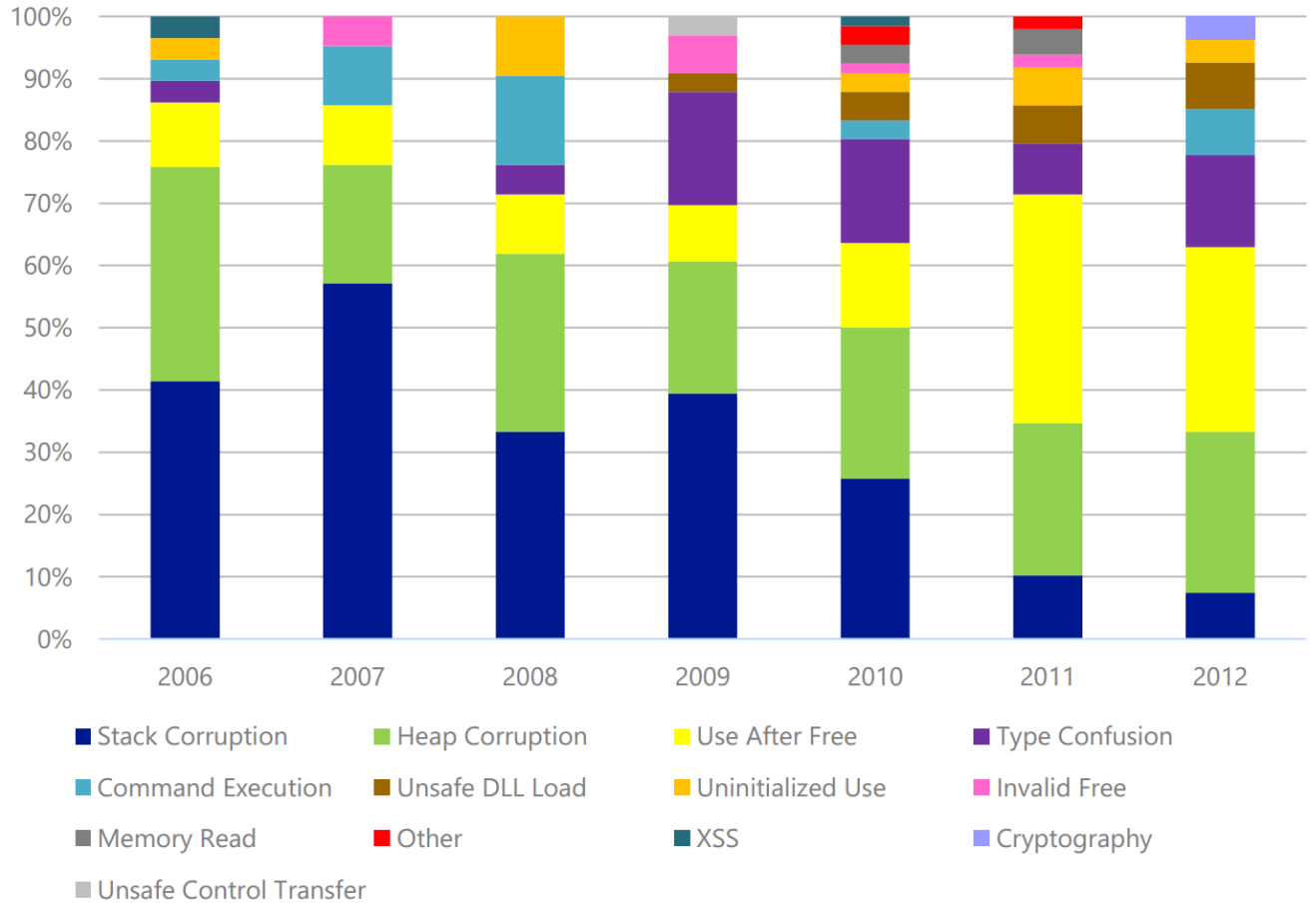
Preventing Use-after-free with Dangling Pointers Nullification

Byoungyoung Lee, Chengyu Song, Yeongjin Jang
Tielei Wang, Taesoo Kim, Long Lu, Wenke Lee

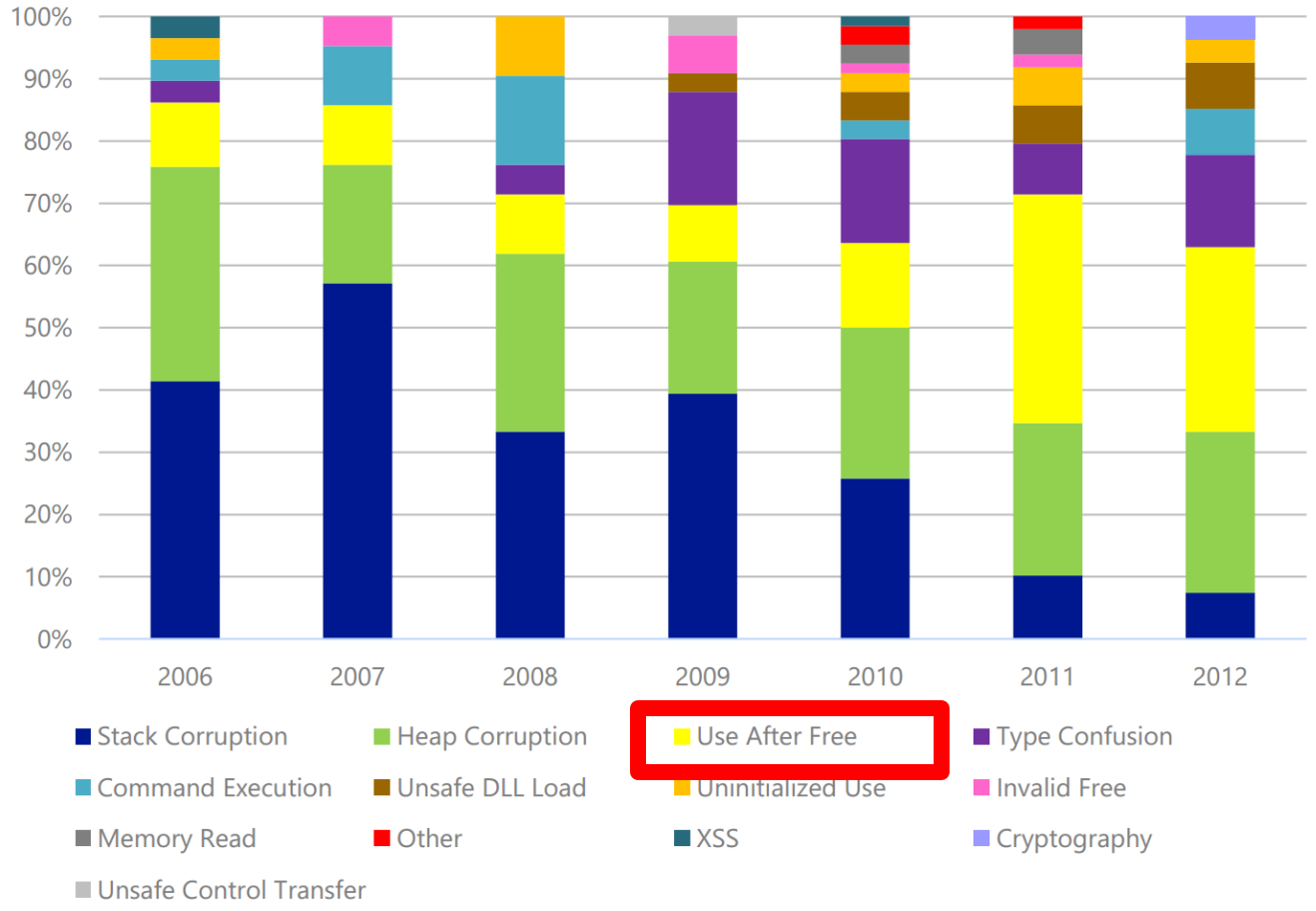
Georgia Institute of Technology
Stony Brook University

Emerging Threat: Use-after-free



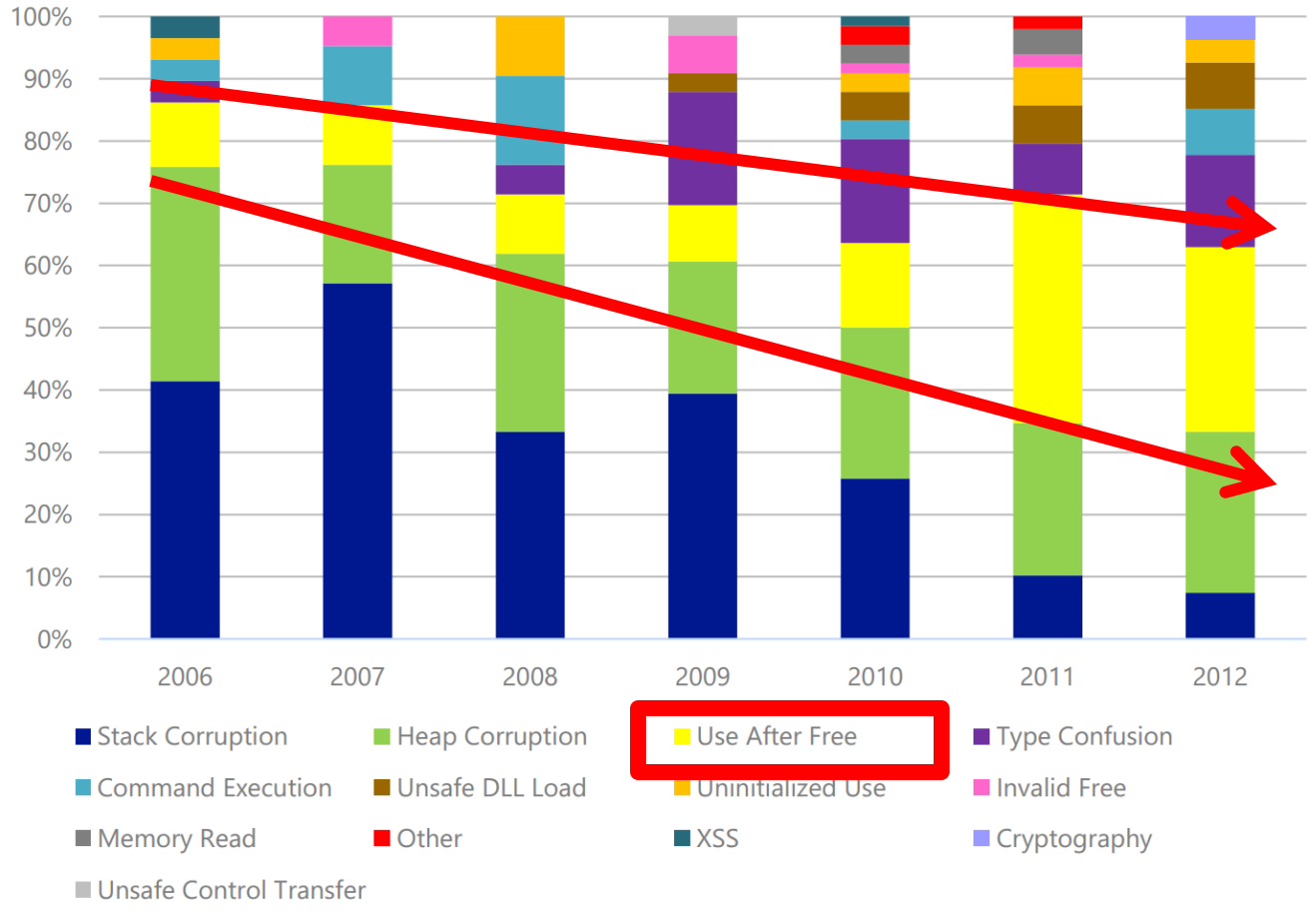
Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free



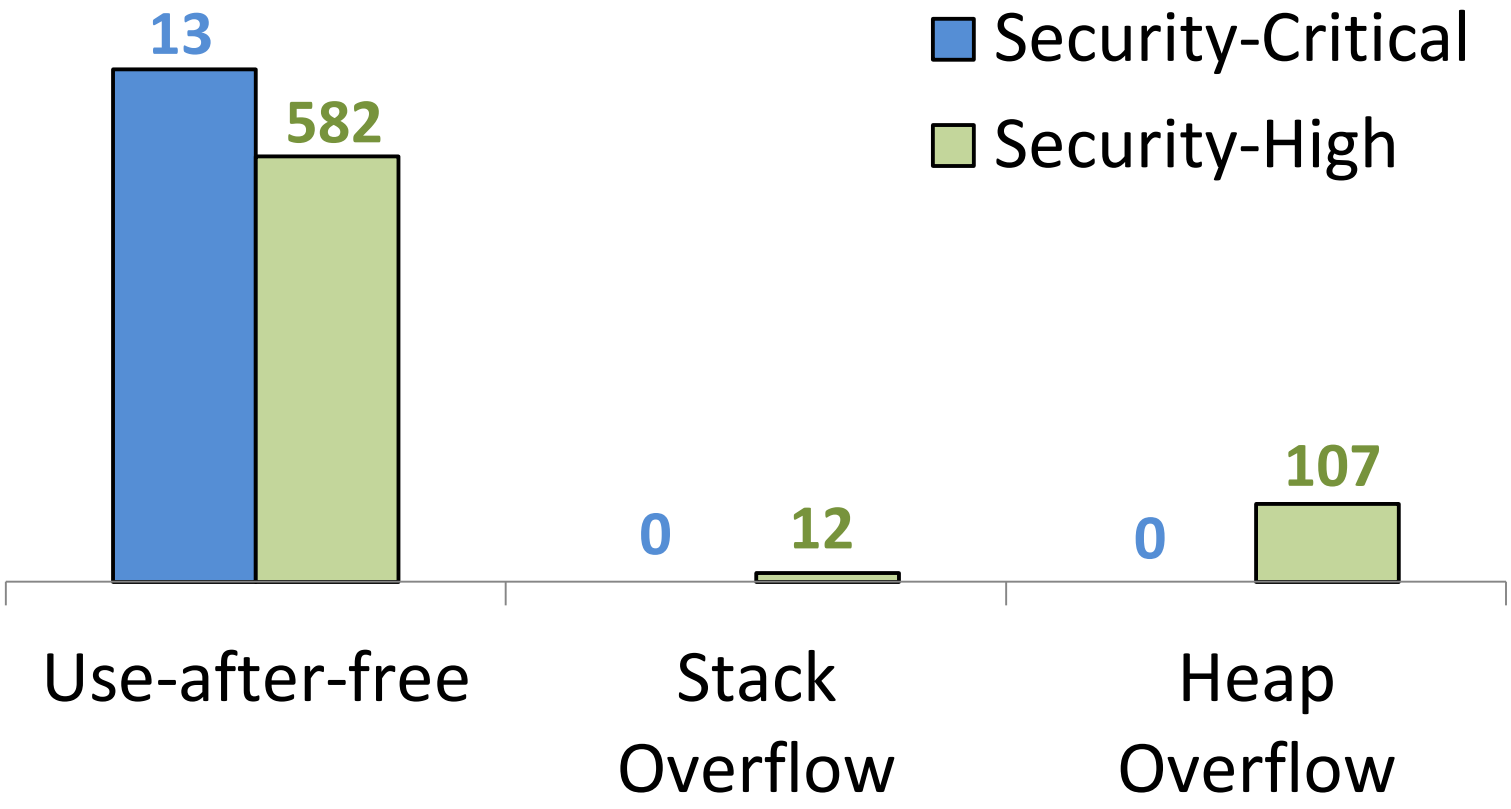
Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free



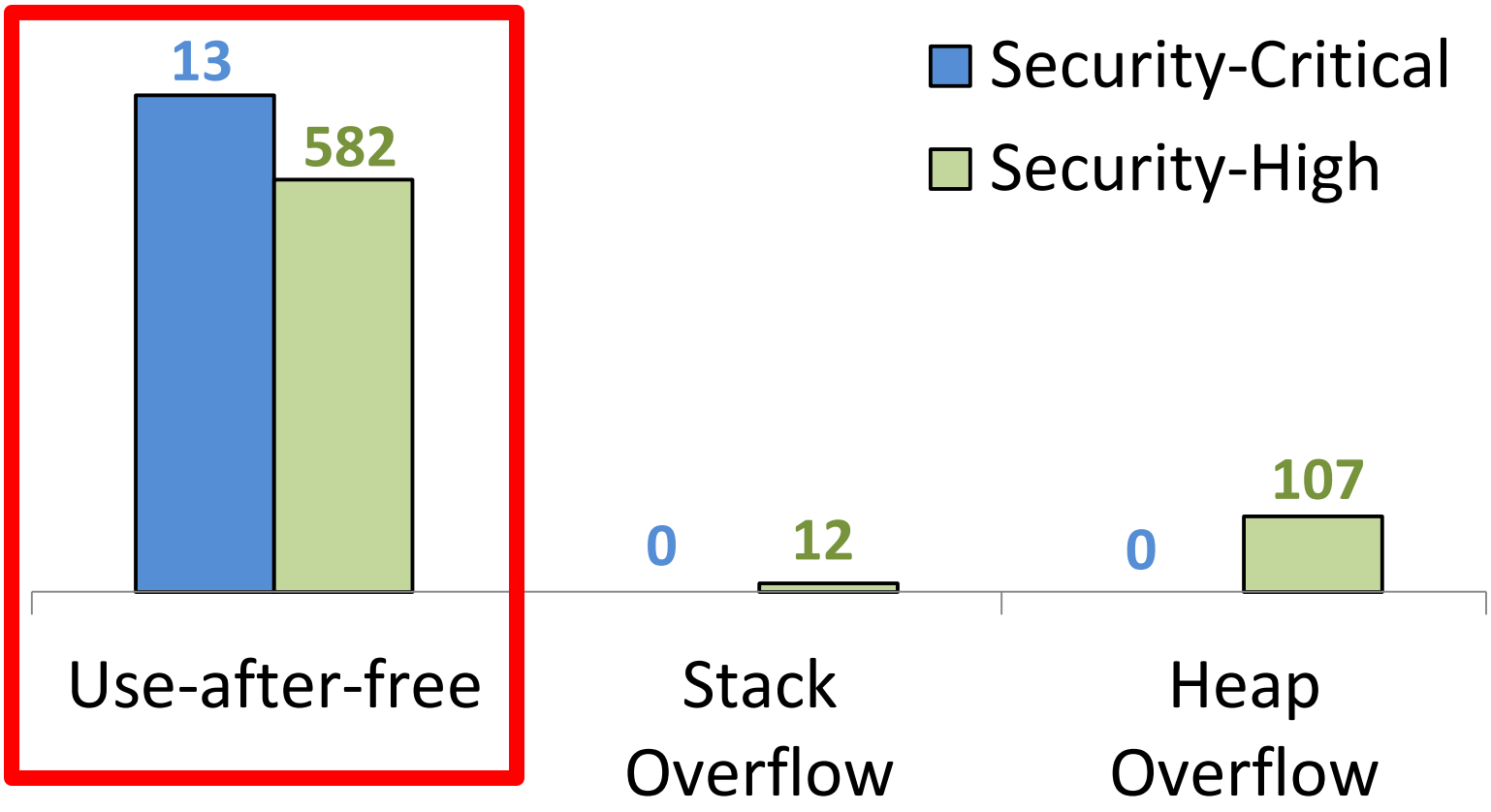
Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free



The number of reported vulnerabilities in Chrome (2011-2013)

Emerging Threat: Use-after-free



The number of reported vulnerabilities in Chrome (2011-2013)

Use-after-free

- A dangling pointer
  - A pointer points to a freed memory region
- Using a dangling pointer leads to undefined program states
  - Easy to achieve arbitrary code executions
  - so called use-after-free

Understanding Use-after-free

```
class Doc : public Element {
    // ...
    Element *child;
};

class Body : public Element {
    // ...
    Element *child;
};
```

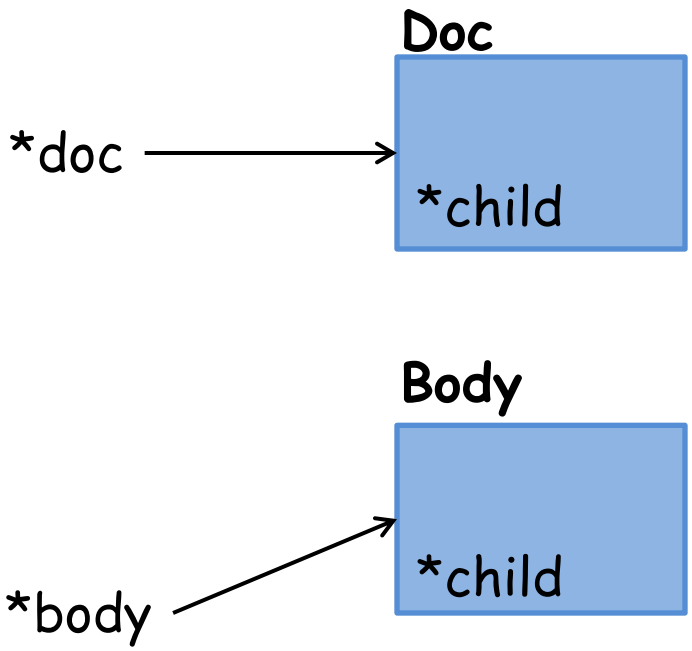
```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

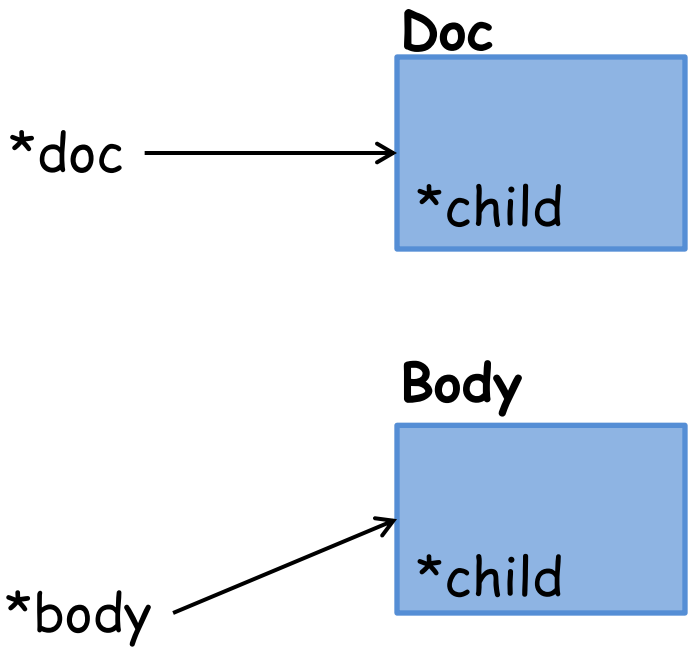
```
Doc *doc = new Doc();
Body *body = new Body();
```

```
doc->child = body;
```

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

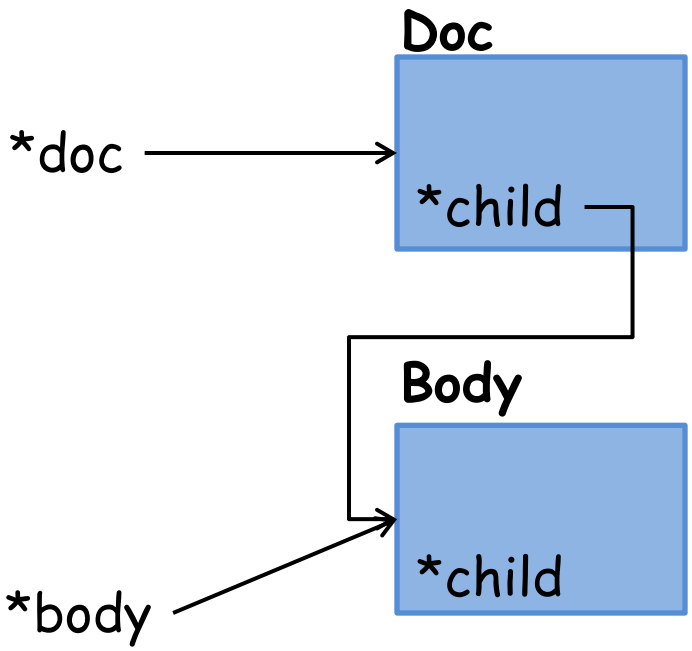
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

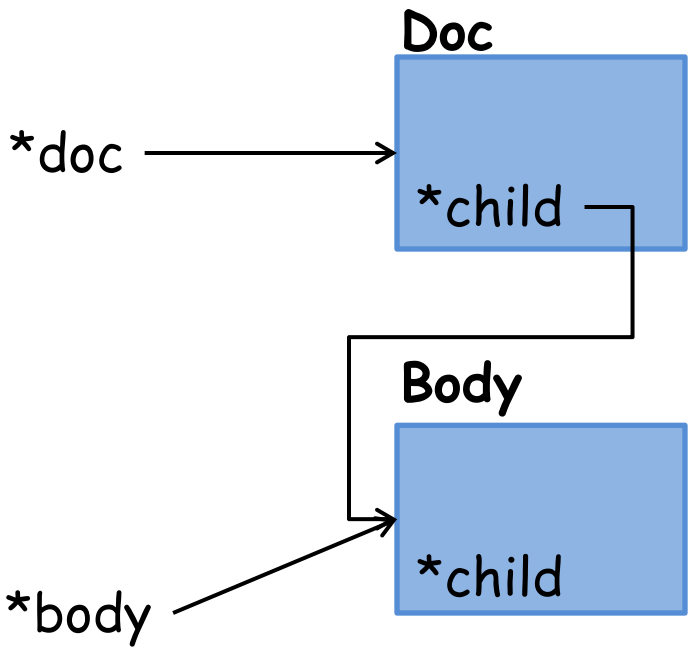
Propagate pointers

```
doc->child = body;
```

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

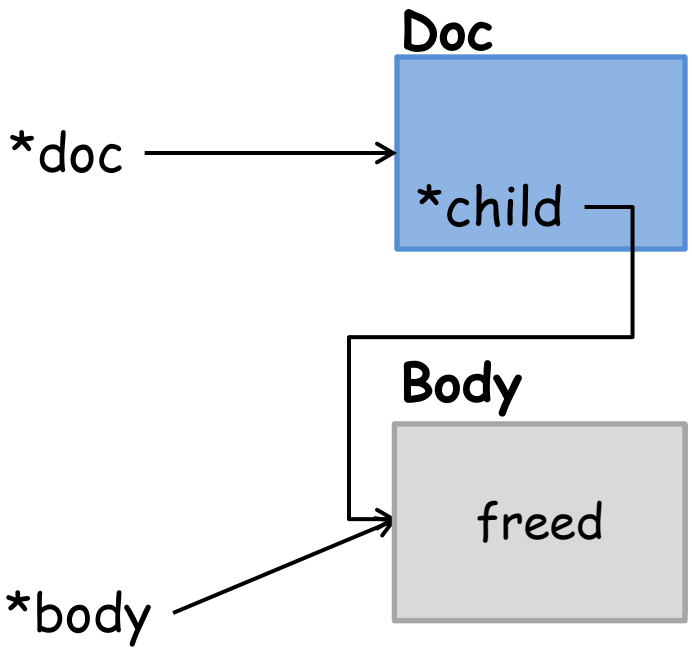
```
doc->child = body;
```

Free an object

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

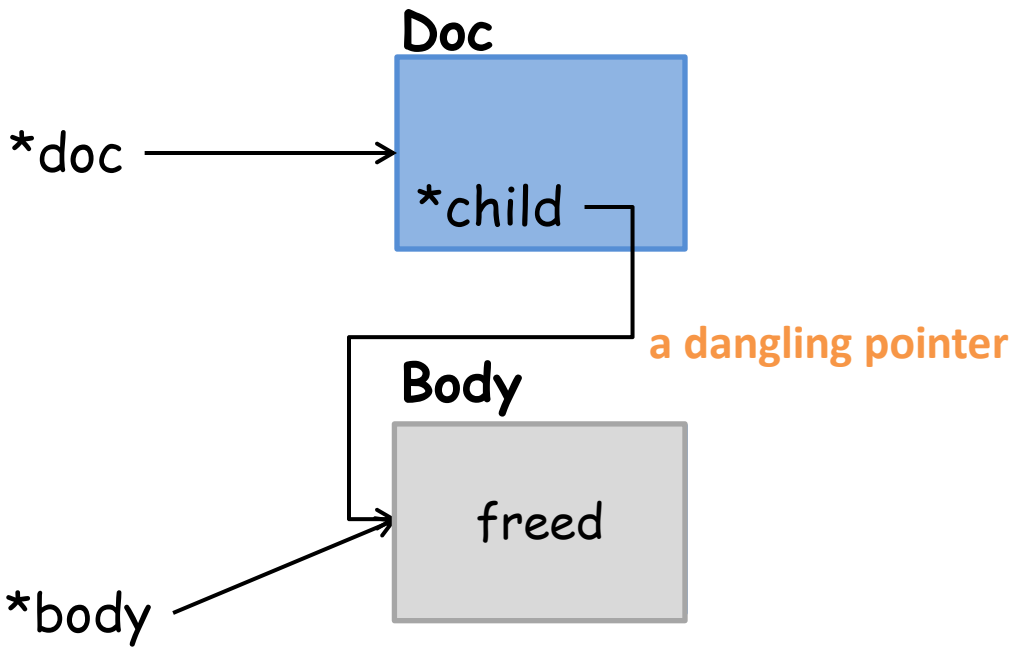
```
doc->child = body;
```

Free an object

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

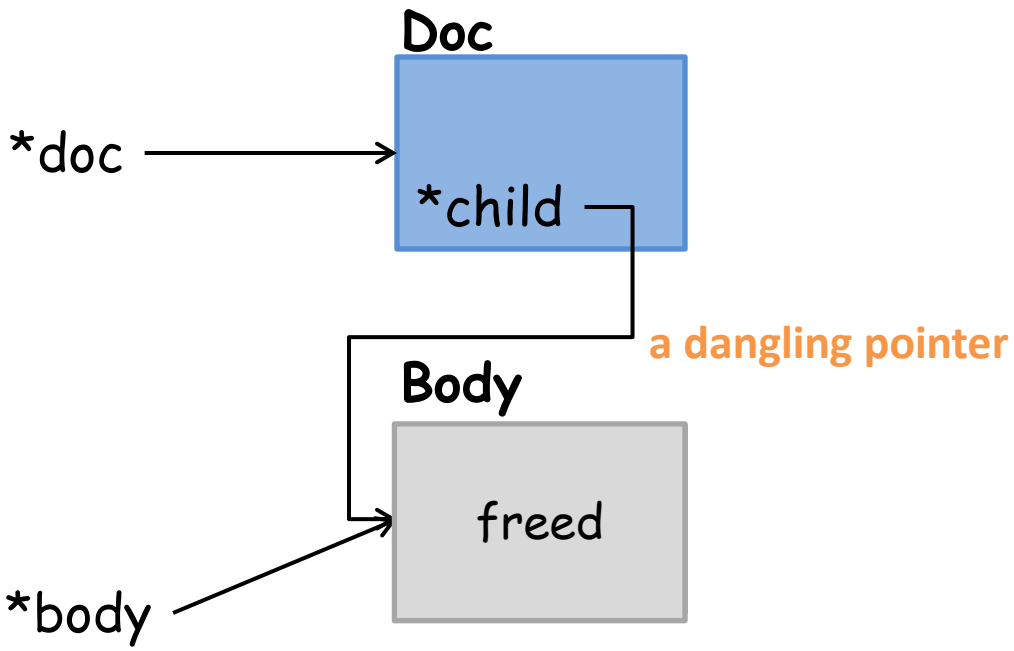
```
doc->child = body;
```

Free an object

```
delete body;
```

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

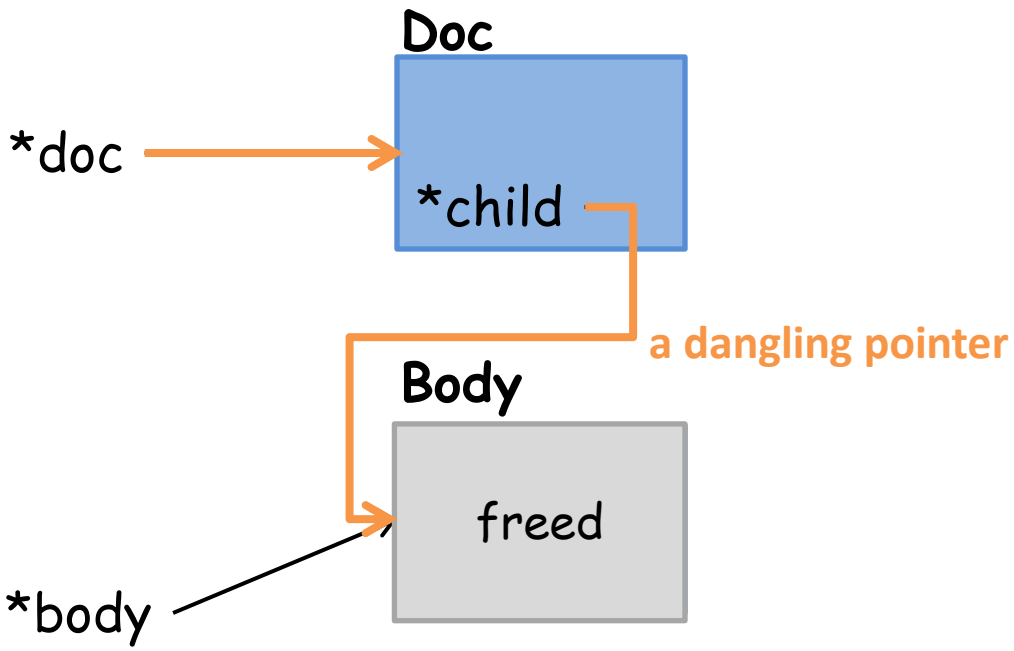
Free an object

```
delete body;
```

Use a dangling pointer

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

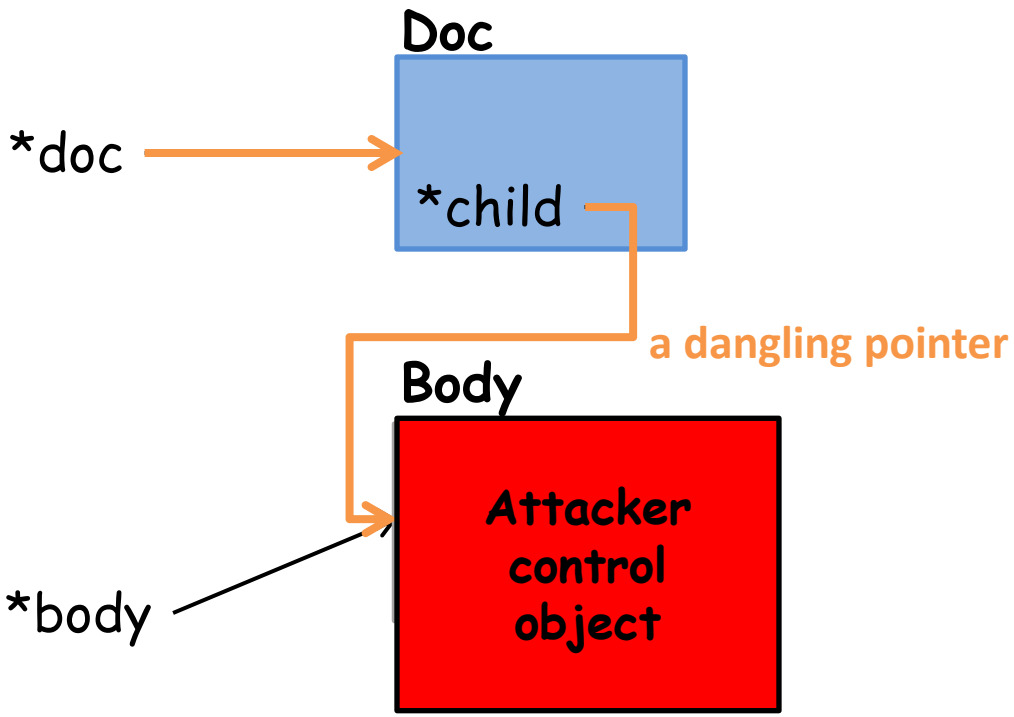
Free an object

```
delete body;
```

Use a dangling pointer

```
if (doc->child)
    doc->child->getAlign();
```

Understanding Use-after-free



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

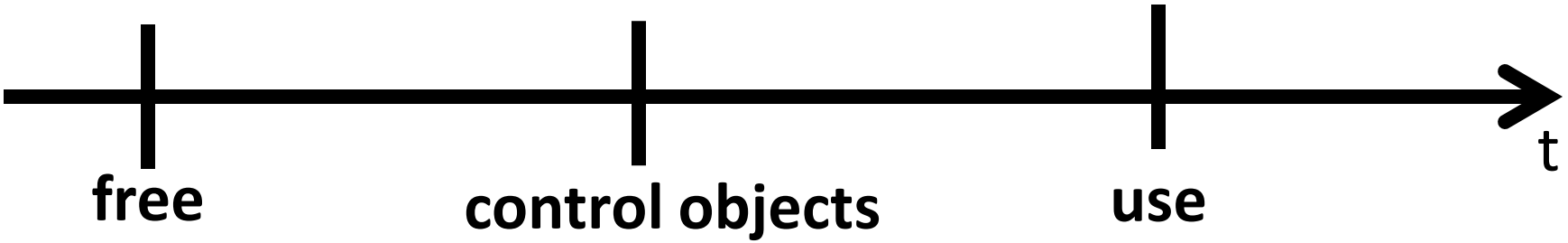
Free an object

```
delete body;
```

Use a dangling pointer

```
if (doc->child)
    doc->child->getAlign();
```

Related Work on Use-after-free



Safe Allocators

AddressSanitizer

Delayed free

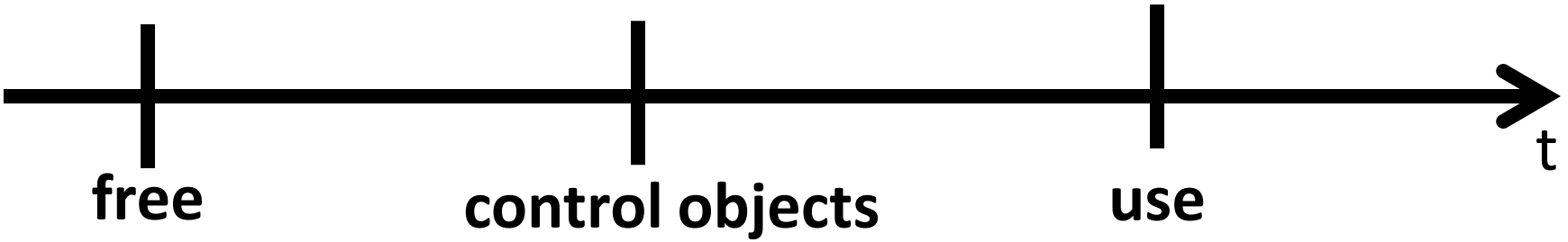
Vtable protection

Control Flow Integrity

Memory Safety

Use-after-free detector

Related Work on Use-after-free



Safe Allocators

AddressSanitizer

Delayed free

Vtable protection

Control Flow Integrity

Memory Safety

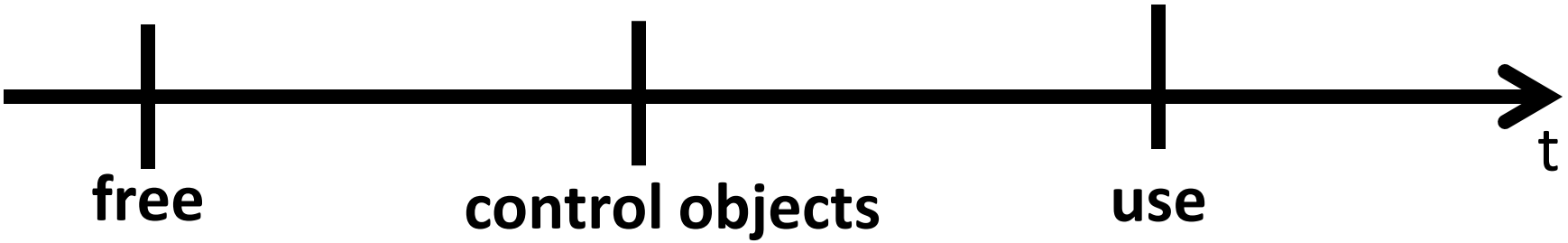
Use-after-free detector

Make exploitation harder, but still bypassable ☹

or

Difficult to support large-scale software ☹

Related Work on Use-after-free



DangNull

Safe Allocators

AddressSanitizer

Delayed free

Vtable protection

Control Flow Integrity

Memory Safety

Use-after-free detector

Make exploitation harder, but still bypassable ☹

or

Difficult to support large-scale software ☹

DangNull: Use-after-free detector

- Tracking Object Relationships
  - Coarse grained pointer semantic tracking

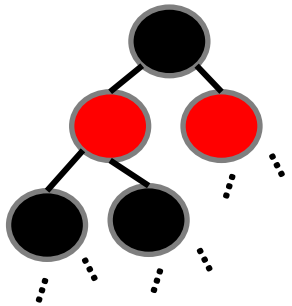
  ➔ Support large-scale software

- Nullify dangling pointers
  - Immediately eliminate all dangling pointers

  ➔Non-bypassable to sophisticated attacks

Tracking Object Relationships

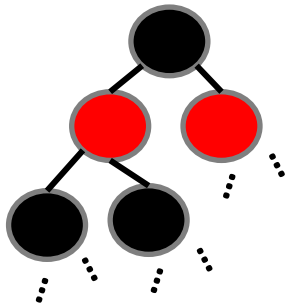
- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



Tracking Object Relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair

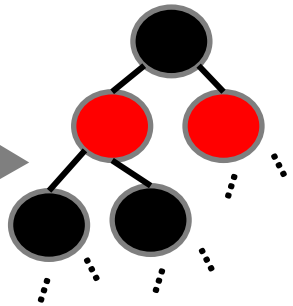
```
Doc *doc = new Doc();
```



Tracking Object Relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair

```
Doc *doc = new Doc();
```



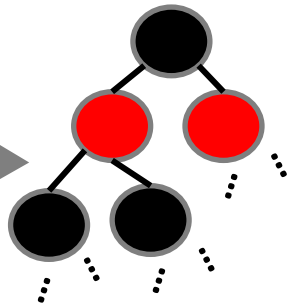
Insert shadow obj:

- Base address of allocation
- Size of Doc

Tracking Object Relationships

- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair

```
Doc *doc = new Doc();
```



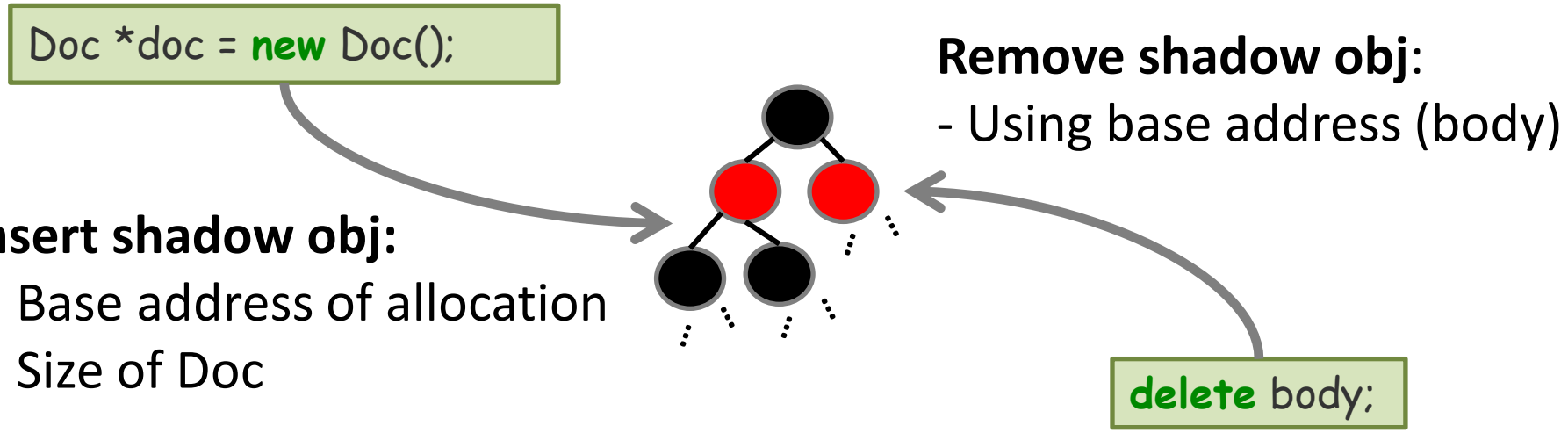
Insert shadow obj:

- Base address of allocation
- Size of Doc

```
delete body;
```

Tracking Object Relationships

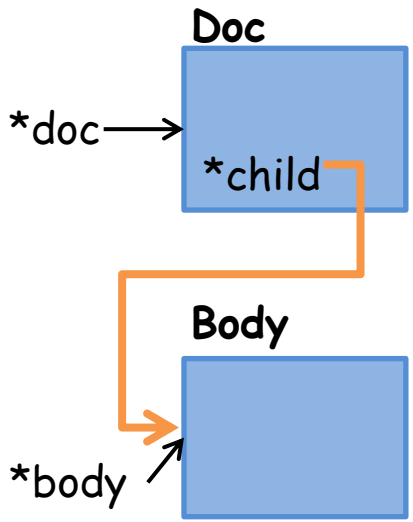
- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

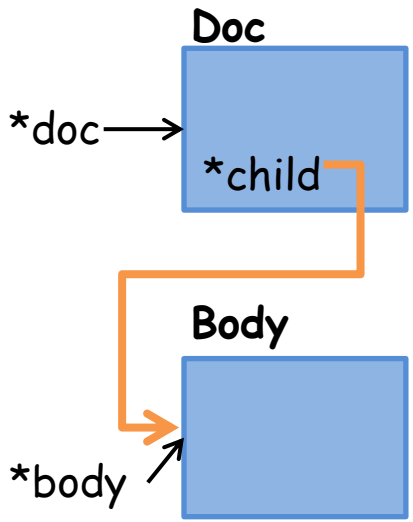
```
doc->child = body;
```



Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

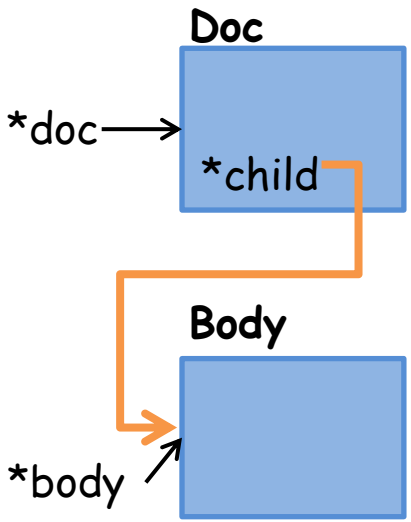
```
doc->child = body;
trace(&doc->child, body);
```



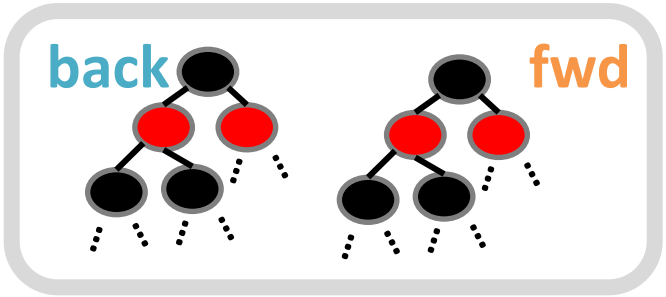
Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

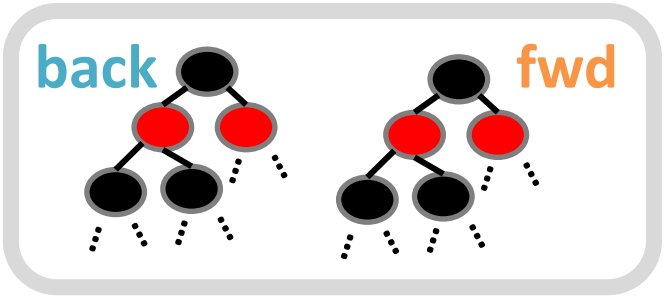
```
doc->child = body;
trace(&doc->child, body);
```



Shadow obj. of Doc



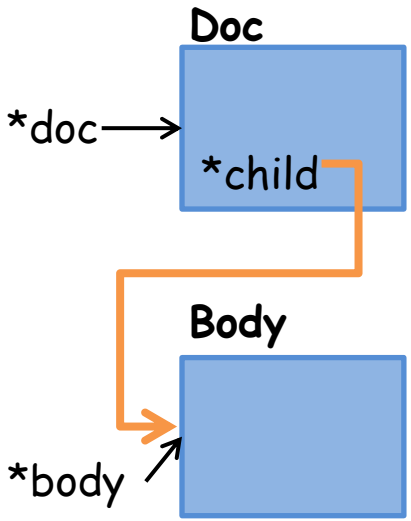
Shadow obj. of Body

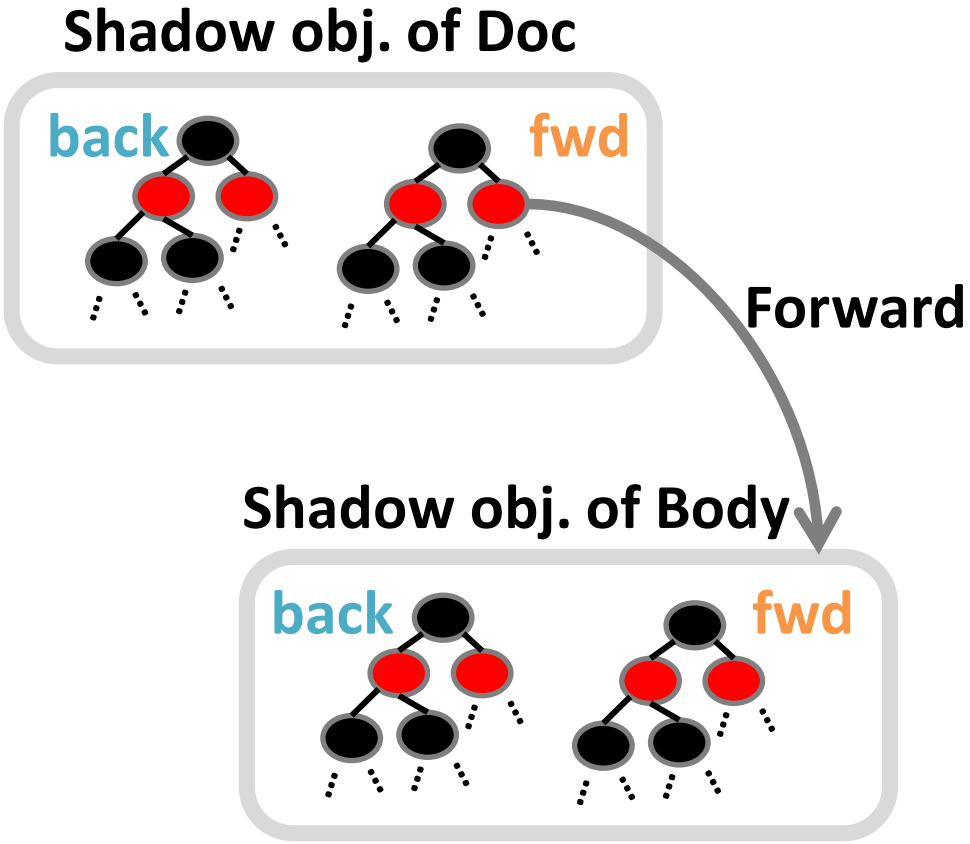


Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

```
doc->child = body;
trace(&doc->child, body);
```

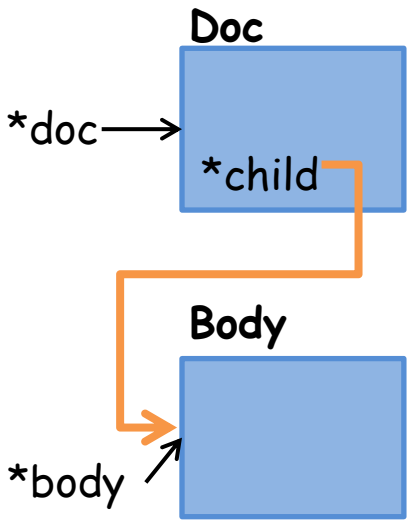


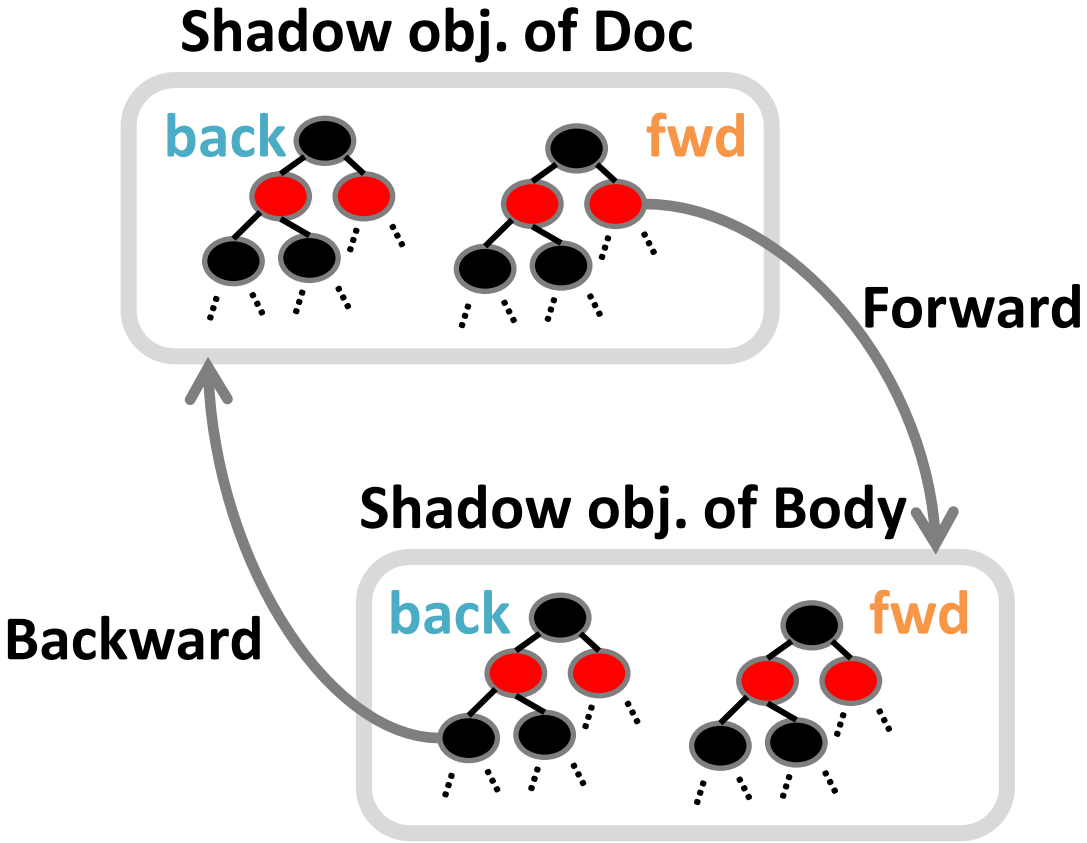


Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

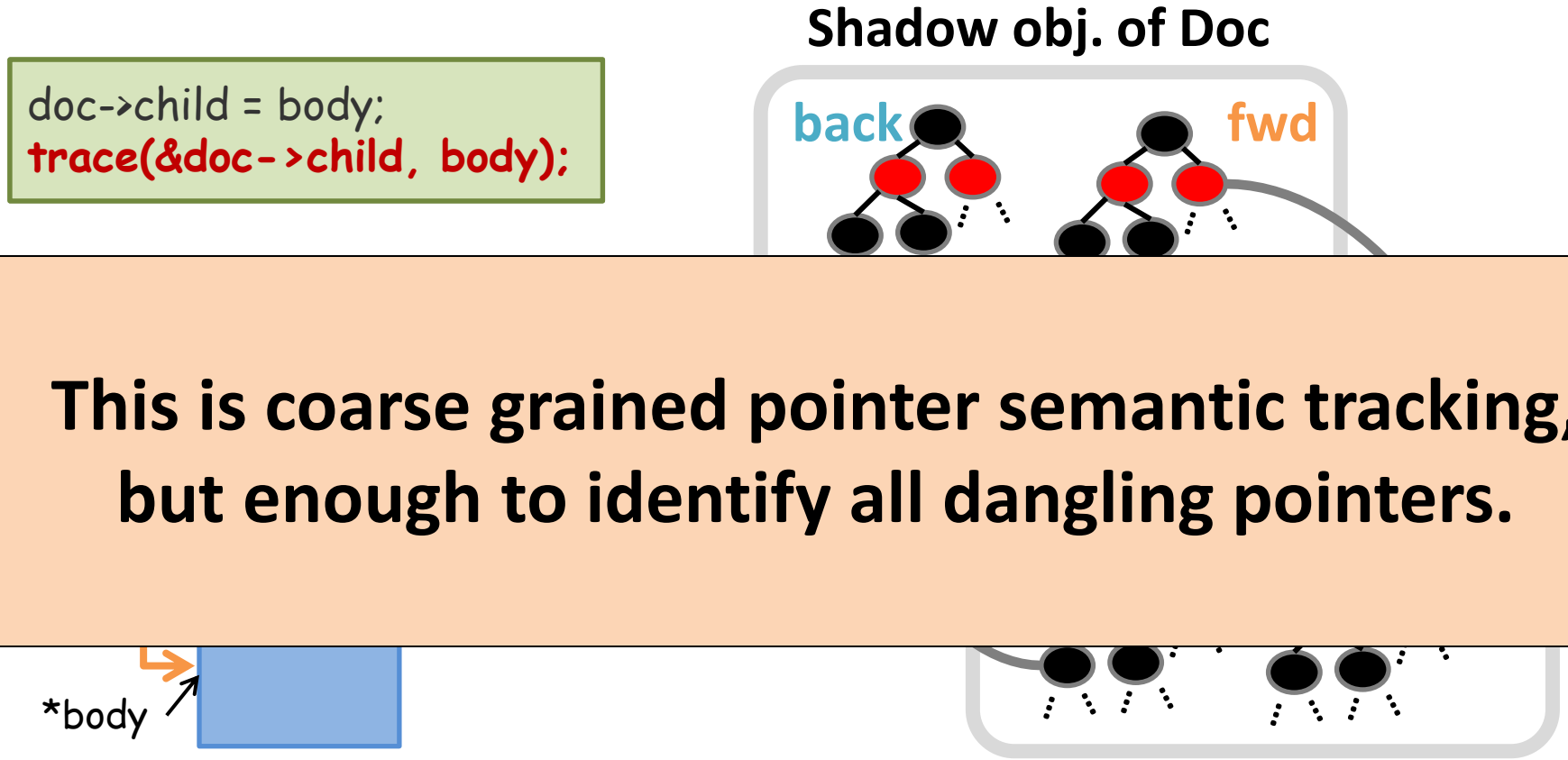
```
doc->child = body;
trace(&doc->child, body);
```





Tracking Object Relationships

- Instrument pointer propagations
  - Maintain backward/forward pointer trees for a shadow obj.

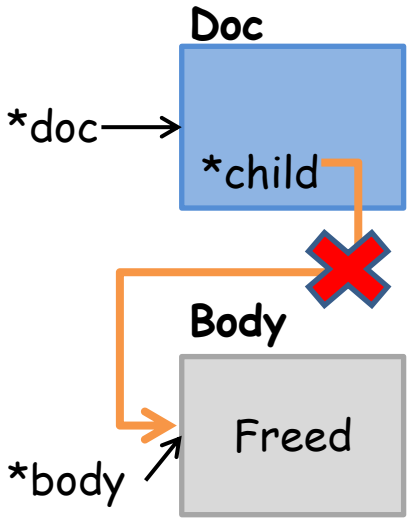


This is coarse grained pointer semantic tracking, but enough to identify all dangling pointers.

Nullifying Dangling Pointers

- Nullify all backward pointers of Body, once it is deleted.
  - All backward pointers of Body are dangling pointers
  - Dangling pointers have no semantics

```
delete body;
```



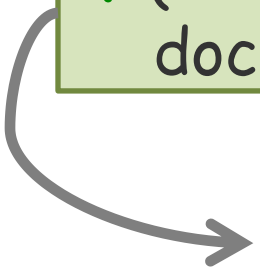
Nullifying Dangling Pointers

```
delete body;
```

Nullification

```
doc->child = NULL
```

```
if (doc->child)
    doc->child->getAlign();
```



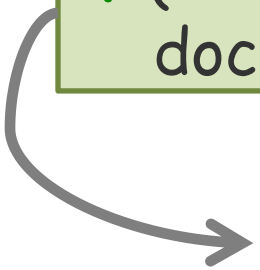
Null-dereference is safely contained in pre-mapped nullpadding

Nullifying Dangling Pointers

```
delete body;
```

Immediately eliminate all dangling pointers!

```
if (doc->child)
    doc->child->getAlign();
```



Null-dereference is safely contained in pre-mapped nullpadding

Implementation

- Prototype DangNull
  - Instrumentation: LLVM pass, +389 LoC
  - Runtime: compiler-rt, +3,955 LoC
- To build target applications,
  - SPEC CPU 2006: one extra compiler and linker flag
  - Chromium: +27 LoC to .gyp build configuration file

Performance Evaluation

- Chromium browser
  - Instrumented 140k/16,831k (0.8%) instructions
  - Passed all unit tests and layout tests
  - Overall 28.9% overheads on various benchmarks
  - A page loading time for the Alexa top 100 websites
    - 7% increased load time
  - While visiting http://google.com,
    - 123k shadow objects and 32k shadow pointers
    - 7k nullifications

Conclusion

- Presented DangNull, which detects use-after-free
- Supporting large-scale software
- Non-bypassable to sophisticated attacks

Demo

- Running Chromium browser (version 29.0.1547.65)
  - Hardened using DangNull
  - Testing use-after-free exploit (PoC)
    - CVE-2013-2909: Heap-use-after-free in WebCore::RenderBlock::determineStartPosition