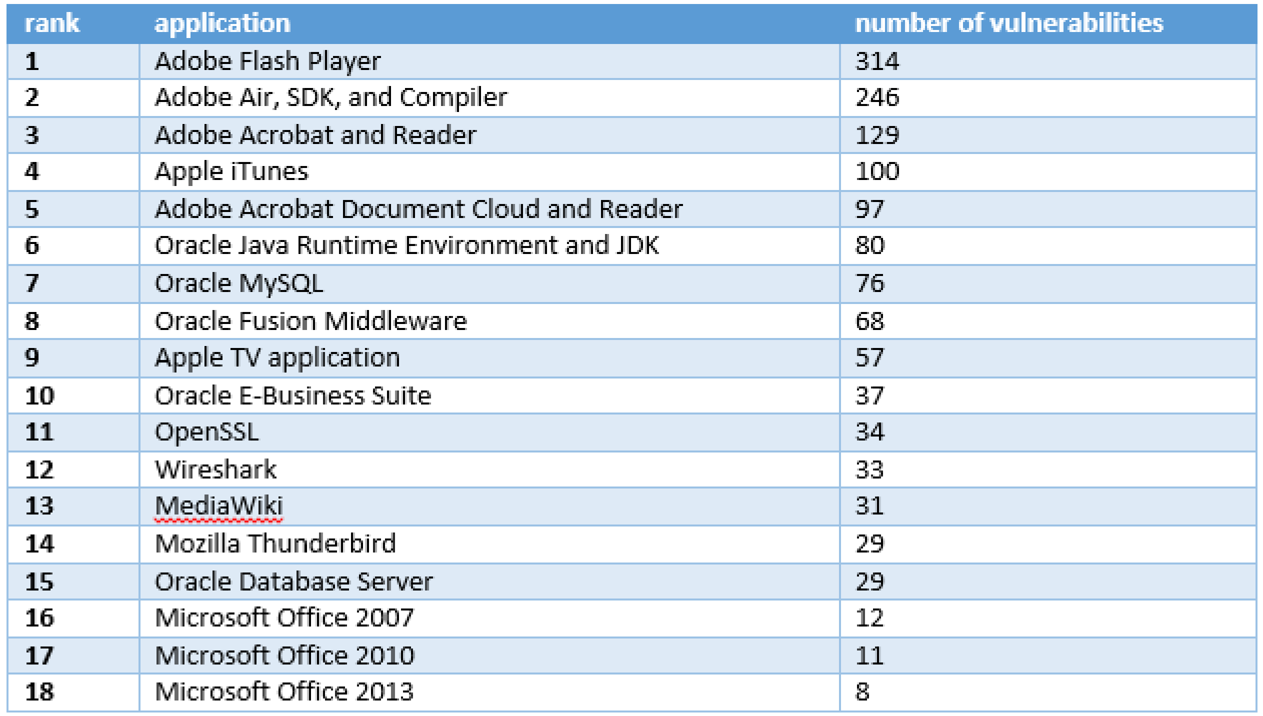
- Common Vulnerabilities and Exposures
- A vulnerability get a CVE (e.g. CVE-2017-1234)
  - Which software is affected
  - Which version
  - When did it got fixed
  - …

Security: CVE

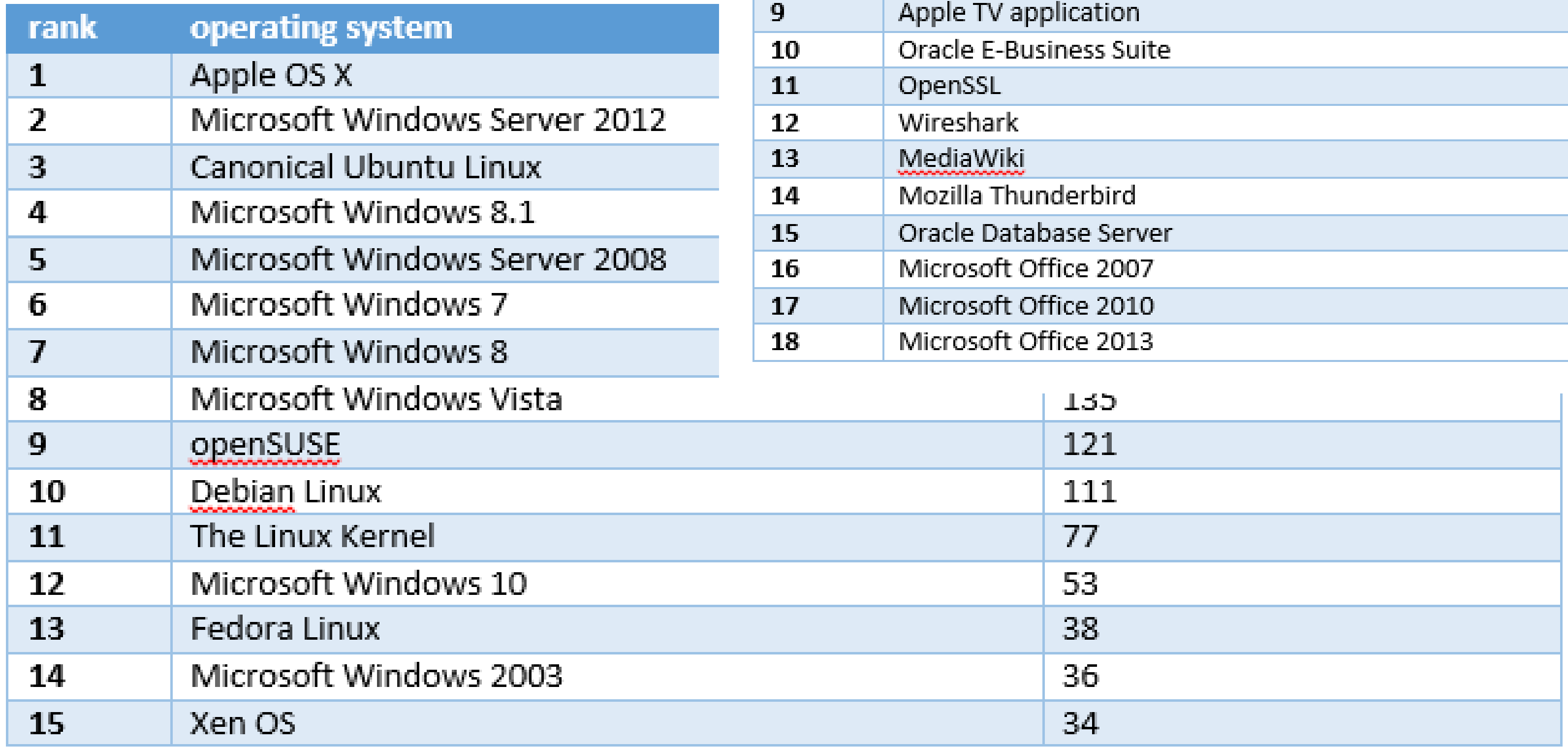


rank	browser	number of vulnerabilities
1	Microsoft Internet Explorer	231
2	Google Chrome	187
3	Mozilla Firefox	178
4	Apple Safari	135
5	Mozilla Firefox Extended Support Release	94

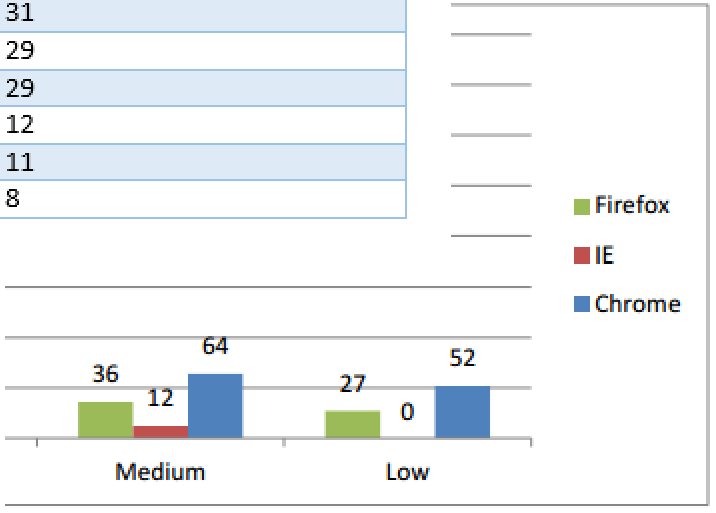




rank	application	number of vulnerabilities
1	Adobe Flash Player	314
2	Adobe Air, SDK, and Compiler	246
3	Adobe Acrobat and Reader	129
4	Apple iTunes	100
5	Adobe Acrobat Document Cloud and Reader	97
6	Oracle Java Runtime Environment and JDK	80
7	Oracle MySQL	76
8	Oracle Fusion Middleware	68
9	Apple TV application	57
10	Oracle E-Business Suite	37
11	OpenSSL	34
12	Wireshark	33
13	MediaWiki	31
14	Mozilla Thunderbird	29
15	Oracle Database Server	29
16	Microsoft Office 2007	12
17	Microsoft Office 2010	11
18	Microsoft Office 2013	8



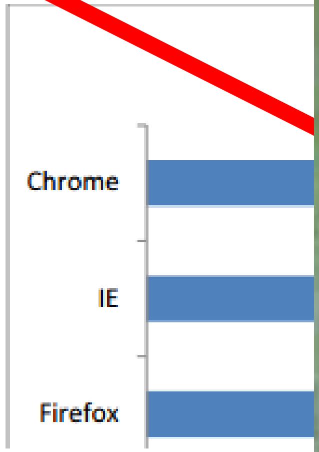
rank	operating system	
1	Apple OS X	
2	Microsoft Windows Server 2012	
3	Canonical Ubuntu Linux	
4	Microsoft Windows 8.1	
5	Microsoft Windows Server 2008	
6	Microsoft Windows 7	
7	Microsoft Windows 8	
8	Microsoft Windows Vista	135
9	openSUSE	121
10	Debian Linux	111
11	The Linux Kernel	77
12	Microsoft Windows 10	53
13	Fedora Linux	38
14	Microsoft Windows 2003	36
15	Xen OS	34

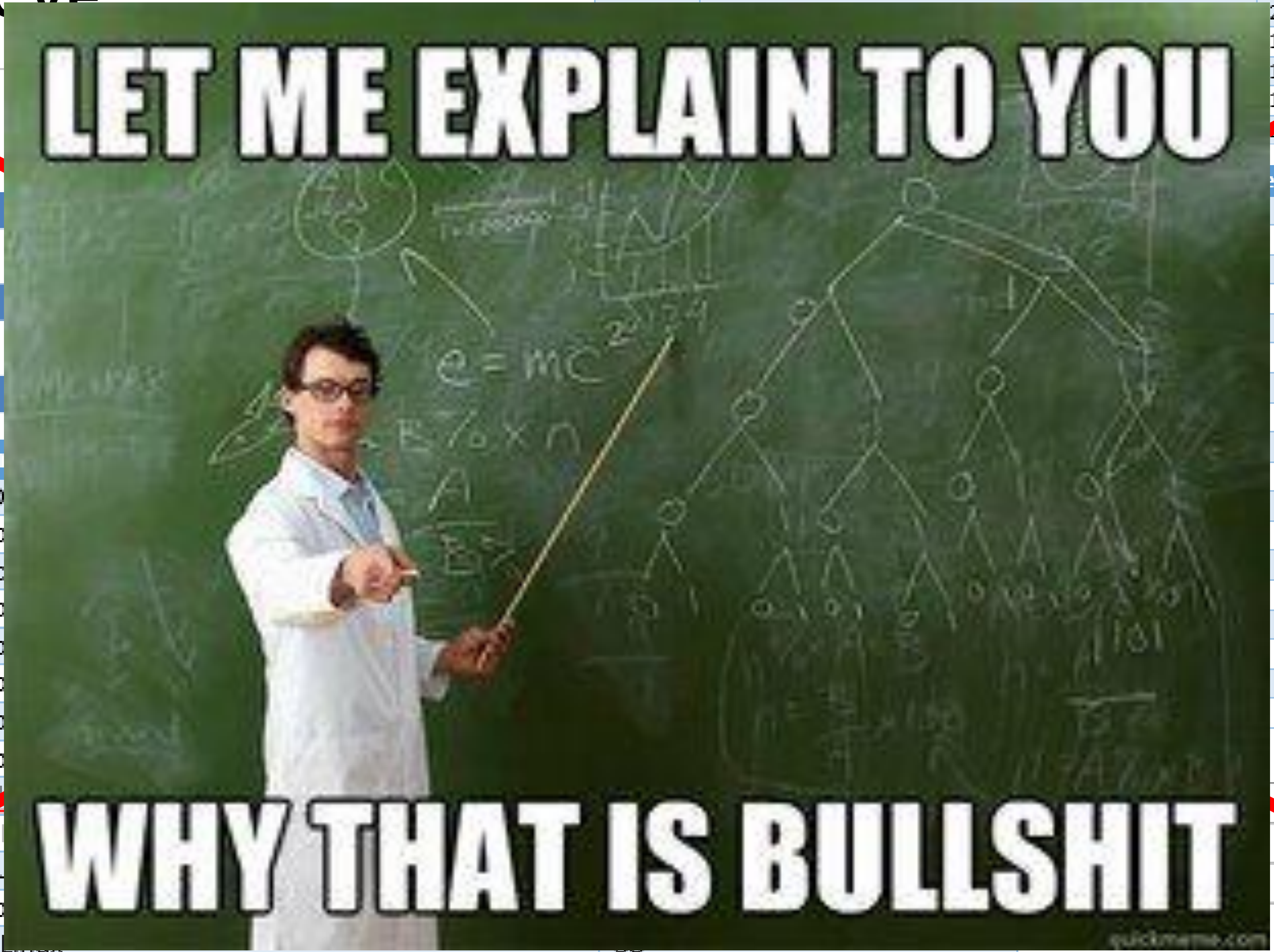


ilities by severity for each browser

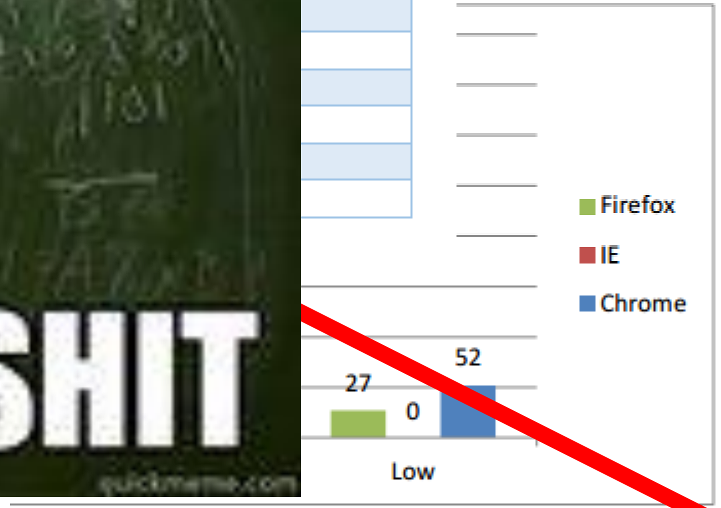
Security: CVE

rank	browser	number of vulnerabilities
		231
		187
		178
		135
		4





rank	operati	
1	Apple O	
2	Microso	
3	Canonic	
4	Microso	
5	Microso	
6	Microso	
7	Microso	
8	Microso	
9	openSU	
10	Debian	
11	The Linu	
12	Microso	
13	Fedora Linux	
14	Microsoft Windows 2003	36
15	Xen OS	34



ilities by severity for each browser

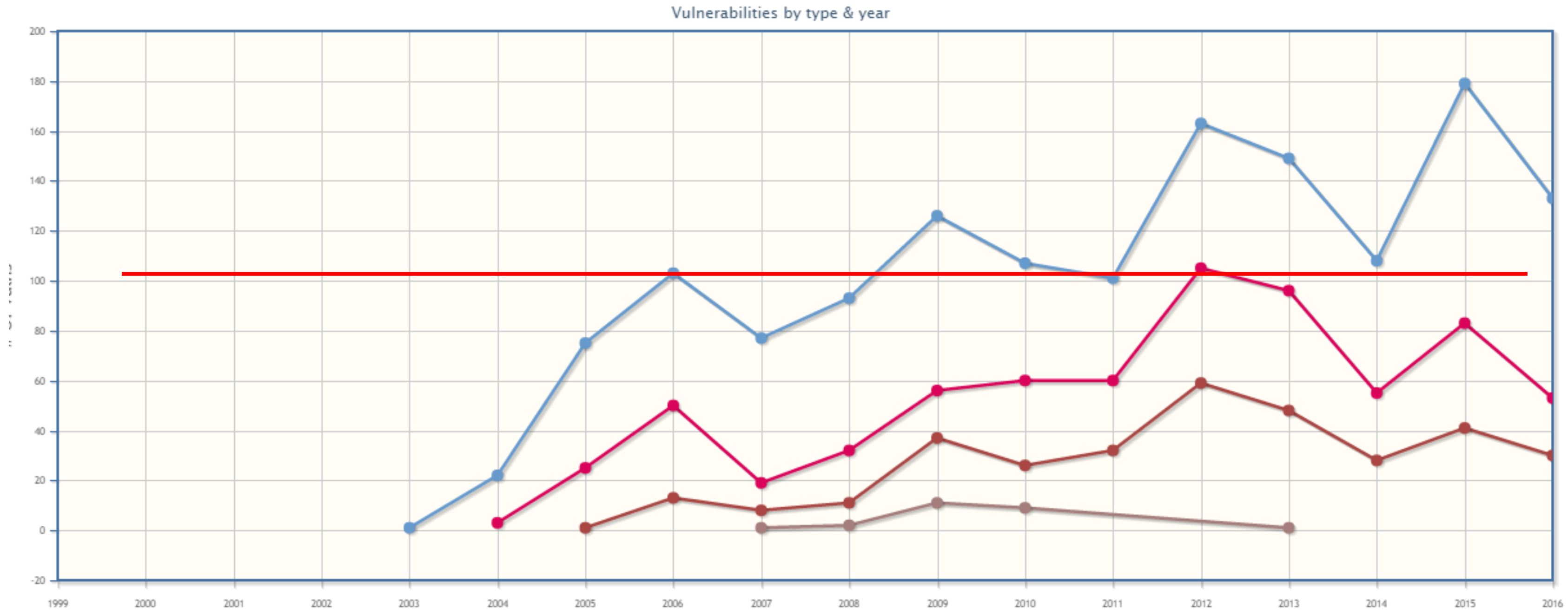
Security: CVE

Weakness comparison fails: (not just CVE)

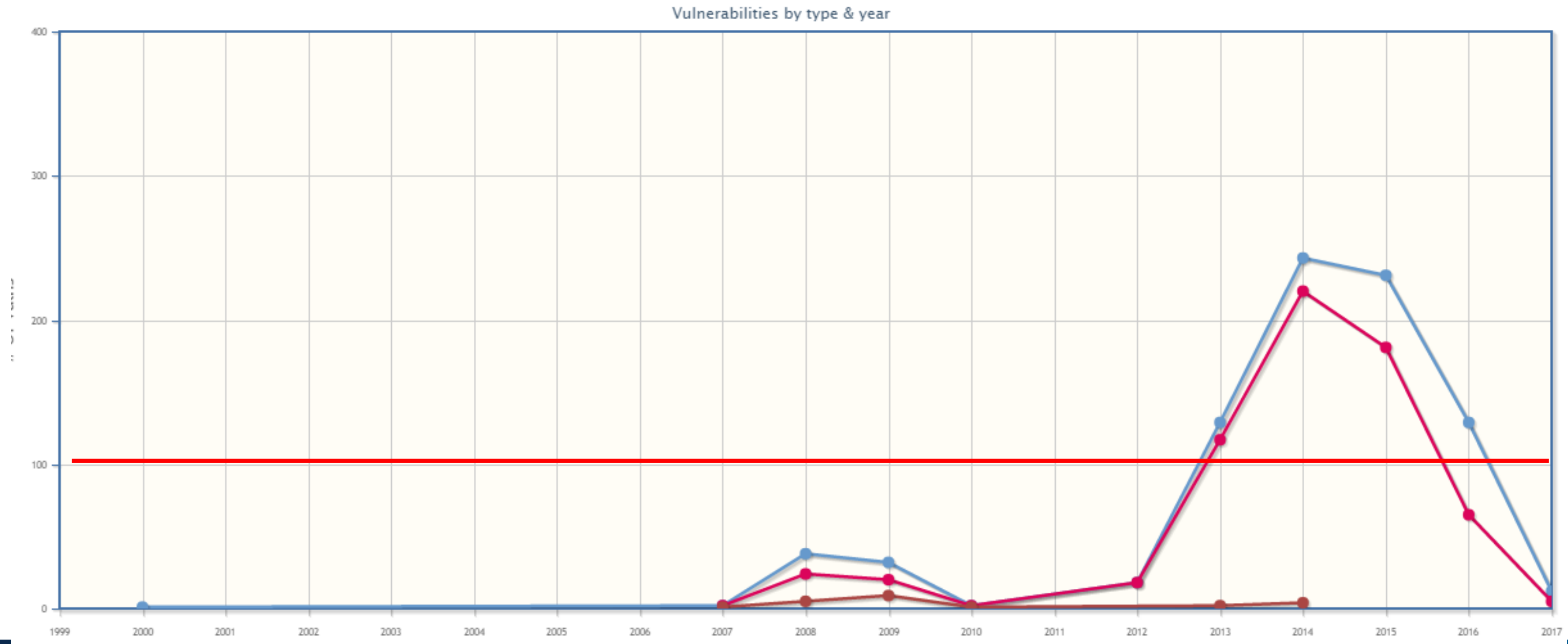
- Scope: “Windows vs Linux”
  - What is in Linux? Linux Kernel? Suse? LIBC? Bash? Apache?
  - What is in Windows? Internet Explorer? IIS?
- Severity mismatch
  - When is a vulnerability “critical”? When is it “high”?
  - Microsoft categorizes differently than Mozilla, or Google
- Number of vulnerabilities in CVE / bulletin
  - 1 vulnerability, one CVE / securtiy bullettin ?
  - 1 CVE for each product affected? (Cisco: RCE in product x, y, z)
  - 1 CVE for each individual bug? (e.g. UAF in component x, y, z)
- Vulnerablity disclosure
  - CVE’s for all the bugs found internally? (e.g. fuzzing)
  - CVE for all the bugs found by looking for similar bugs?
- …

-> Don’t compare different product’s security issues by counting <-

FF


Vulnerabilities by type & year
200
180
160
140
120
100
80
60
40
20
0
-20
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016

IE


Vulnerabilities by type & year
400
300
200
100
0
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017

Heap Attacks:
Use After Free (UAF)

Introduction

Heap Attack: UAF

UAF:

`Use after free`

Or more correctly:

`Use a an object, after the memory it has been pointing to has been freed, and now a different object is stored at that location`

Heap Attack: UAF

So, what is UAF?

- We have a pointer (of type A) to an object
- The object get’s free()’d
  - This means that the memory allocater marks the object as free
  - The object will not be modified!
  - (Similar to deleting a file on the harddisk)
  - The pointer is still valid
- Another object of type B (of the same size) get’s allocated
- Memory allocator returns the previously free’d object memory space

- Attacker has now a pointer (type A) to another object (type B)!
- This object can be modified
  - Depending on the types A and B

Heap Attack: UAF

Example: heapnote.c:

- Has: Todos
  - Can add, remove and edit a Todo
  - Has two todo lists:
    - Work
    - Private
  - Todo’s are created in one list
  - Todo’s can be added to the other list

- Has: Alarms
  - Can add, remove and edit Alarms
  - Alarms are managed in a separate Alarm list

- *Note: I tried to make a simple as possible tool which is vulnerable to UAF, not a real tool. Therefore, it does not fully makes sense. Sorry.*

Heap Attack: UAF

Heapnote.c:

```
Todo's:
  todo add <list> <prio> <todotext>
  todo edit <list>:<entry> <prio> <todotext>

List:
  todolist view <list>
  todolist add <listDst> <listSrc>:<entry>
  todolist del <list> <entry>

Alarm:
  alarm add <alarmText>
  alarm list
  alarm view <alarmIndex>
  alarm del <alarmIndex>
```

Heap Attack: UAF

```
struct Todo {
        char *body;
        int priority;
        int id;
}
```

```
struct Alarm {
        char *name;
        void (*fkt)()
        int id;
}
```

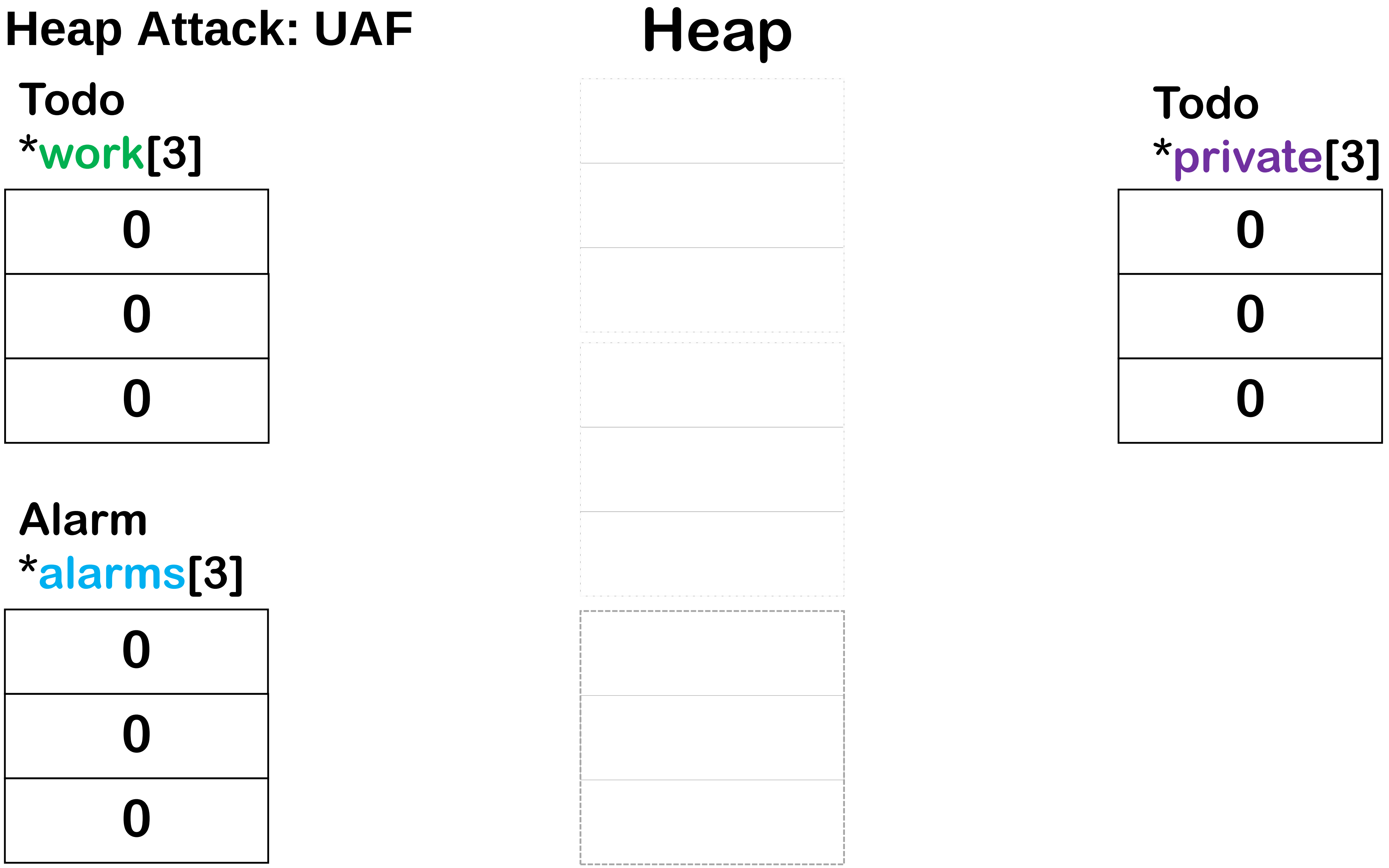
Heap Attack: UAF

```
struct Todo {
      char *body;
      int priority;
      int id;
}
```

```
struct Alarm {
      char *name;
      void (*fkt)()
      int id;
}
```

	Struct Todo:	**Struct Alarm:**
+0	`char *body`	`char *name`
+8	`int priority`	`void (*cleanup)()`
+16	`int id`	`int id`

Heap Attack: UAF



Heap UAF: Noteheap

Step 1: Add a “Todo”

```
todo add work 123 “test”
```

Todo
*work[3]

0
0
0

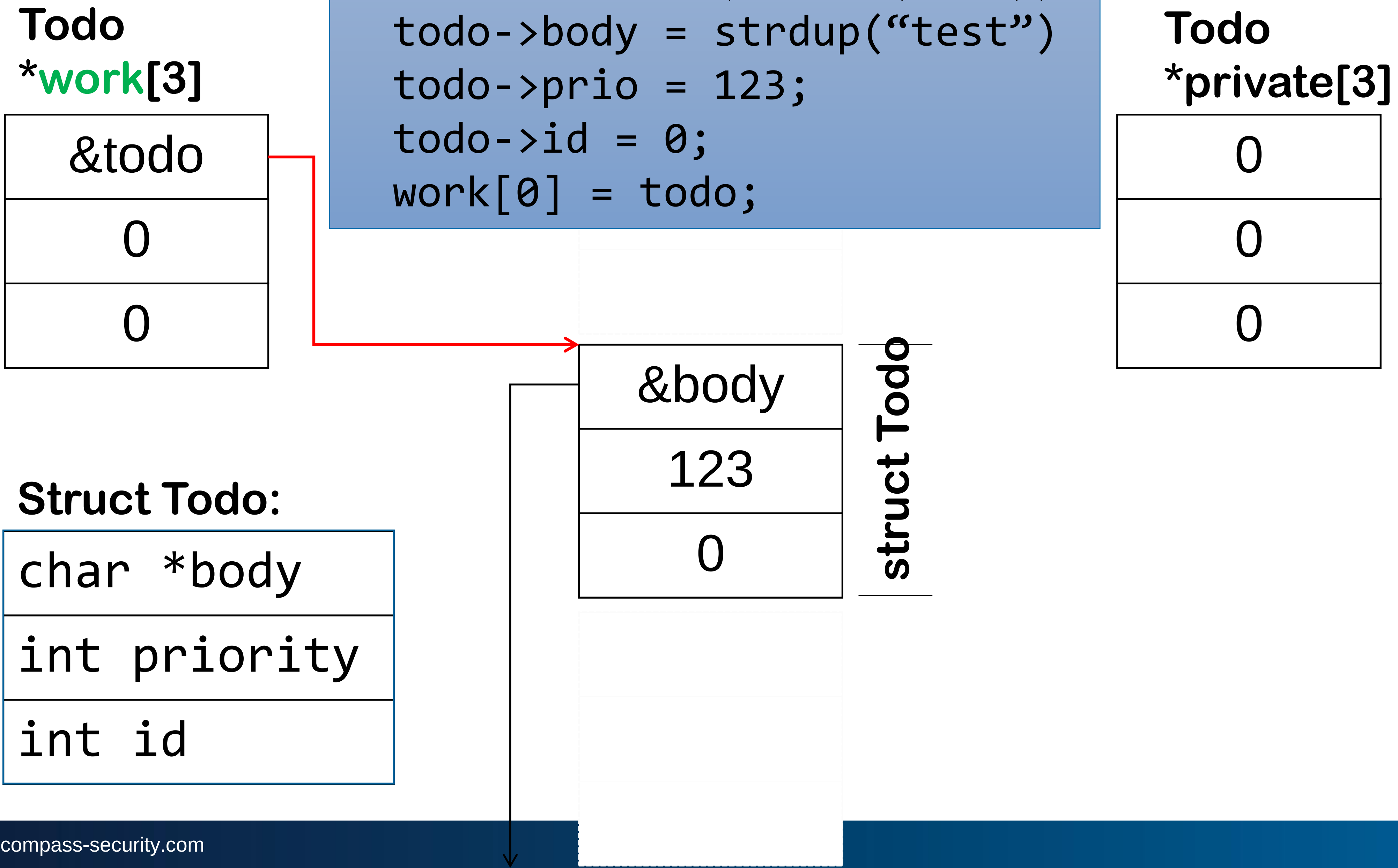
Todo
*private[3]

0
0
0

Struct Todo:

`char *body`
`int priority`
`int id`

```
todo add work 123 "test"
   todo = malloc(sizeof(Todo))
   todo->body = strdup("test")
   todo->prio = 123;
   todo->id = 0;
   work[0] = todo;
```



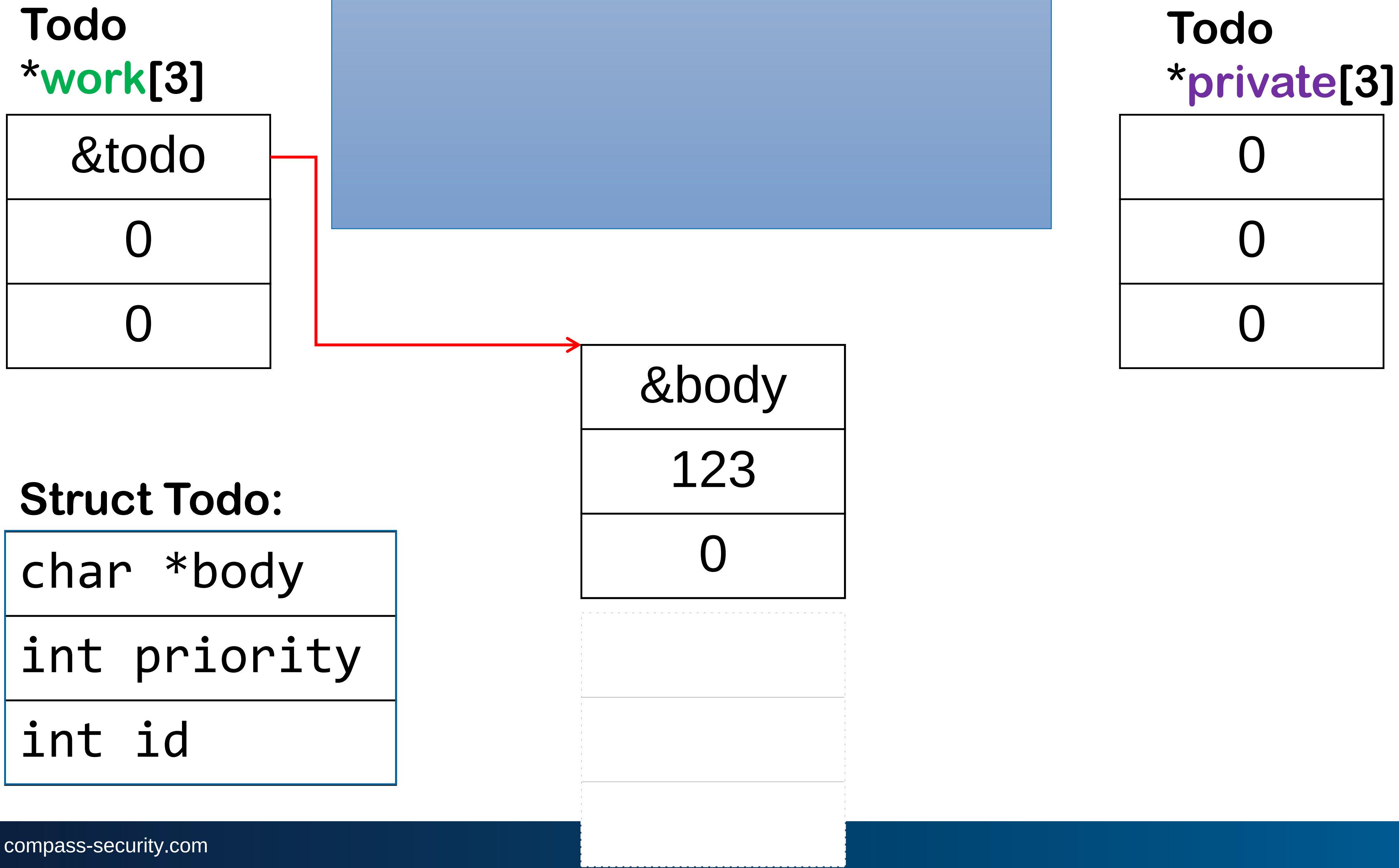
Struct Todo:

`char *body`
`int priority`
`int id`

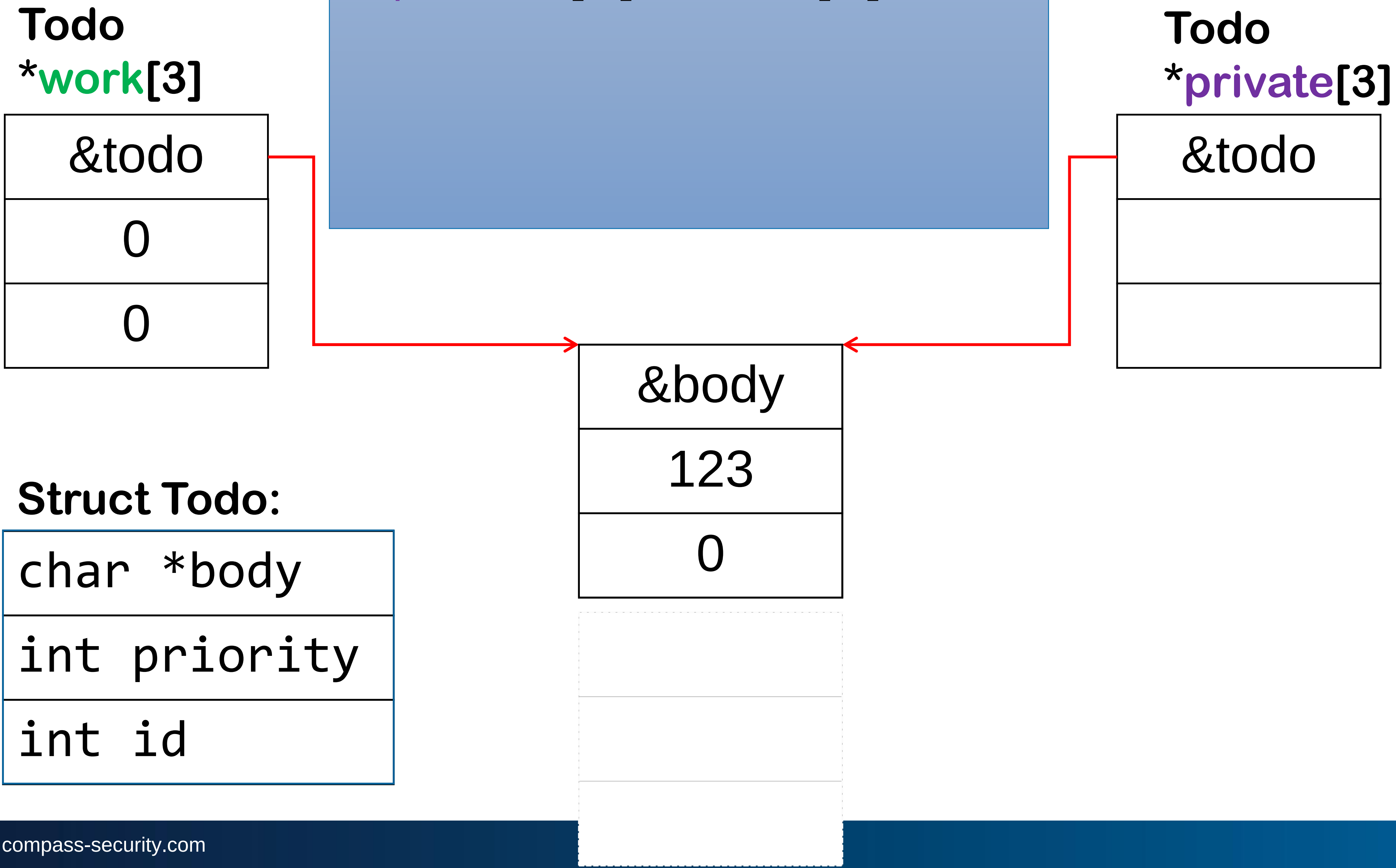
Heap UAF: Noteheap

Step 2: Add the (previously inserted) Todo from the “work” list to the “private” list

```
list add private work:0
```



```
list add private work:0
   private[0] = work[0];
```



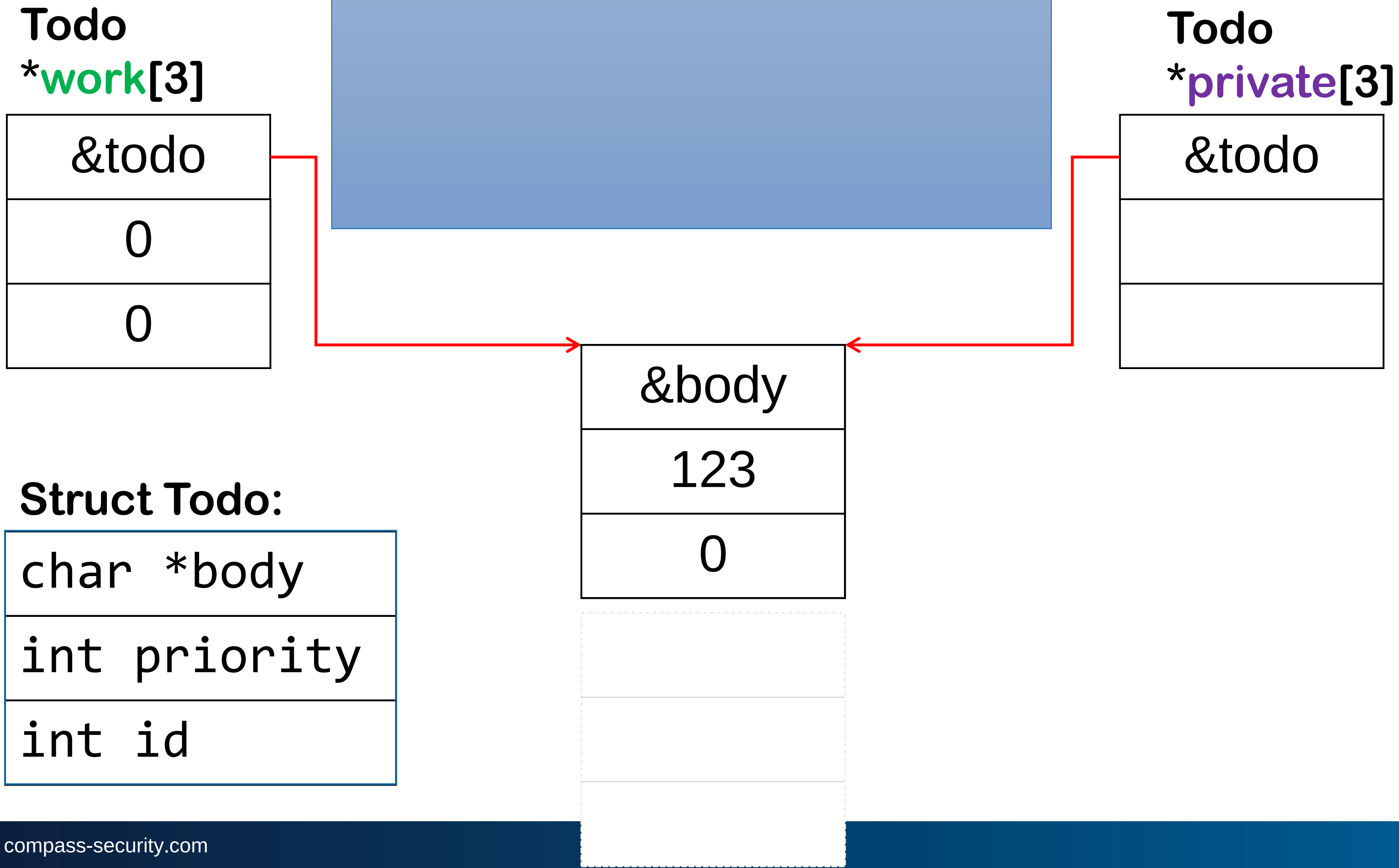
Struct Todo:

`char *body`
`int priority`
`int id`

Heap UAF: Noteheap

Step 3: Delete the “Todo” (via “work” list)

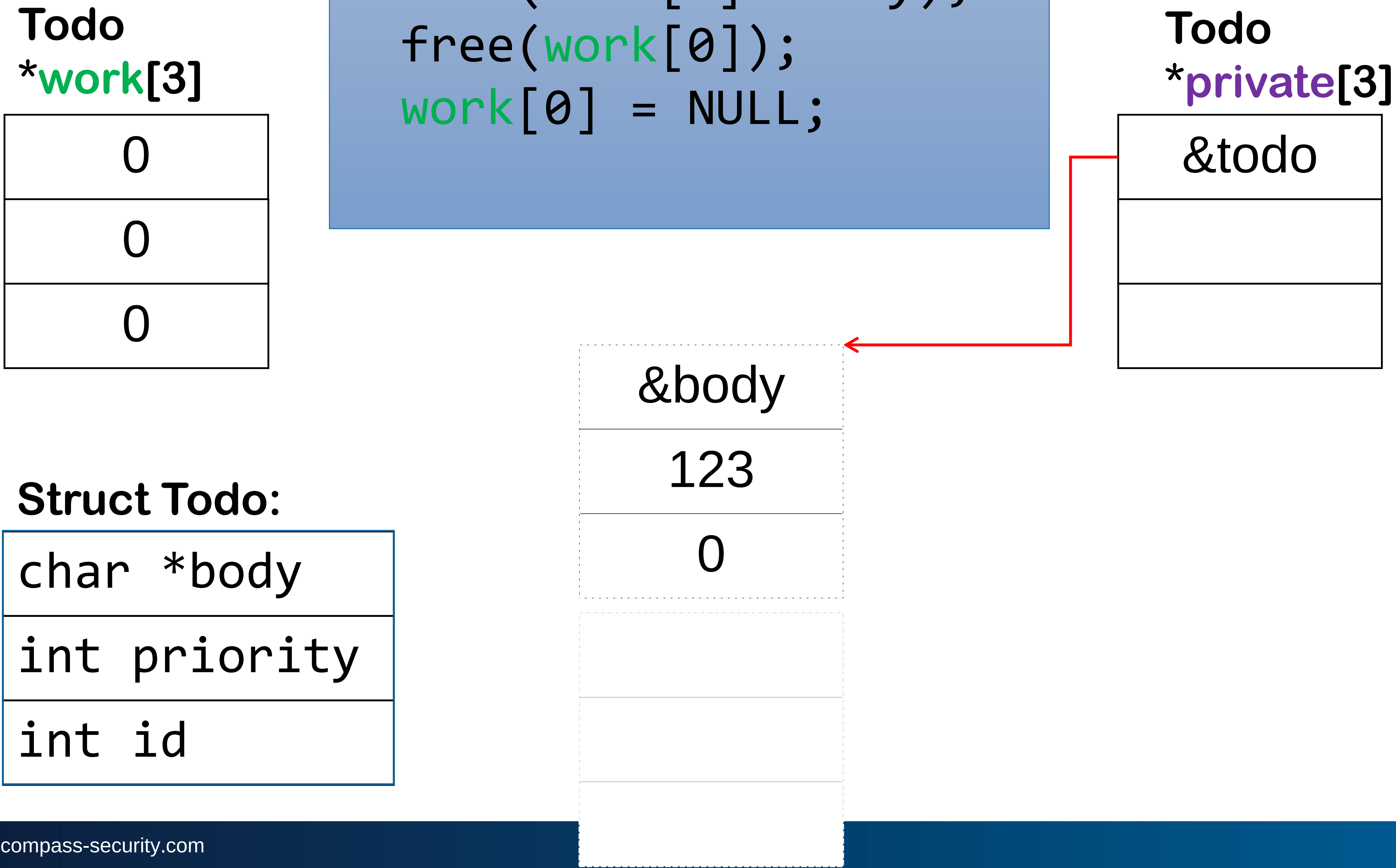
```
list del work:0
```



Struct Todo:

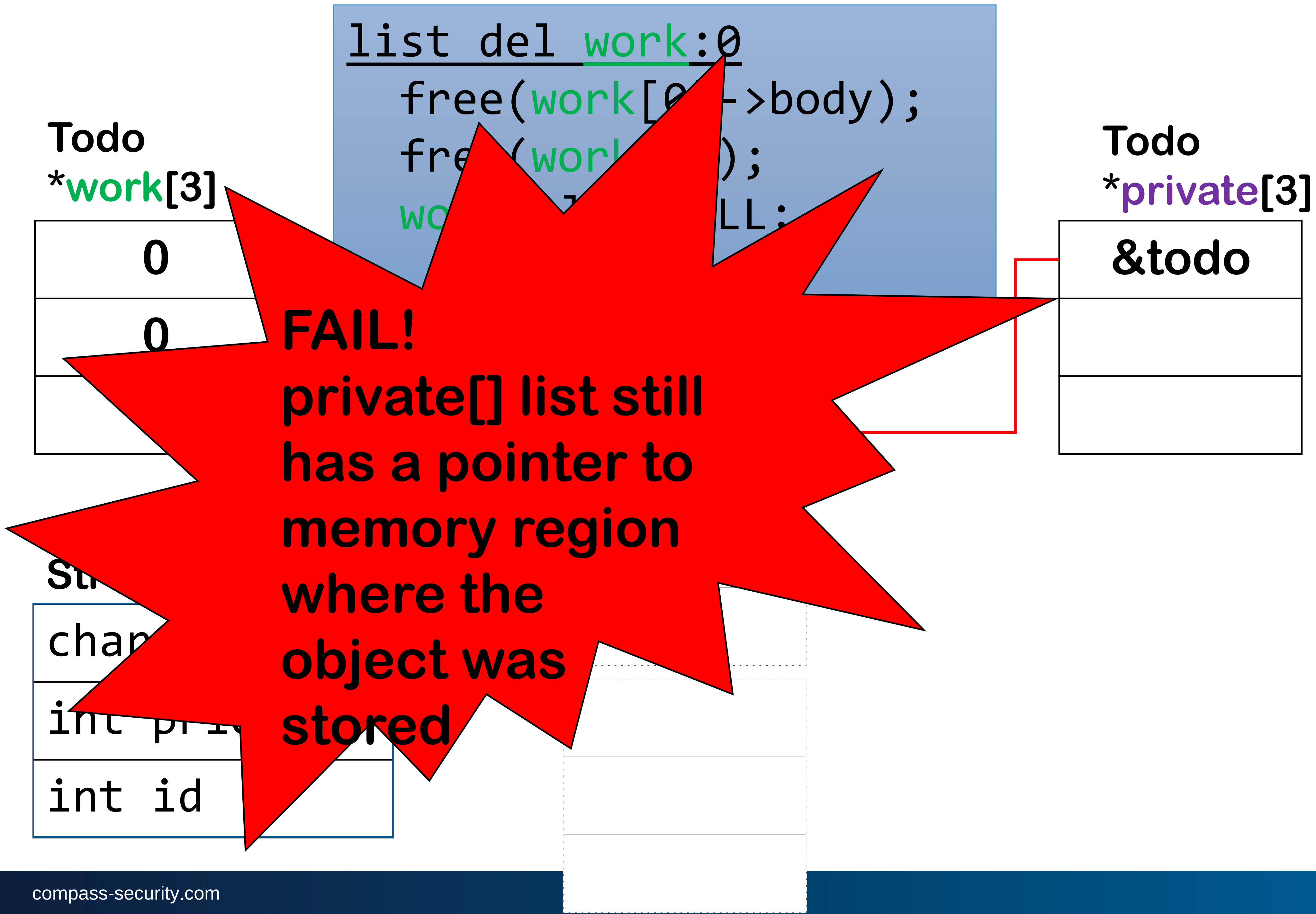
`char *body`
`int priority`
`int id`

```
list del work:0
   free(work[0]->body);
   free(work[0]);
   work[0] = NULL;
```



Struct Todo:

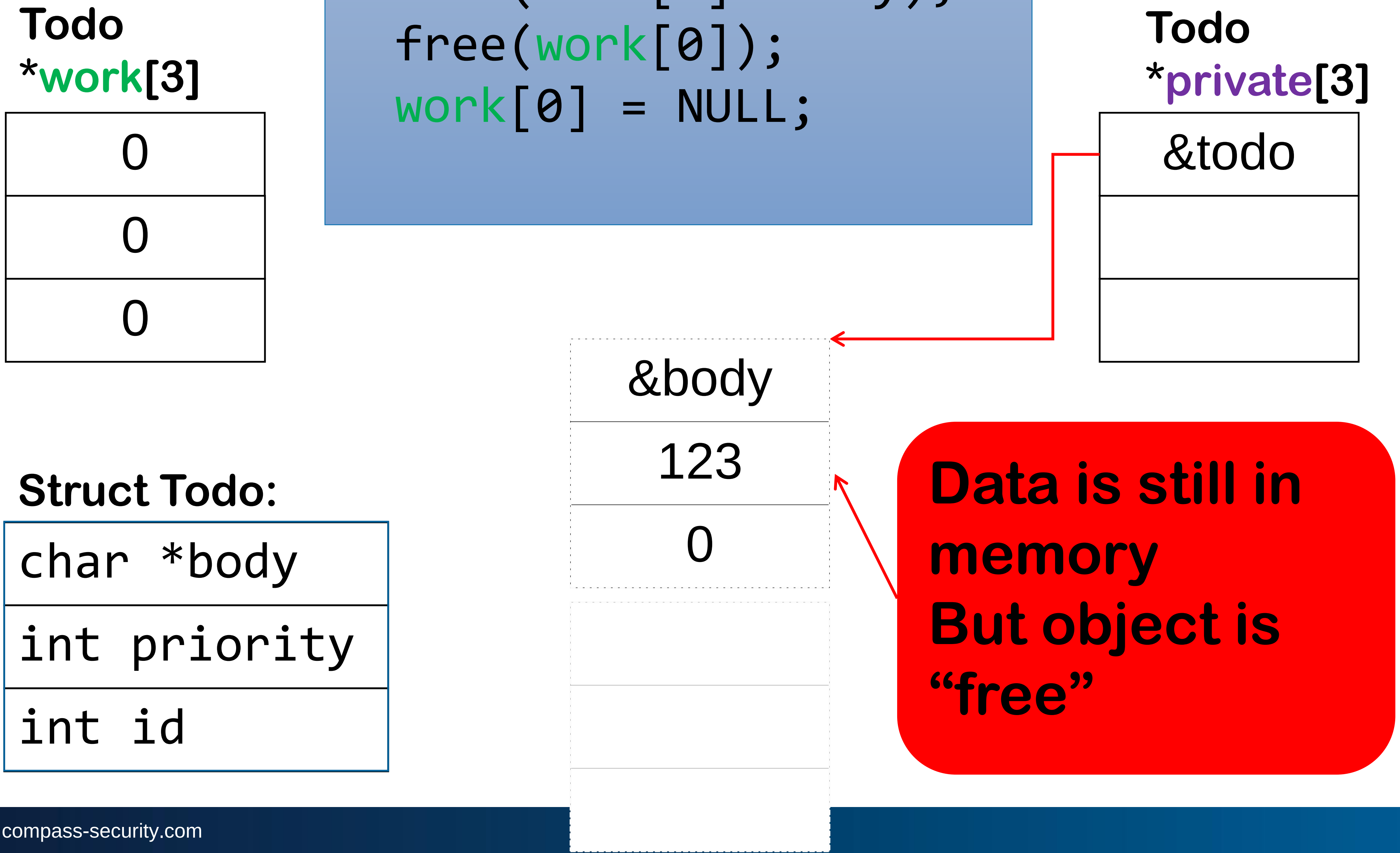
`char *body`
`int priority`
`int id`



```
list_del_work:0
   free(work[0  ->body);
   fre  (work   );
   wo      ]    LL;
```

FAIL!
private[] list still has a pointer to memory region where the object was stored

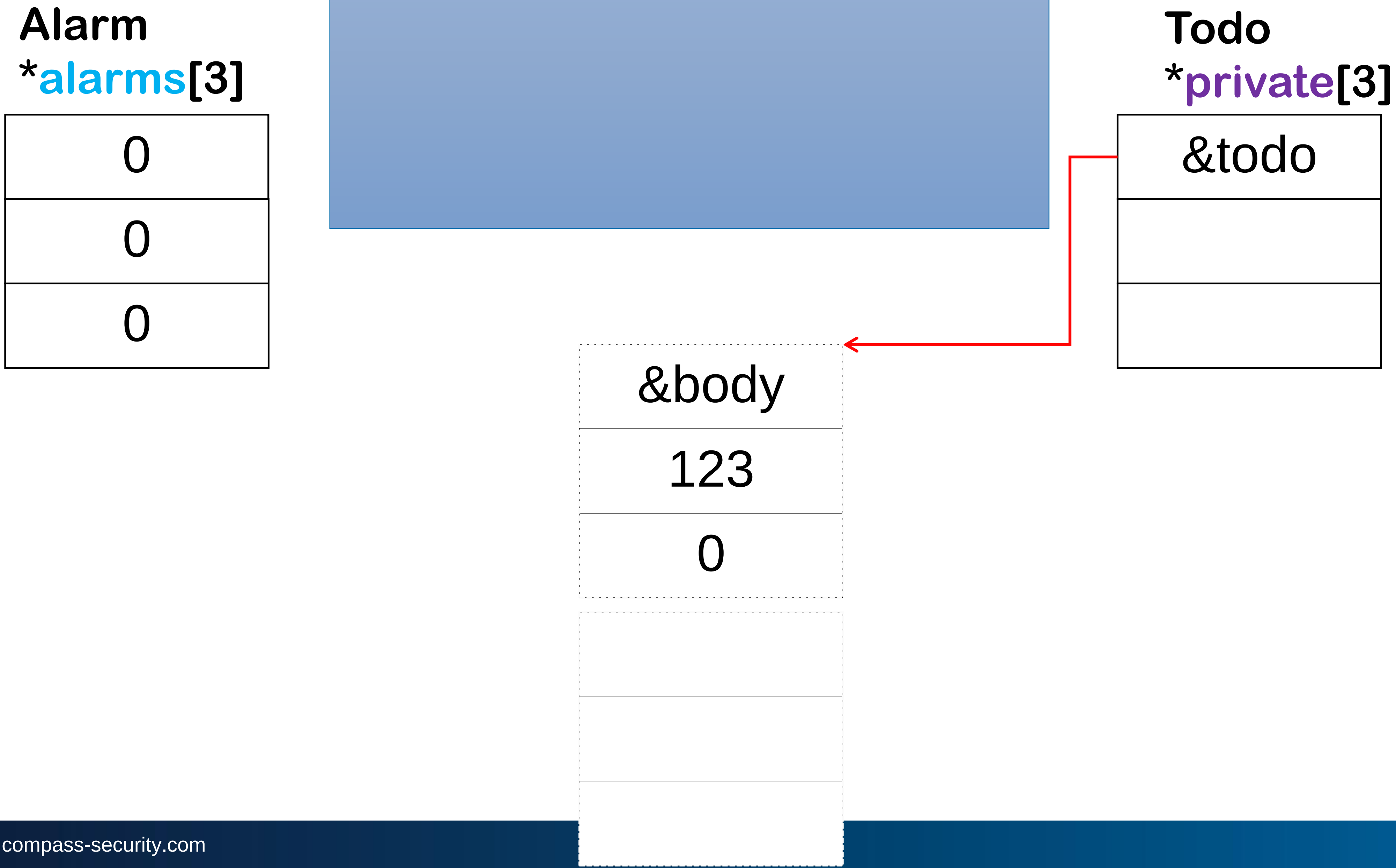
```
list del work:0
   free(work[0]->body);
   free(work[0]);
   work[0] = NULL;
```



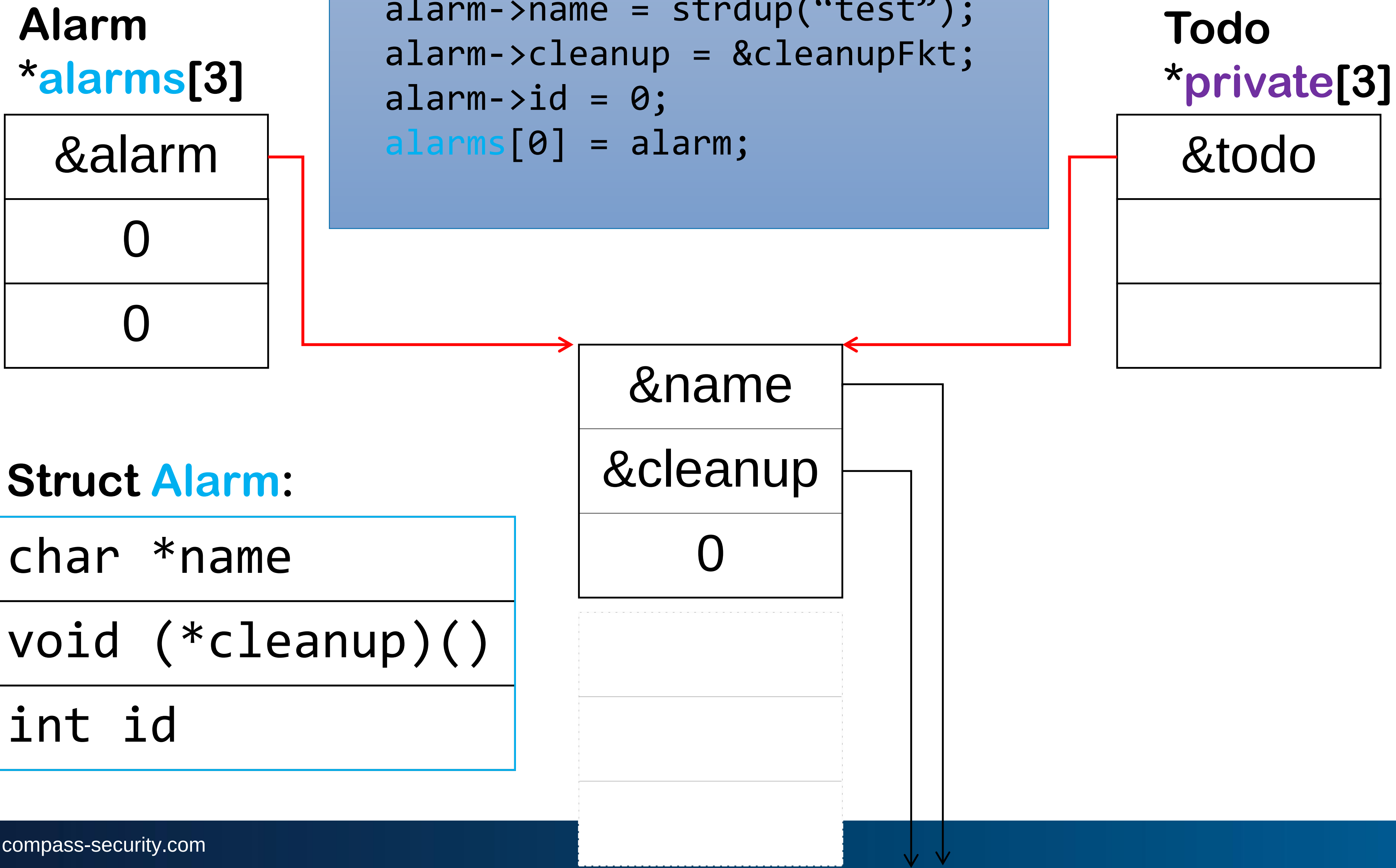
Heap UAF: Noteheap

Step 4: Add an “Alarm”

```
alarm add "test"
```



```
alarm add "test"
  alarm = malloc(sizeof(Alarm));
  alarm->name = strdup("test");
  alarm->cleanup = &cleanupFkt;
  alarm->id = 0;
  alarms[0] = alarm;
```



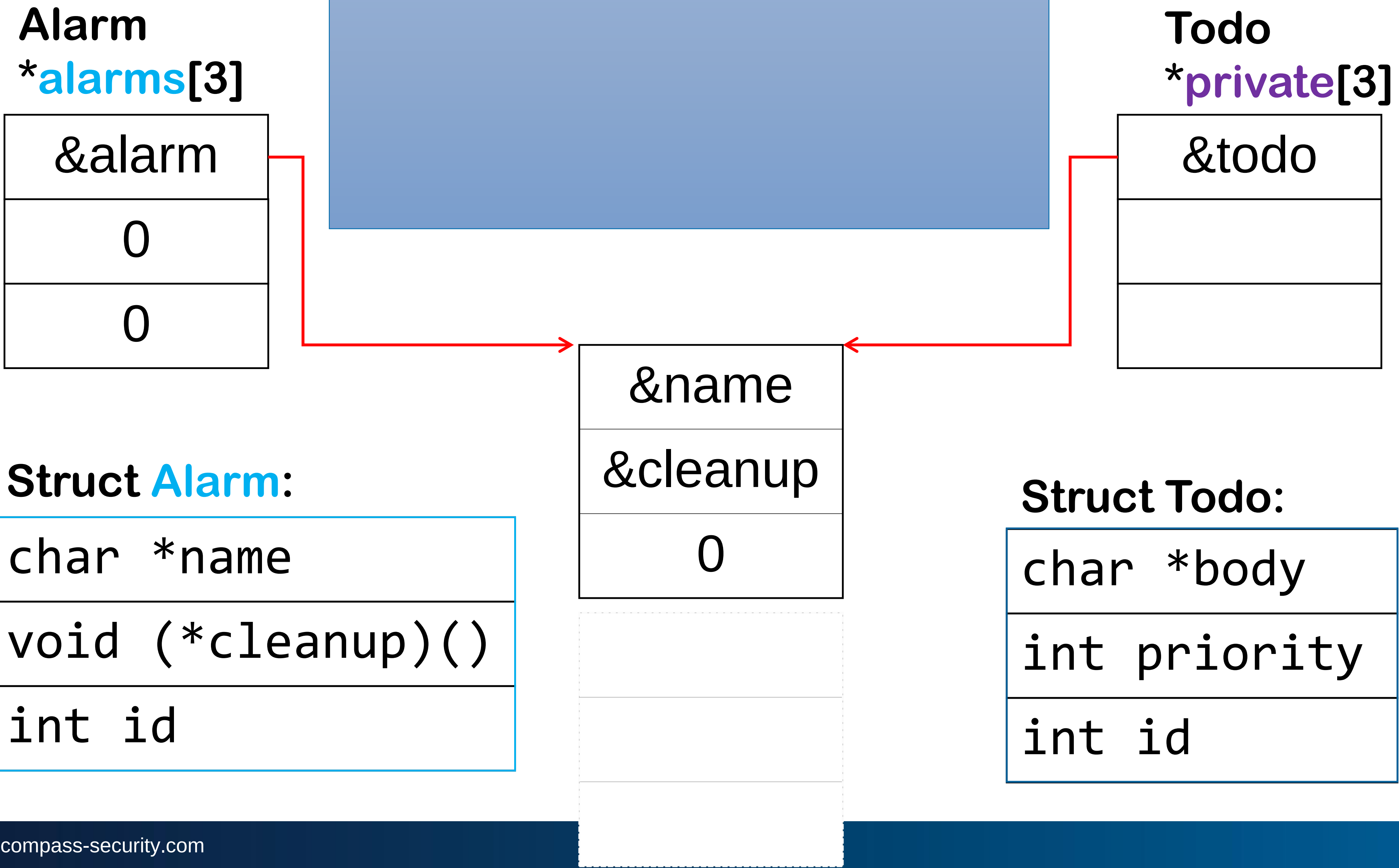
Struct Alarm:

`char *name`
`void (*cleanup)()`
`int id`

Heap UAF: Noteheap

Step 5: Edit the “Todo” (via “private” list)

```
todo edit private:0 456 “AA”
```



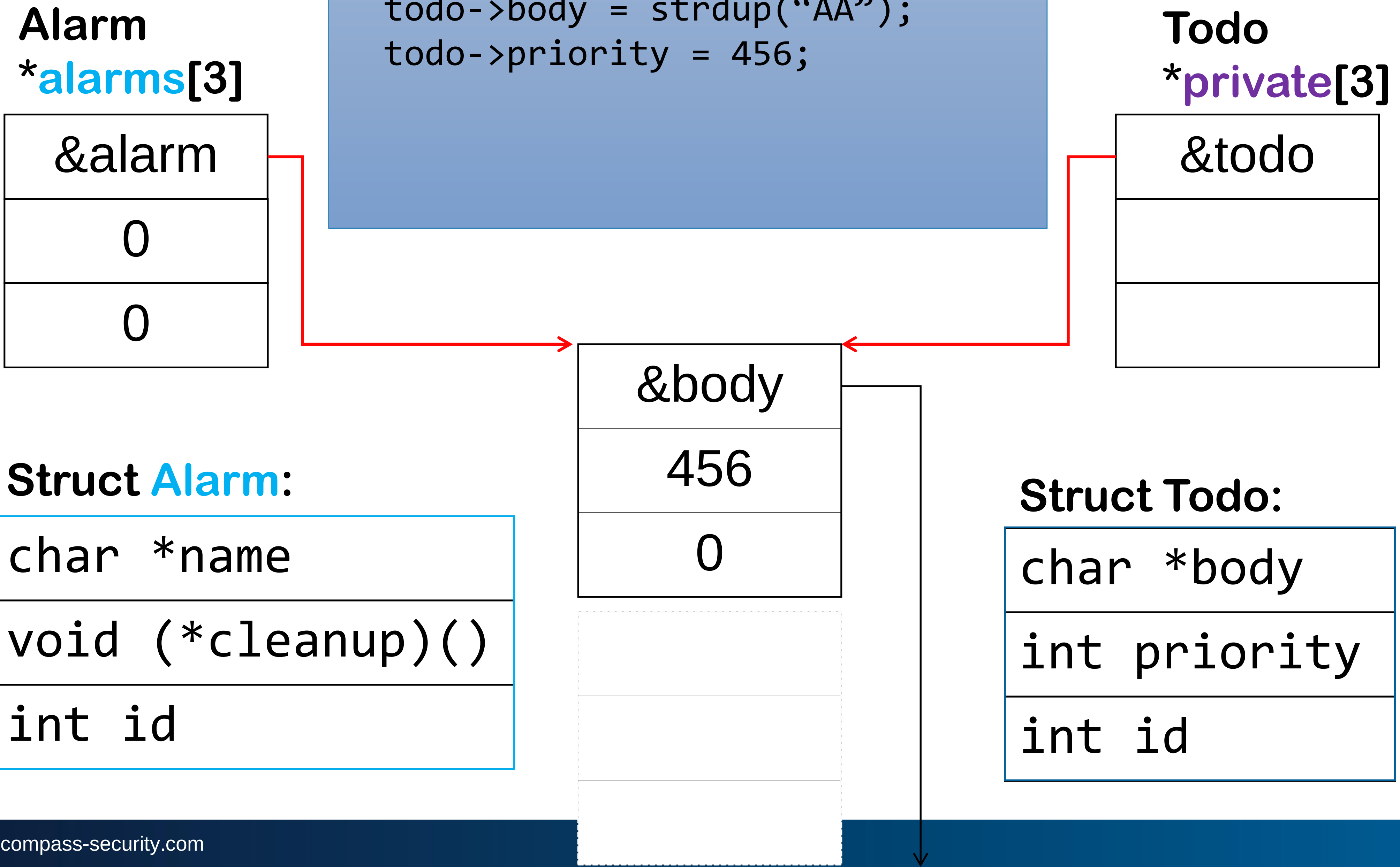
Struct Alarm:

`char *name`
`void (*cleanup)()`
`int id`

Struct Todo:

`char *body`
`int priority`
`int id`

```
todo edit private:0 456 “AA”
   todo = todos[0];
   todo->body = strdup(“AA”);
   todo->priority = 456;
```

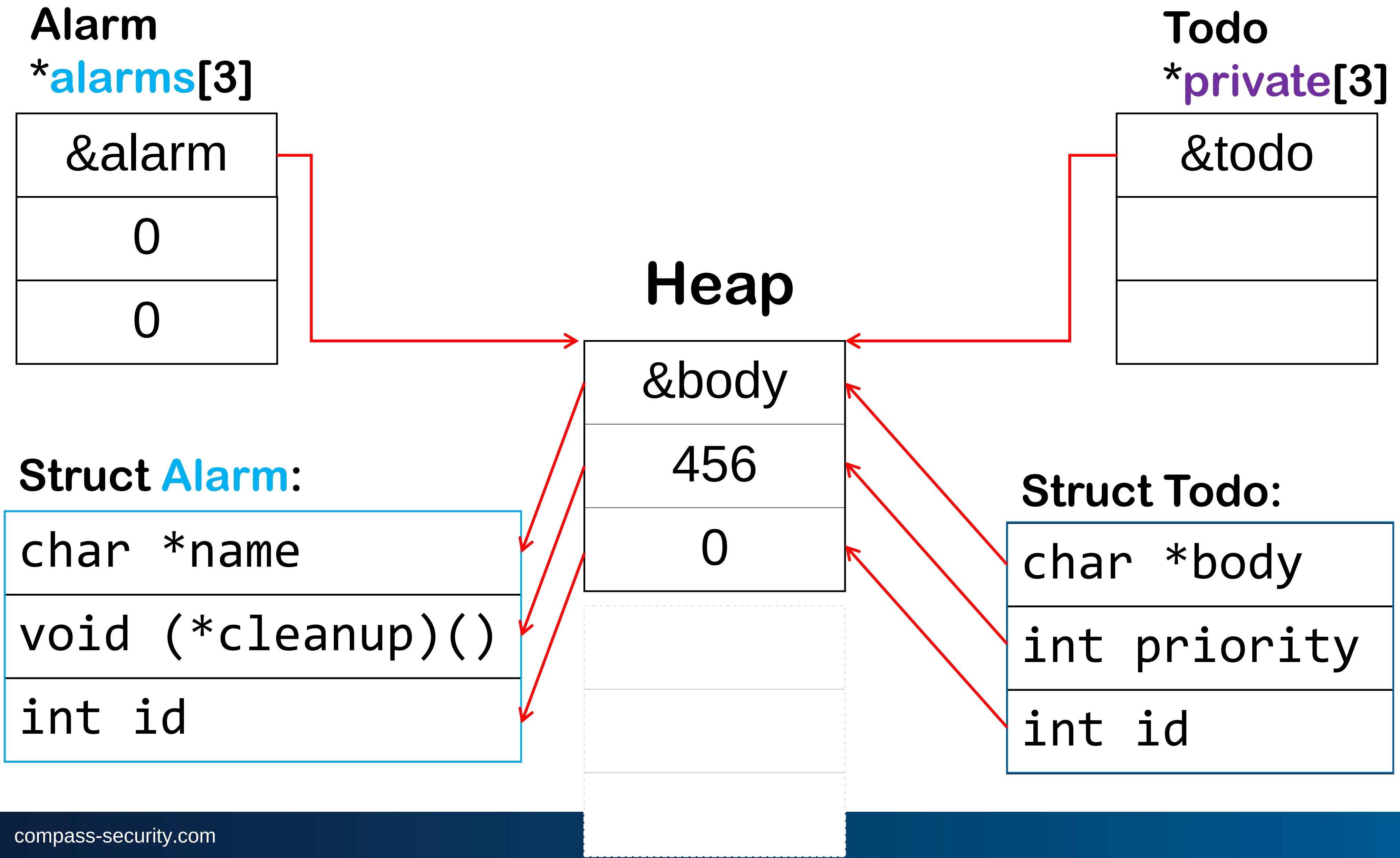


Struct Alarm:

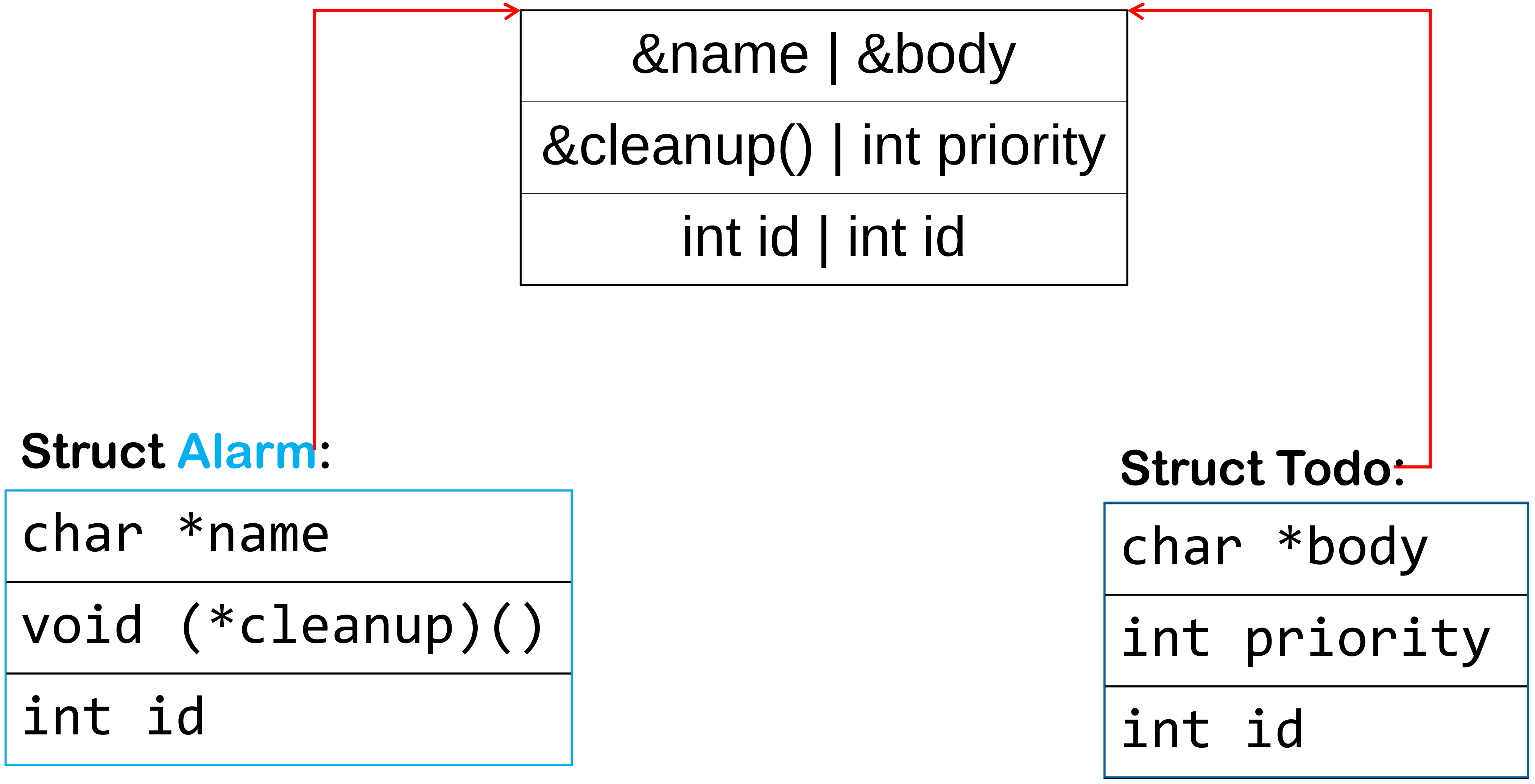
char *name
void (*cleanup)()
int id

Struct Todo:

char *body
int priority
int id


Alarm
*alarms[3]
&alarm
0
0
Todo
*private[3]
&todo
Heap
&body
456
0
Struct Alarm:
char *name
void (*cleanup)()
int id
Struct Todo:
char *body
int priority
int id

Heap


&name | &body
&cleanup() | int priority
int id | int id
Struct Alarm:
char *name
void (*cleanup)()
int id
Struct Todo:
char *body
int priority
int id

Heap UAF: Noteheap

```
todo edit private:0 456 “AA”
  todo = todos[0];
  todo->body = strdup(“AA”);
  todo->priority = 456;

did the same as:
  alarm = alarms[0];
  alarm->name = strdup(“AA”);
  alarm->cleanup = 456;
```

Heap UAF: Noteheap

Result:

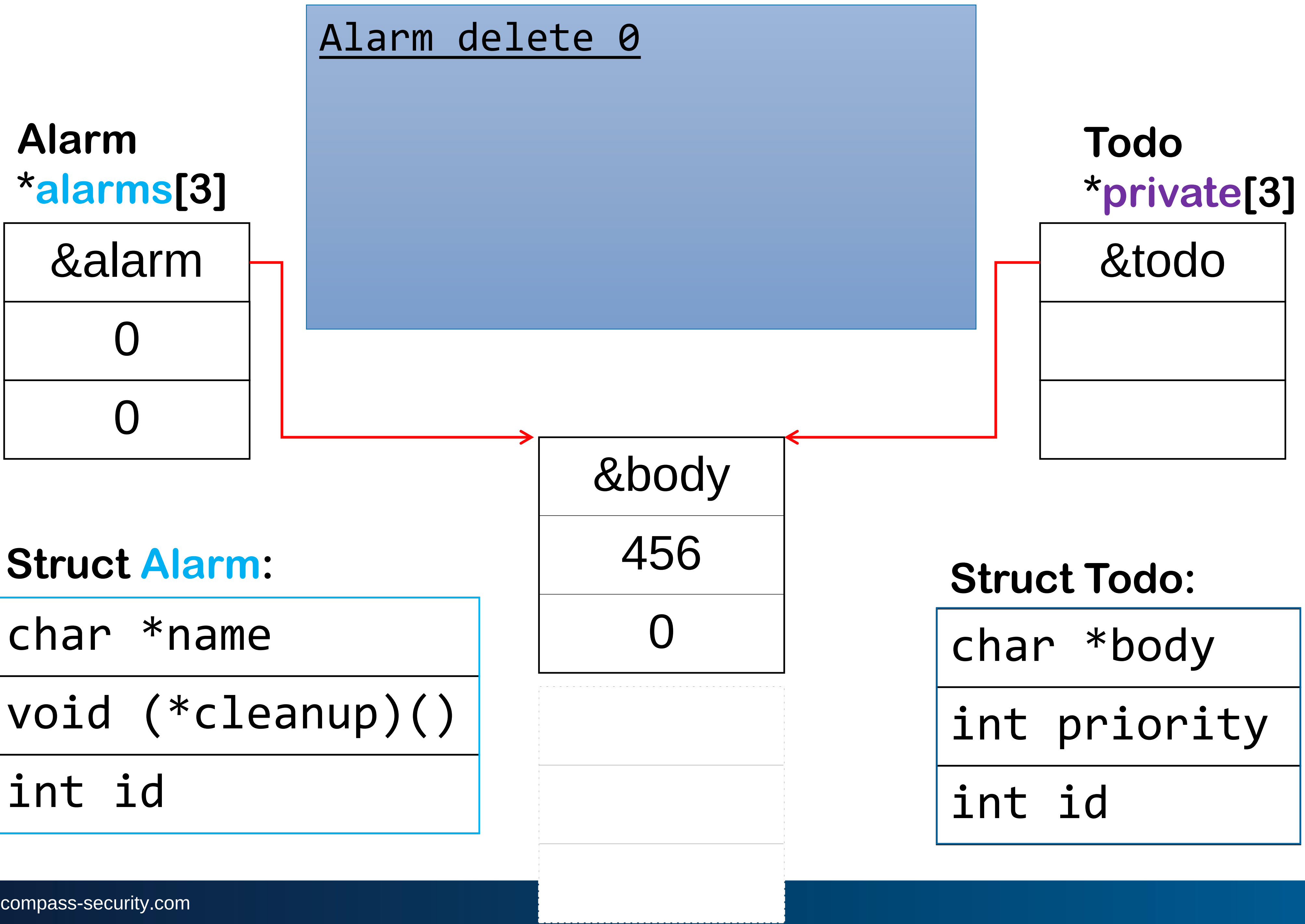
- We allocated a “Todo” object
- We had two references to this “Todo” object: in “work” and “private” list
- We free’d the “Todo” object, and removed the reference in “work” list
- BUT: We still have a reference to the “Todo” object in the “private” list

- We allocate an “Alarm” object
- The “Alarm” object was allocated where the initial “Todo” object was
- We still have a pointer to the initial “Todo” object via the “private” list
- If we modify the initial “Todo”, we change the “Alarm” object

- Therefore: We can modify the function pointer in the a”Alarm” object

Heap UAF: Noteheap

Step 6: Delete the Alarm object



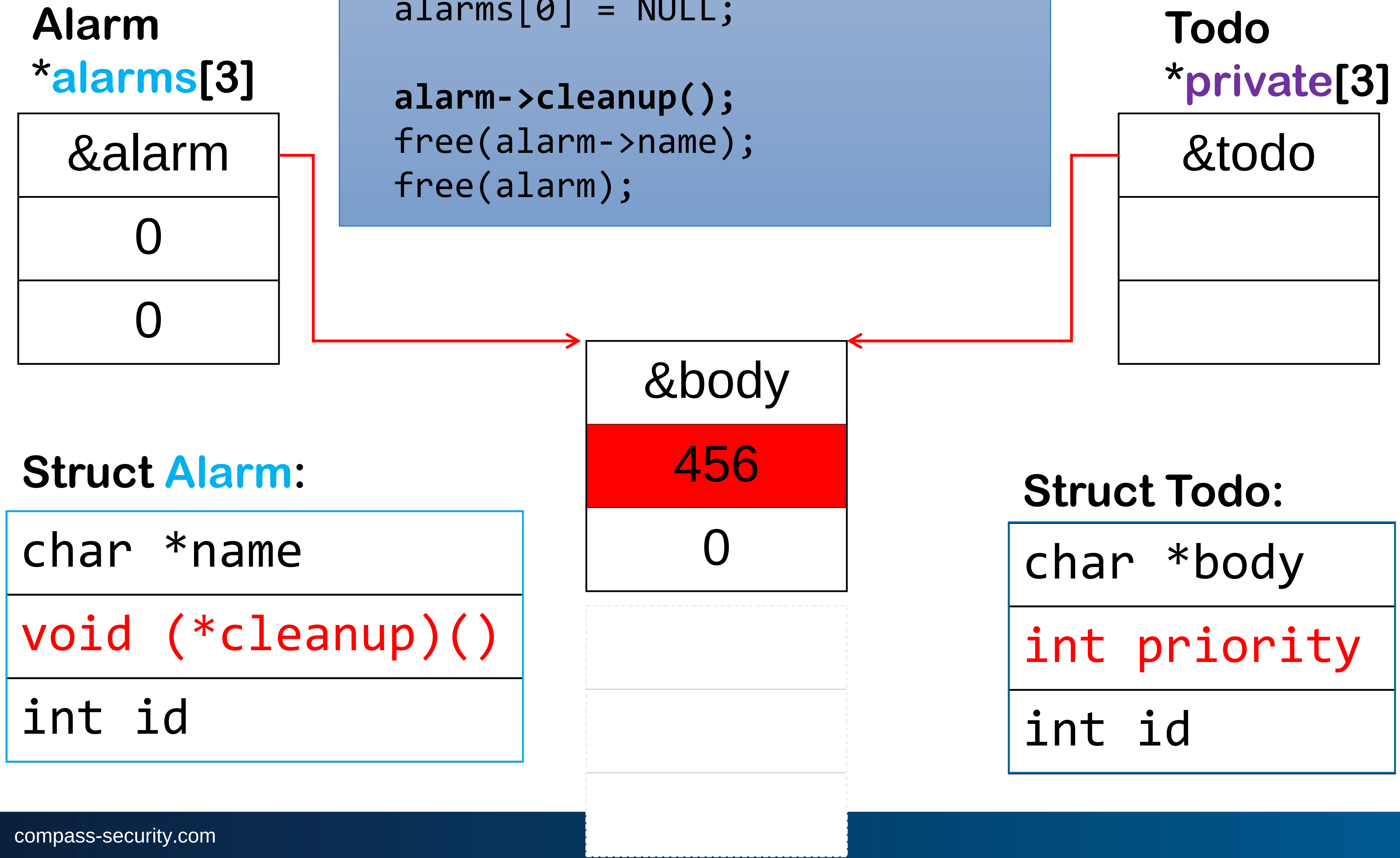
Struct Alarm:

char *name
void (*cleanup)()
int id

Struct Todo:

char *body
int priority
int id

```
Alarm delete 0
  alarm = alarms[0];
  alarms[0] = NULL;

  alarm->cleanup();
  free(alarm->name);
  free(alarm);
```



Heap UAF: Noteheap

The program is calling alarm->cleanup()

We can define where alarm->cleanup is pointing to

Therefore: Can call any memory location (continue code execution where we want it)

Heap Attack: UAF

So, what is UAF?

- We have a pointer (of type A) to an object
- The object get’s free()’d
  - This means that the memory allocater marks the object as free
  - The object will not be modified!
  - (Similar to deleting a file on the harddisk)
  - The pointer is still valid
- Another object of type B (of the same size) get’s allocated
- Memory allocator returns the previously free’d object memory space

- Attacker has now a pointer (type A) to another object (type B)!
- This object can be modified
  - Depending on the types A and B
  - Can modify pointers, sizes etc.

Object Oriented Languages

vtables

Object Oriented Languages

Dobin: *“OO ist just some fancy C structs with function pointers”*

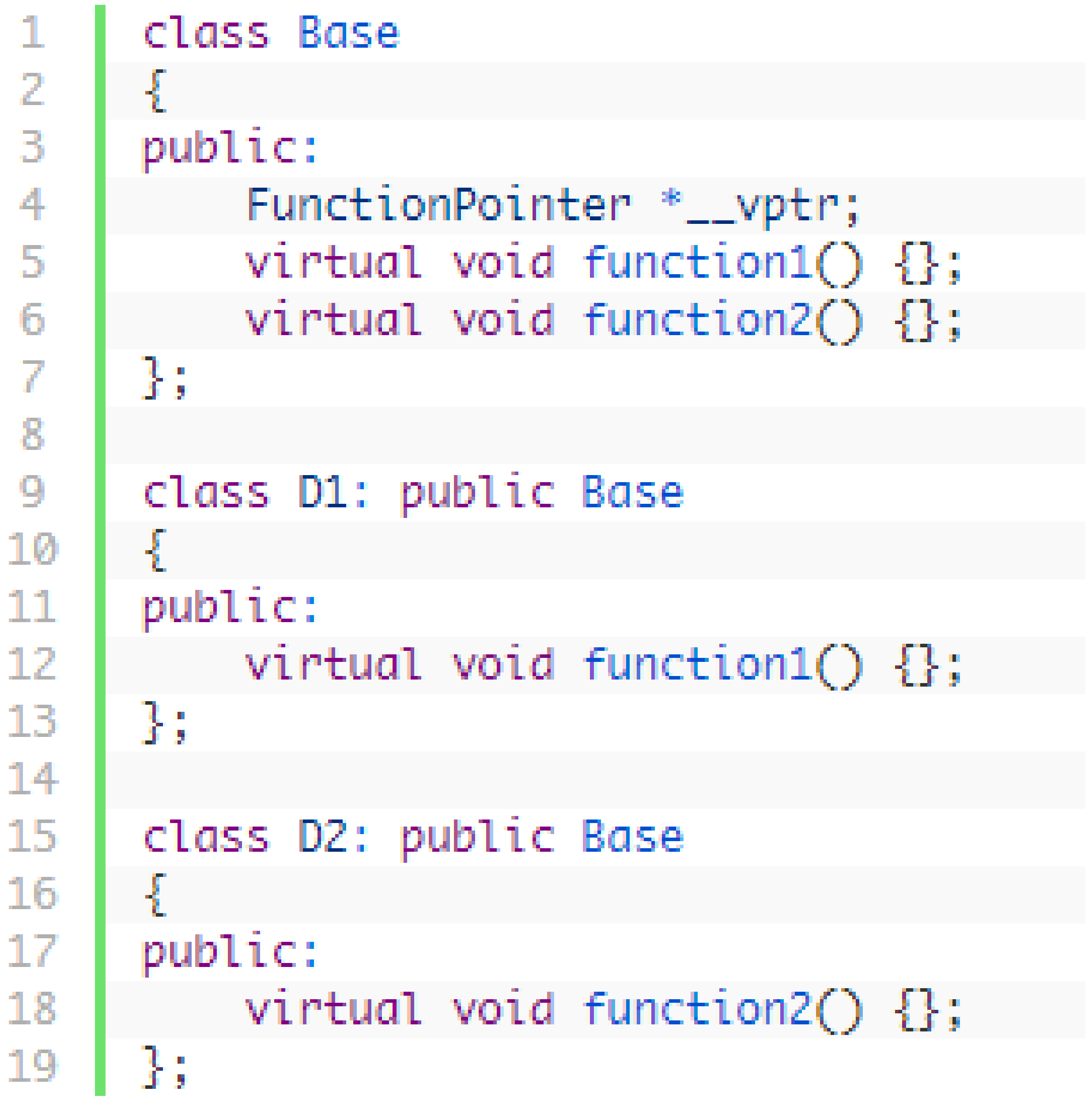
OO in C:

```
typedef struct animal {
      int (*constructor)(void *self);
  int (*write)(void *self, void *buff);
  void *data;
} AnimalClass;

AnimalClass animal;
animal.constructor = &constructor;
animal.data = malloc(…);
…
animal.constructor(&animal);
```

Object Oriented Languages

C++ vtables

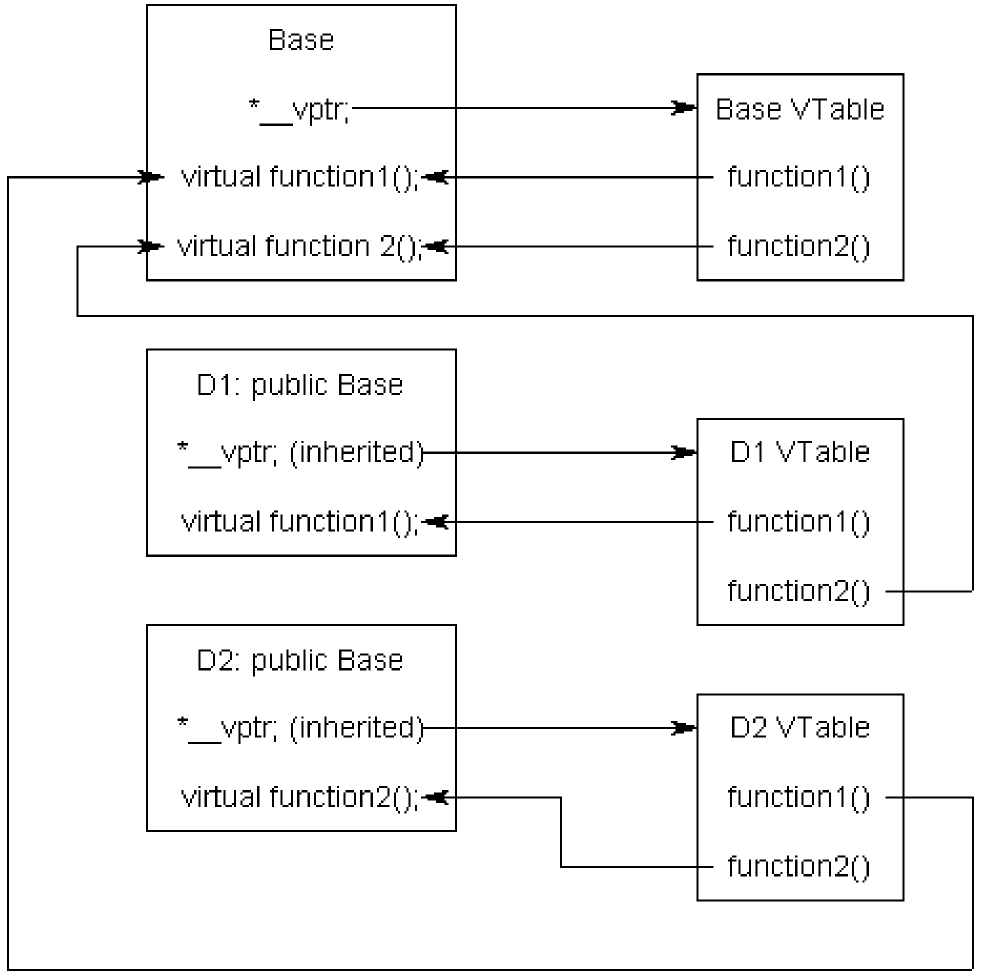
The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.



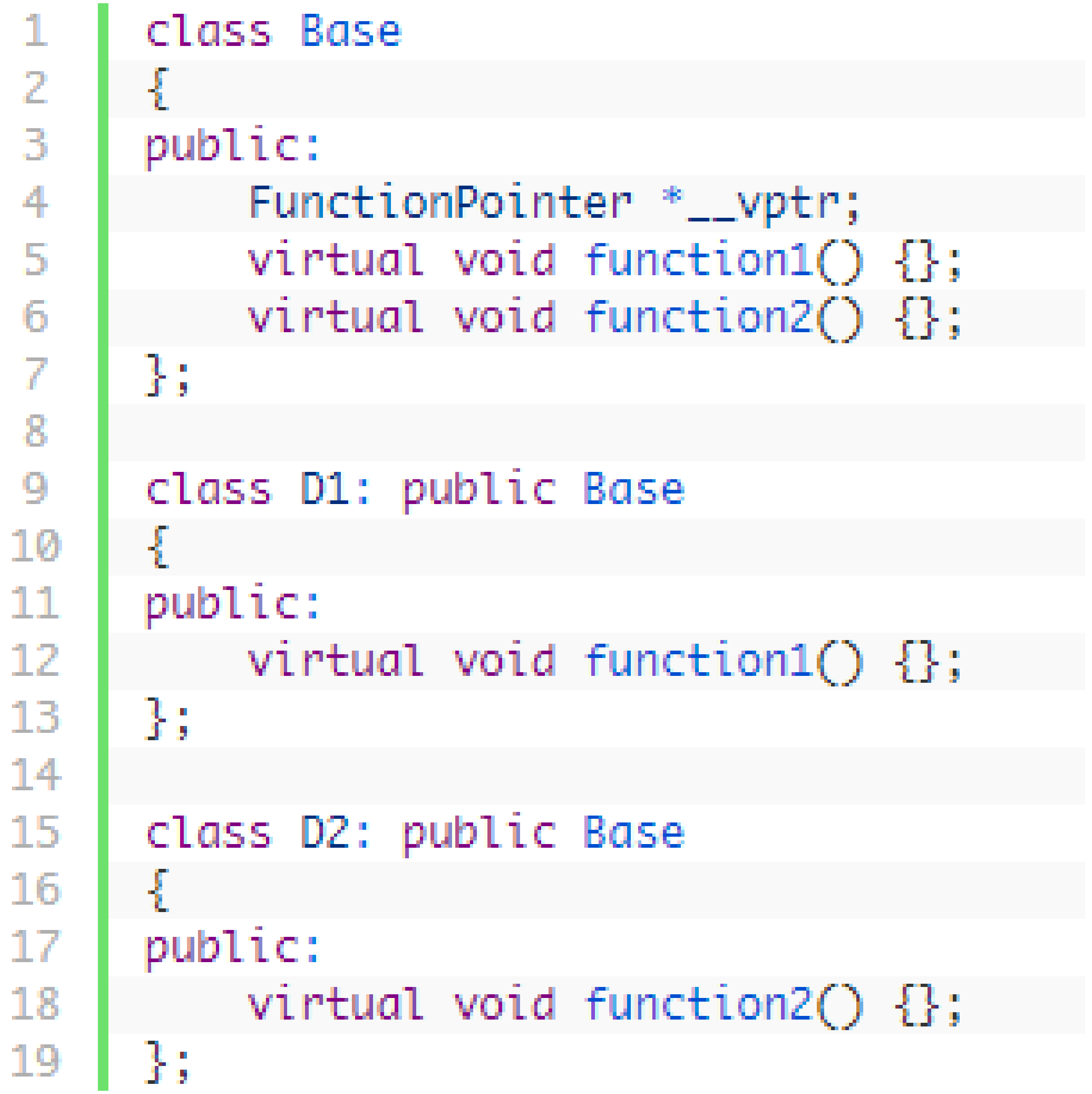
```
class Base
{
public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

Object Oriented Languages

C++ vtables

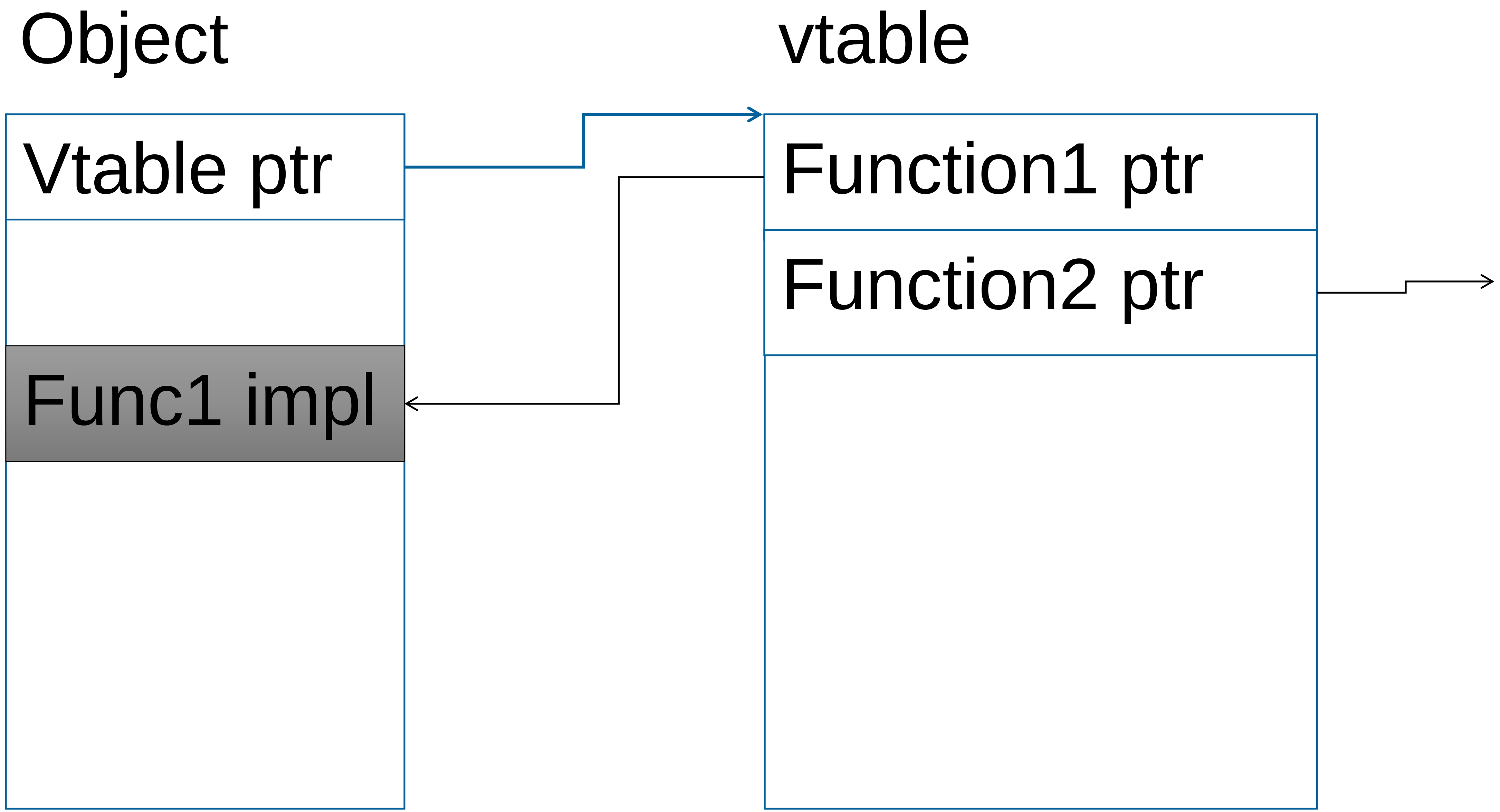


http://www.learncpp.com/cpp-tutorial/125-the-virtual-table/



```
class Base
{
public:
    FunctionPointer *__vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

Vtables


Object
vtable
Vtable ptr
Function1 ptr
Function2 ptr
Func1 impl

Object Oriented Languages

Recap:

- OO languages heavily use function pointers
- C++ use vtables
  - First element of object struct is pointer to vtable
  - Vtables is an array of pointers to the appropriate functions
- OO is therefore particulary affected by UAF

Garbage Collection

Garbage Collection

Dobin: *“Garbage collection is just fancy structs with reference counter”*

```
typedef struct animal {
        int (*constructor)(void *self);
  int (*write)(void *self, void *buff);
  void *data;
        int refCount;
} AnimalClass;

AnimalClass animal;
animal.refCount = 0;
…
Animal animal2 = &animal;
Animal.refCount++;
```

Garbage Collection

Objects keep track on how many references are to them

A separate thread (garbage collector) regularly checks the references on objects

Garbage collector free’s objects if they are not needed anymore (similar to a manual free)

Garbage Collection

Recap:

- Garbage collector periodically free’s unused objects

ROP: Stack Pivoting

ROP: Stack pivoting

At an UAF:

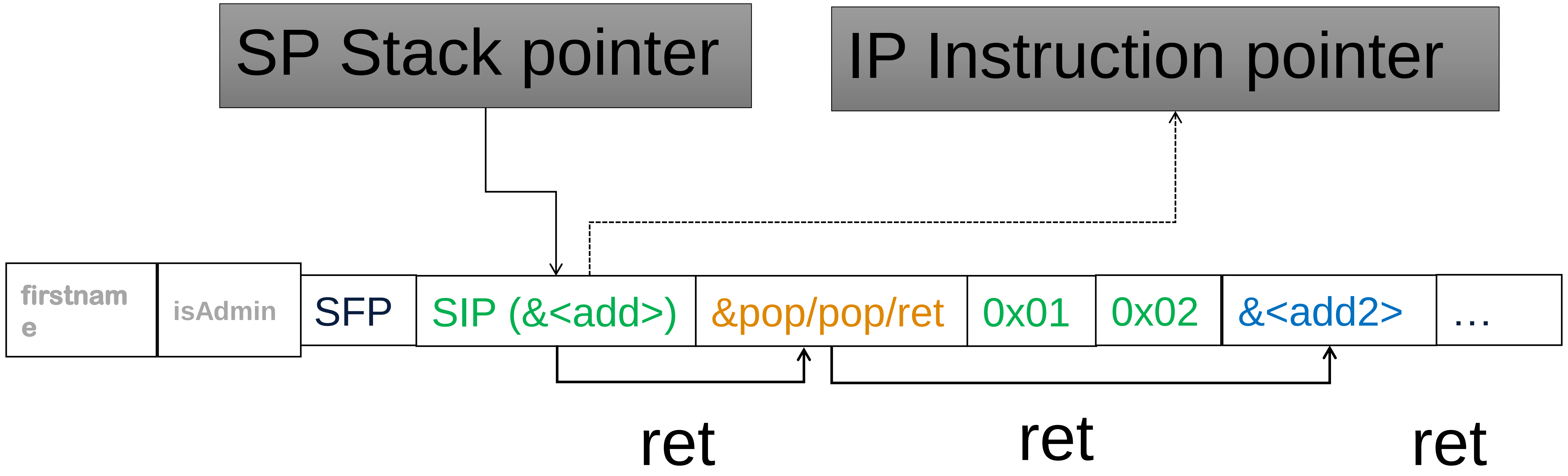
Ok, we can call any function in memory (e.g. via alarm->cleanup())

What we want: Execute ROP chain

- Problem:
  - We can call() any function
  - But the stack pointer is not modified (unlike in a Stack based overflow)

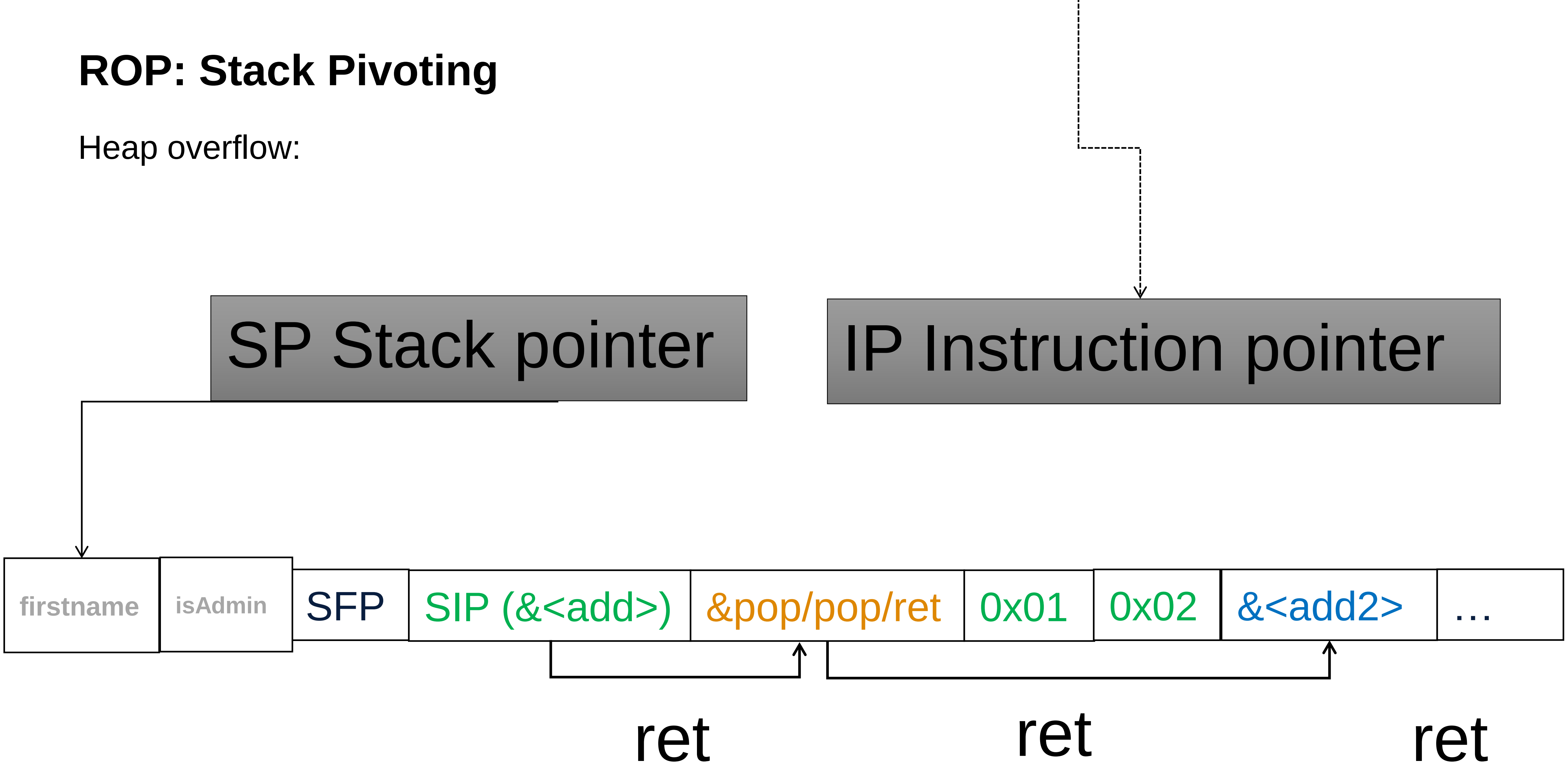
ROP: Stack Pivoting

Remember: Stack overflow



ROP: Stack Pivoting

Heap overflow:



ROP: Stack pivoting

Stack exploit:

- Overwrite SIP
- On return():
  - pop EIP from ESP (get next instruction pointer from stack)
  - Do stuff…
  - pop EIP from ESP (get next instruction pointer from stack)

Heap exploit:

- Overwrite function pointer
- On call():
  - Get next instruction from the function pointer (heap -> EIP)
  - Do stuff…
  - pop EIP from ESP (get next instruction pointer from stack)
    - ESP points to user data
    - CRASH

ROP: Stack pivoting

Solution: Stack pivoting

Example stack pivot gadget:

```
mov esp, eax
```

- Precondition:
  - EAX points to memory location we control
- After this gadget is executed:
  - We have a “new stack” (at EAX location)
  - SIP will be “taken from EAX” (memory location where EAX points to)

Other examples:

```
xchg esp, eax
add esp, 0x40c
```

ROP: Stack pivoting

Stack pivoting recap:

- Gadgets use RET
- RET takes next IP from stack (SIP@ESP -> EIP)
- It can be necessary to move ESP (stack pointer) so a memory location we control

Other Heap attacks…

Heap Massage / Feng shui

Heap Massage

For attacks to work, the heap needs to be in a predictable state

- Allocation of objects:
  - In place of an existing pointer (UAF)
  - Close to each other (inter-chunk overflow)
  - Beginning/End of a BIN (inter-chunk overflow)

Heap massage

Solution:

- Heap massage / heap grooming / heap feng-shui

Allocate/Deallocate objects before (and during) the exploit to put the heap in a predictable state

Objective:

- Allocations should put the allocated chunks in a specific order
- E.g.: inter-chunk overflow
  - Put a chunk to free “on top” of the chunk to overflow

Heap massage

Example:

Allocate 10’000 chunks of 64 byte size

Free one

- Perform overflow
  - Allocate a vulnerable chunk
  - Overflow into the next chunk

Free() all other 99’999 chunks

Profit!

Conclusion

Heap Attacks: Conclusion

Heap-based attacks are very powerful

They are currently state-of-the-art