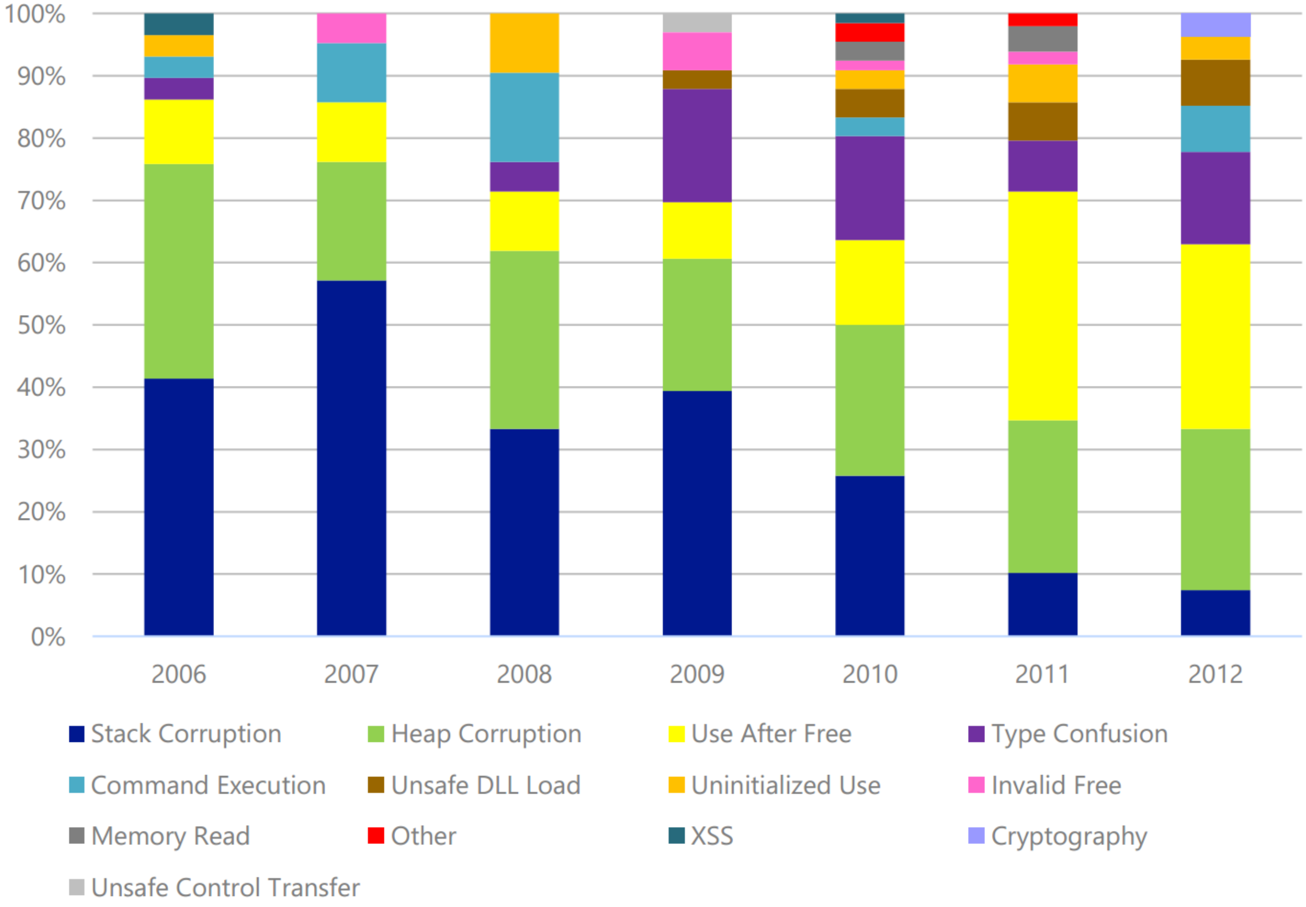
Preventing Use-after-free with Dangling Pointers Nullification

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

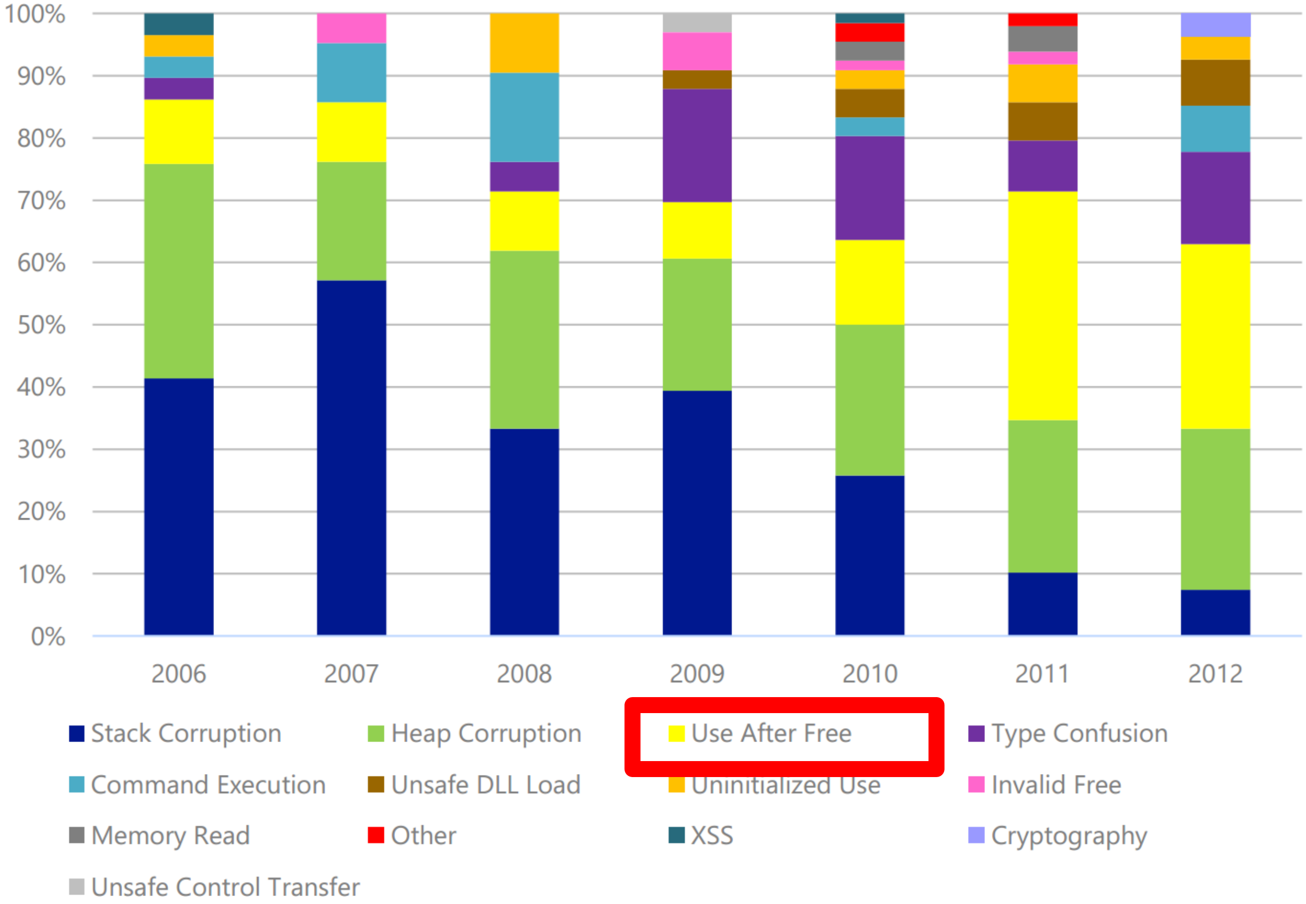
Georgia Institute of Technology
Stony Brook University

Emerging Threat: Use-after-free



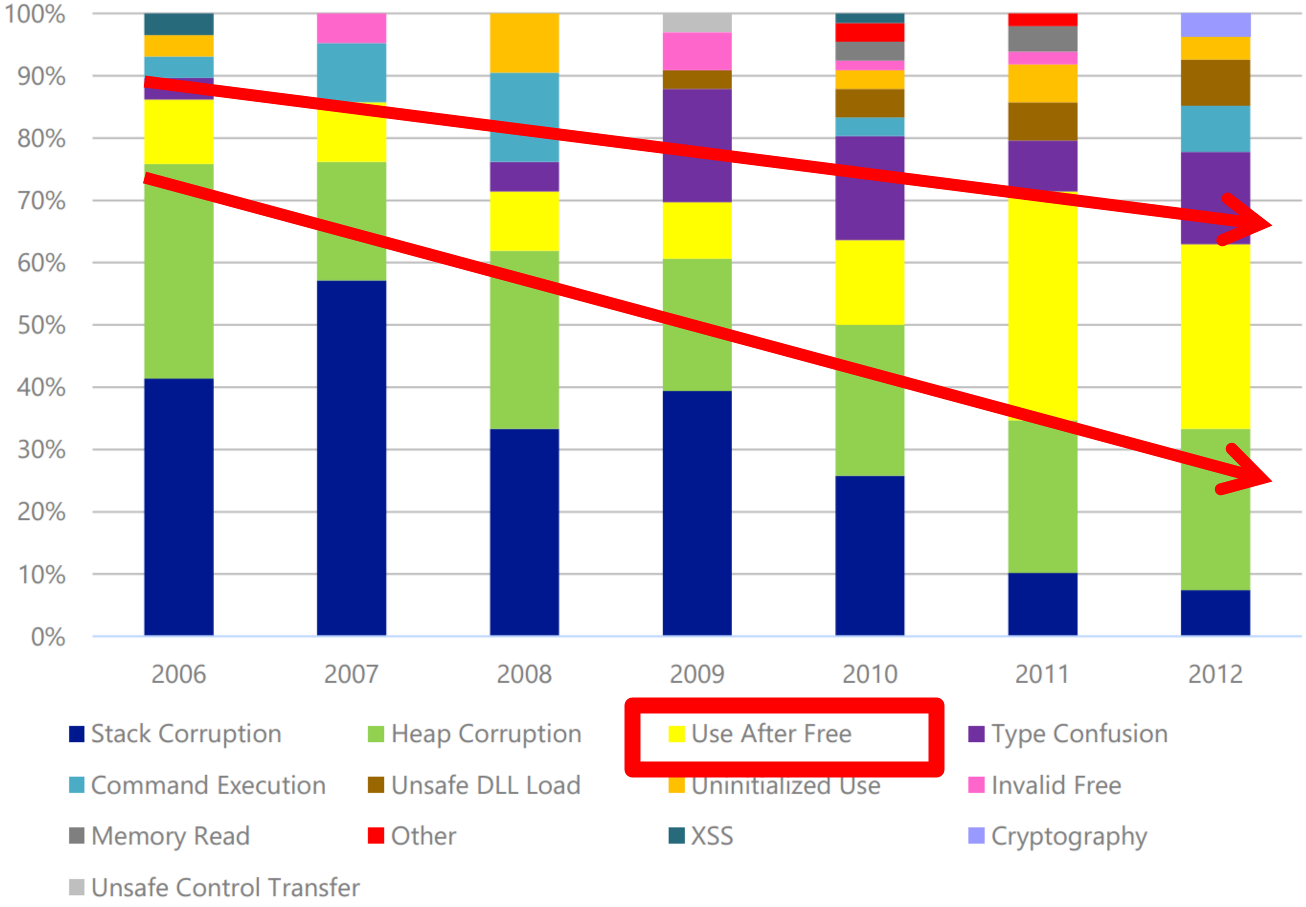
Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free



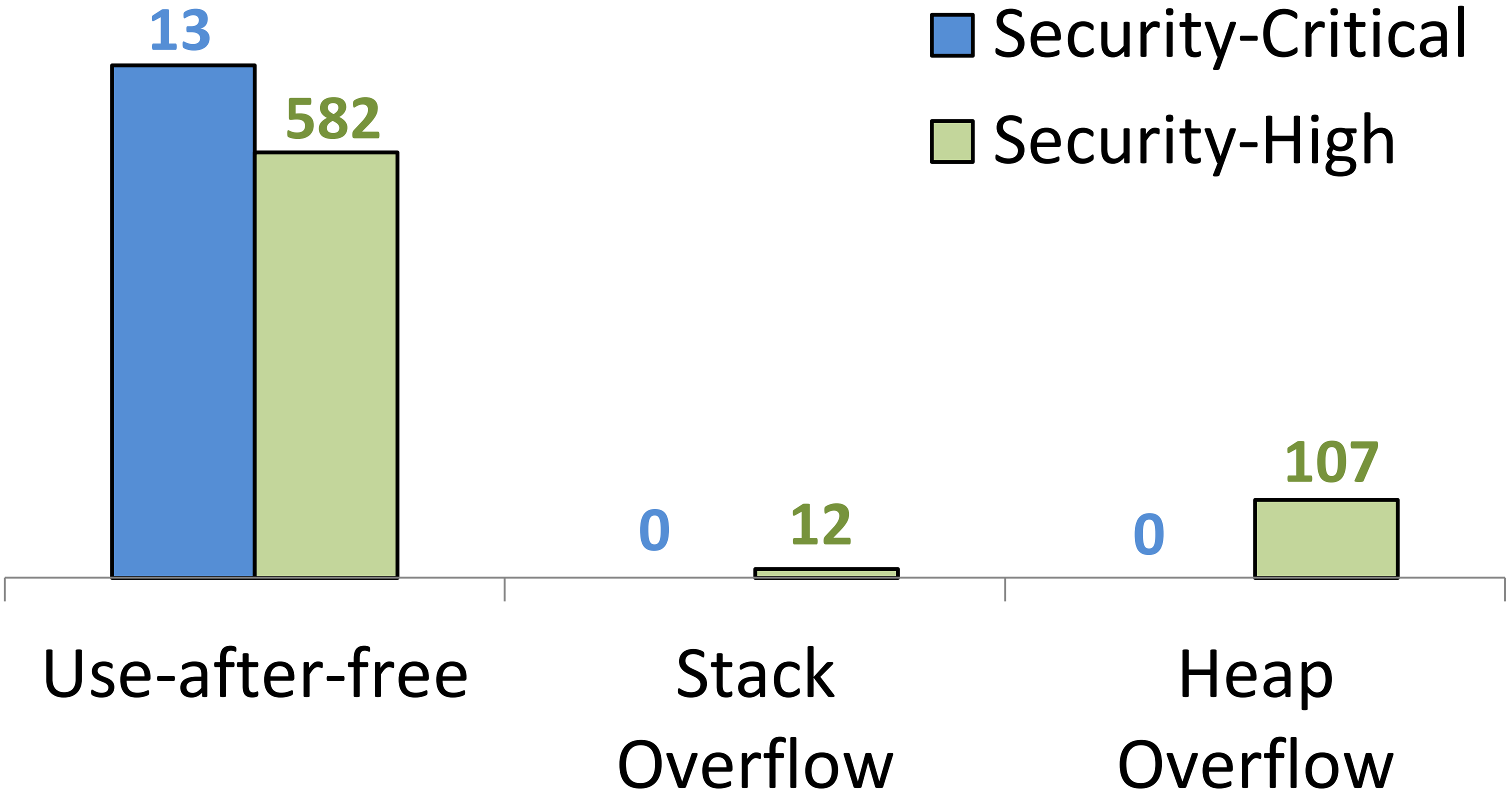
Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free



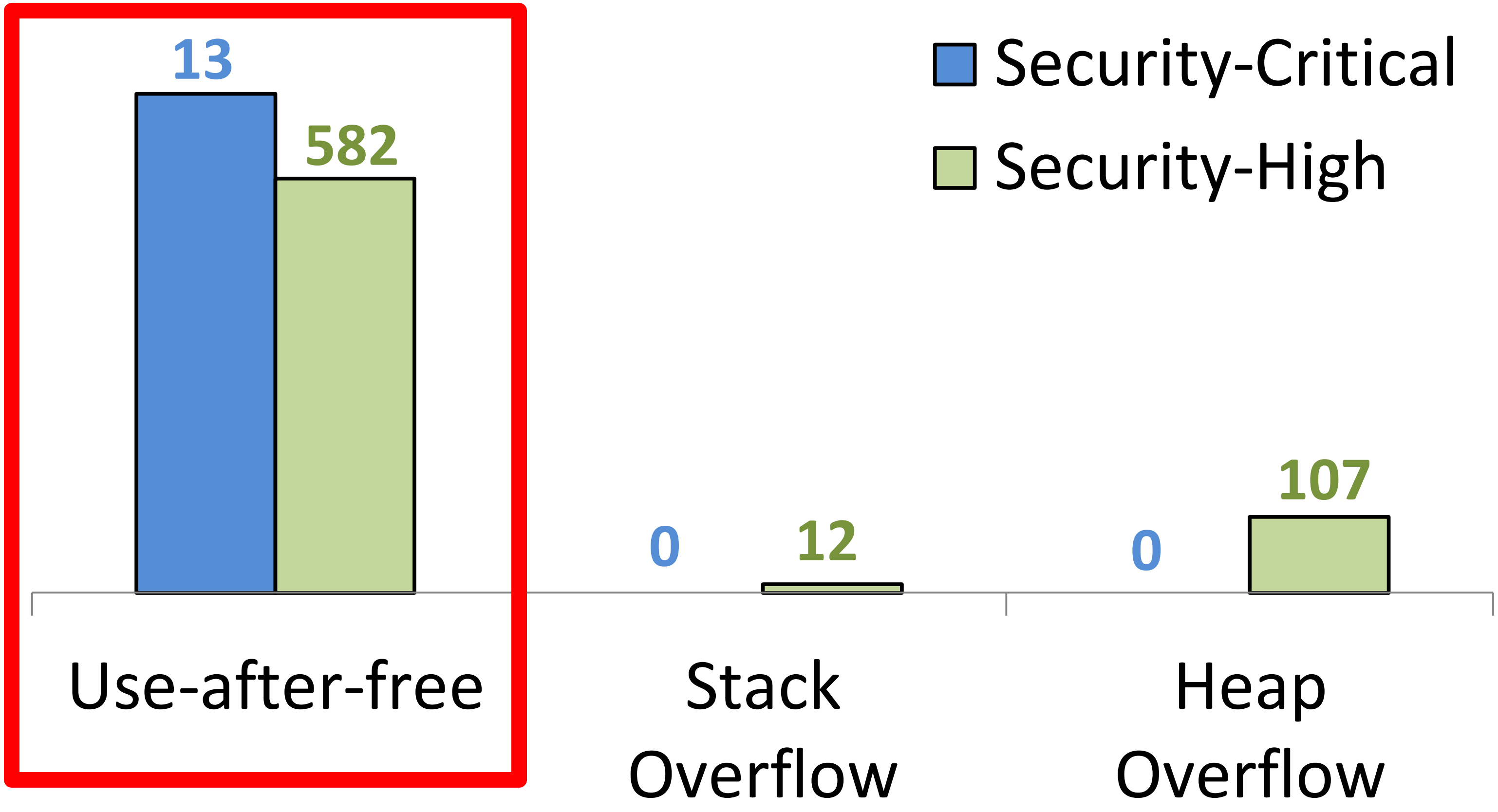
Software Vulnerability Exploitation Trends, Microsoft, 2013

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
  - May lead to arbitrary code executions
  - so called use-after-free

Understanding Use-after-free

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

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

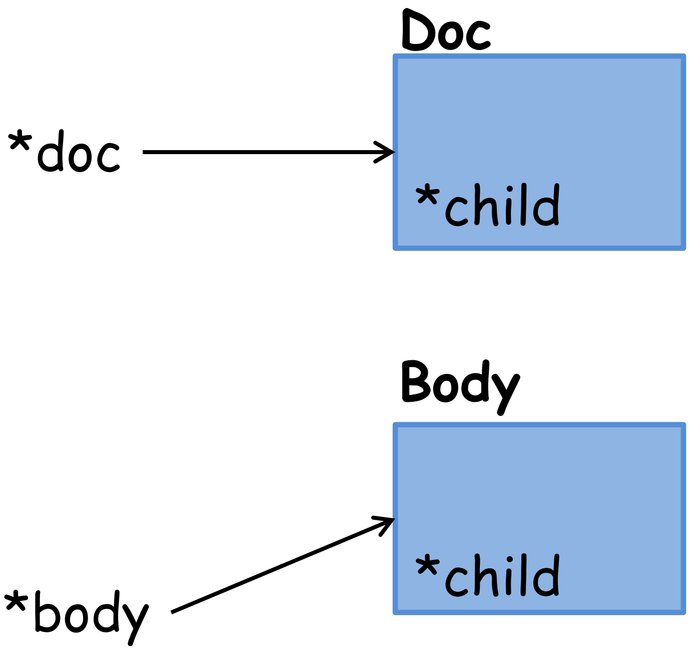
```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

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

Understanding Use-after-free



Allocate objects

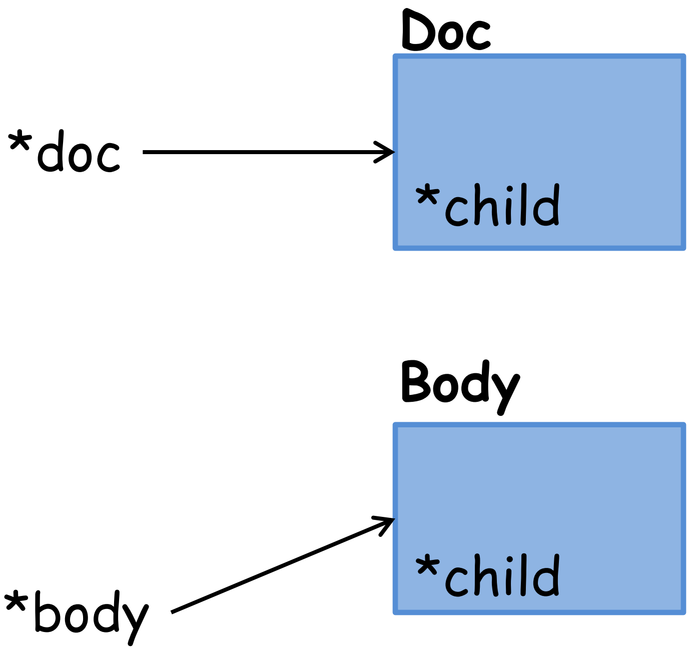
```
Doc *doc = new Doc();
Body *body = new Body();
```

```
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();
```

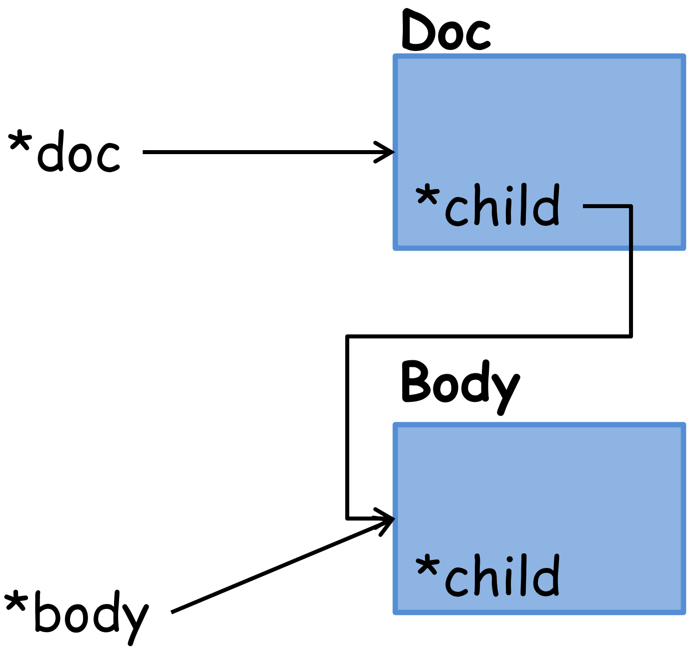
Propagate pointers

```
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();
```

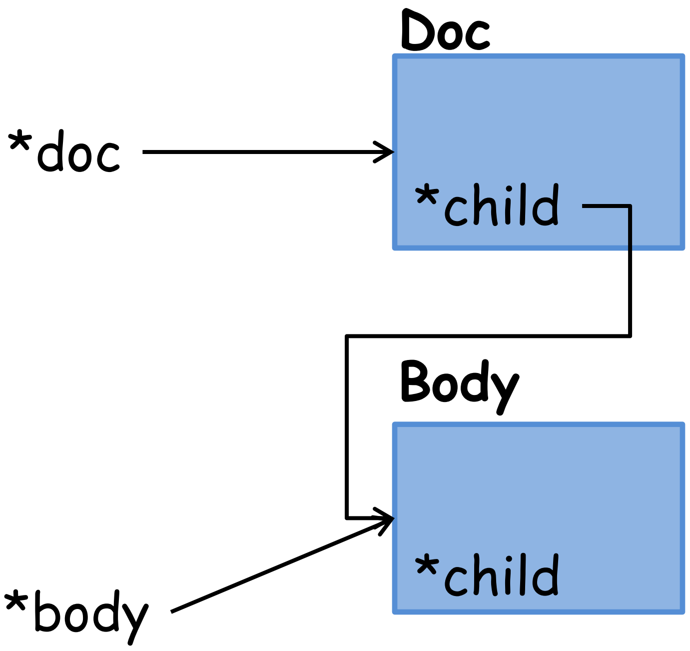
Propagate pointers

```
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

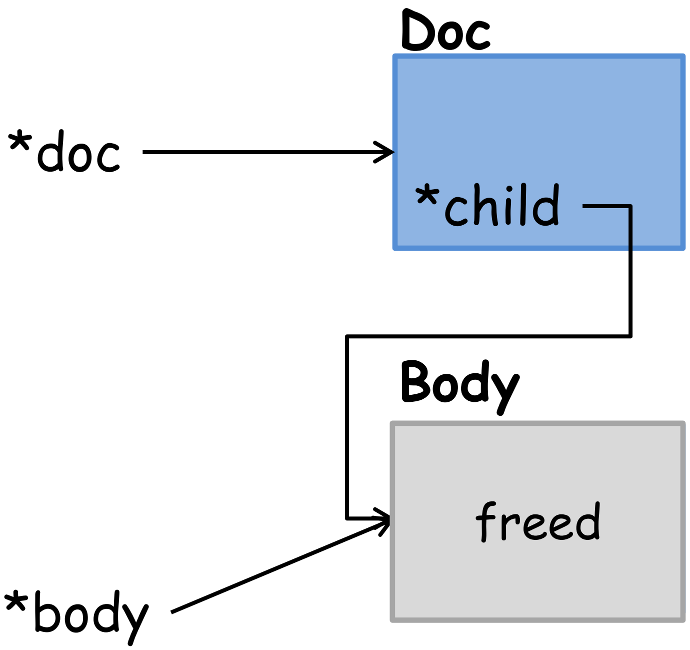
```
doc->child = body;
```

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

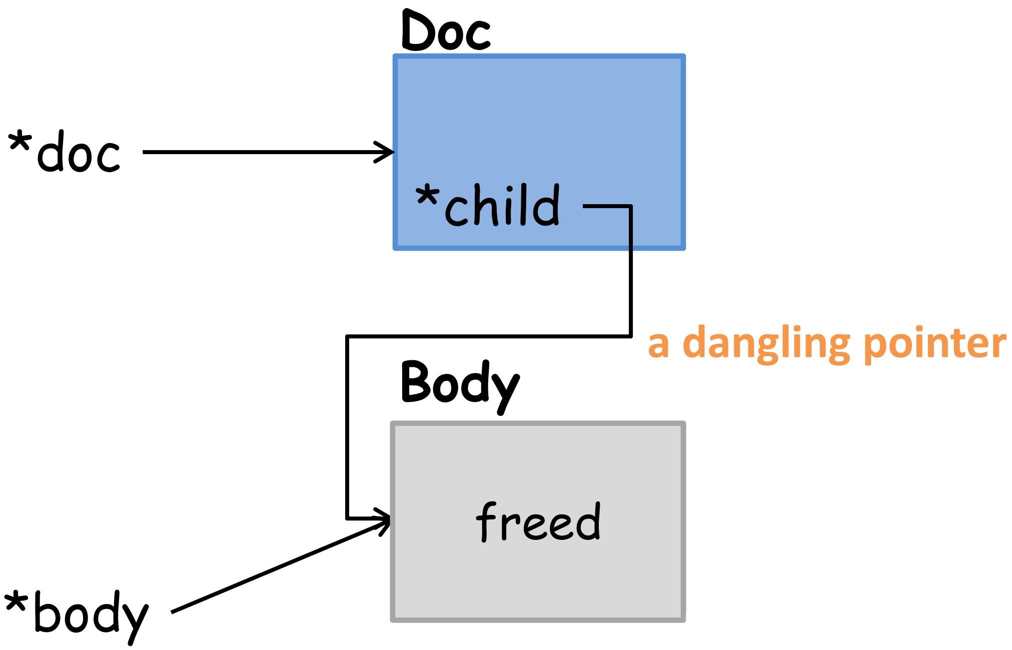
```
doc->child = body;
```

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

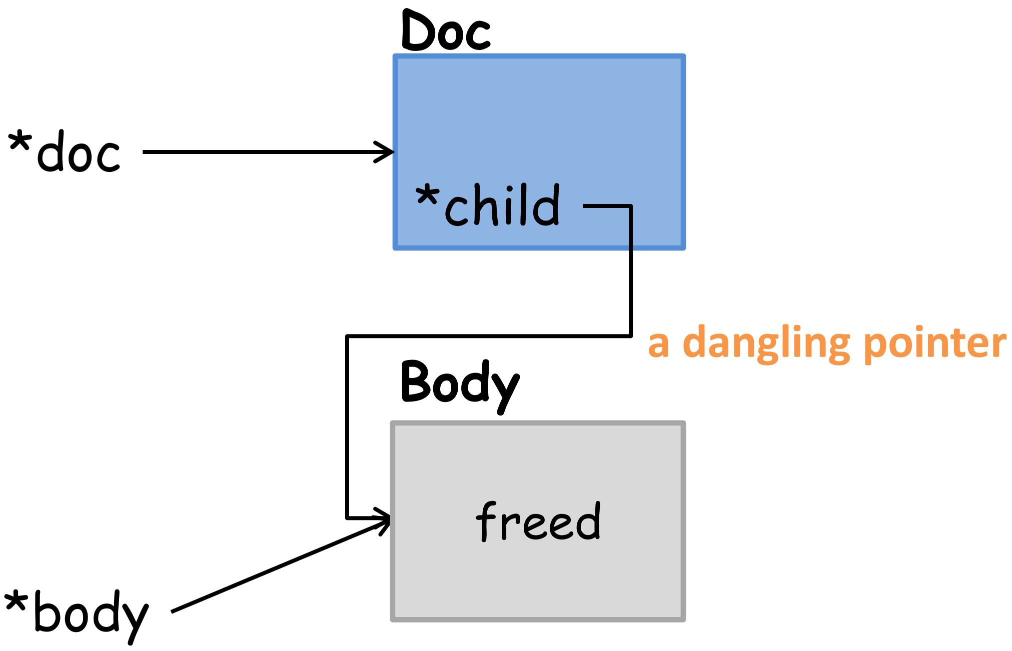
```
doc->child = body;
```

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;
```

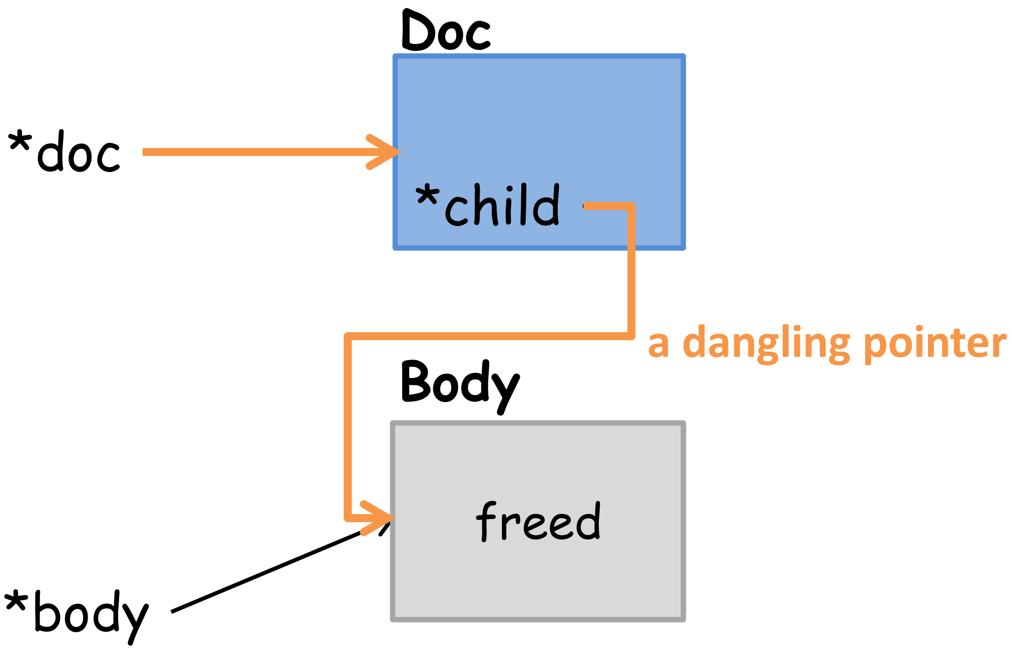
Free an object

```
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();
```

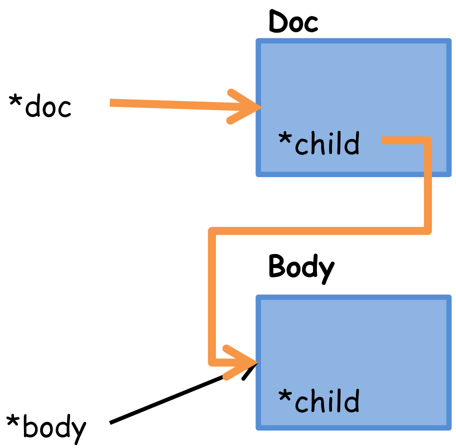
Why use-after-free is challenging

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

doc->child = body;
body->child = div;

delete body;

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



Why use-after-free is challenging

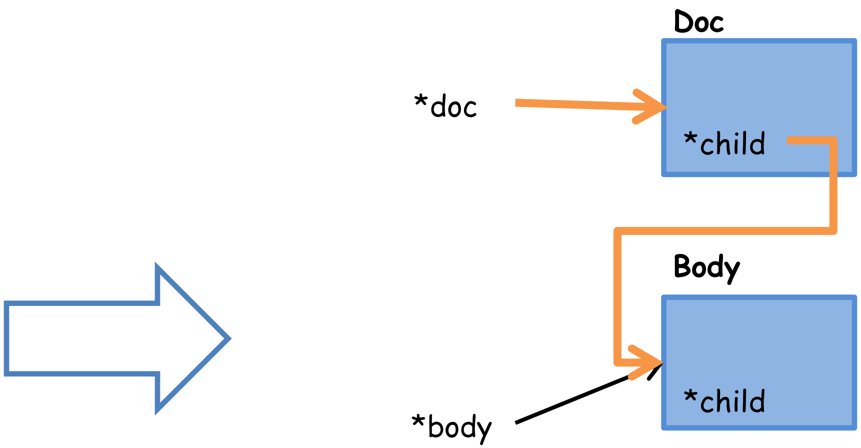
```
Doc *doc = new Doc();
```

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

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

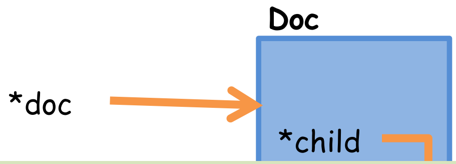
```
delete body;
```

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



Why use-after-free is challenging

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



- ✓ Reconstructing object relationships is challenging
  - ✓ Static analysis
    - ✓ Modules are disconnected and scattered
    - ✓ Difficult to serialize execution orders
  - ✓ Dynamic analysis
    - ✓ Tracing pointer semantics is non-trivial

Contributions

- Present **DangNull**, which detects use-after-free
  - (sometimes) even surviving from use-after-free
- Stop sophisticated attacks
  - Immediately eliminate security impacts of use-after-free
- Support large-scale software
  - Protect popular apps including web browsers

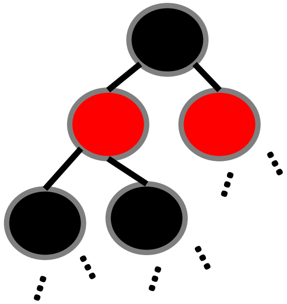
Designs

- Tracking Object Relationships
  - Intercept allocations/deallocations
  - Instrument pointer propagations

- Nullify dangling pointers
  - A value in dangling pointers has no semantics
  - Dereferencing nullified pointers will turn into safe-null dereference

Tracking Object Relationships

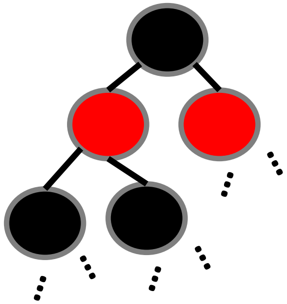
- Intercept allocations/deallocations in runtime
  - Maintain Shadow Object Tree
    - Red-Black tree to efficiently keep object layout information
    - Node: (base address, size) pair



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

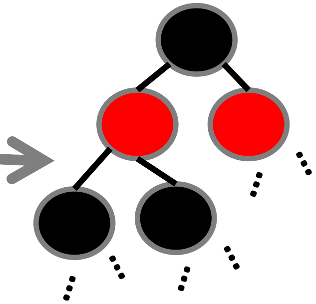
```
Doc *doc = new Doc();
```



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();
```



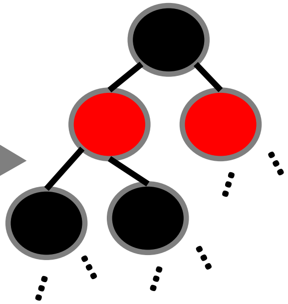
Insert shadow obj:

- 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:

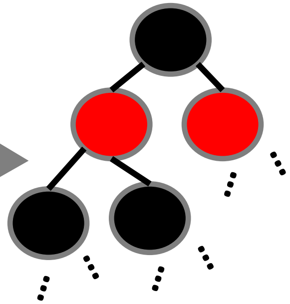
- Base address of allocation
- Size of Doc

```
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

`Doc *doc = new Doc();`



Remove shadow obj:

- Using base address (body)

Insert shadow obj:

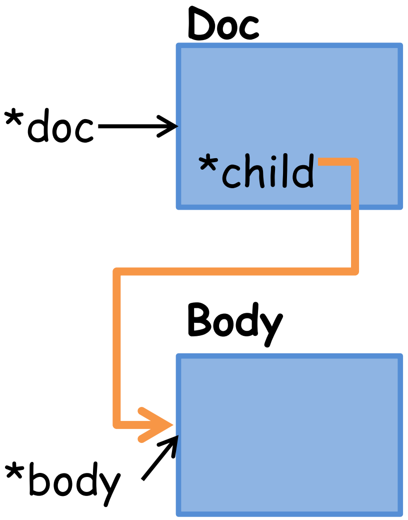
- Base address of allocation
- Size of Doc

`delete body;`

Tracking Object Relationships

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

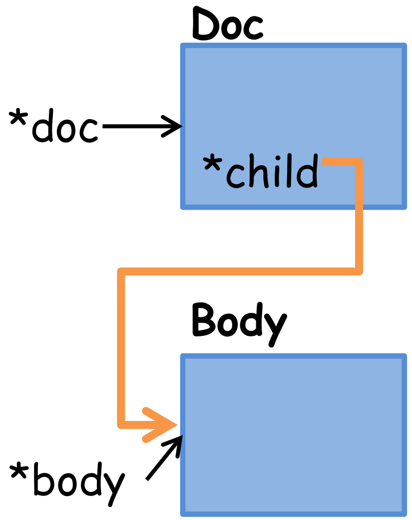
```
doc->child = body;
```



Tracking Object Relationships

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

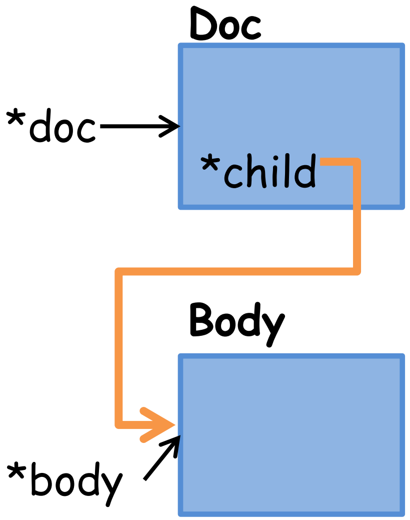
```
doc->child = body;
trace(&doc->child, body);
```



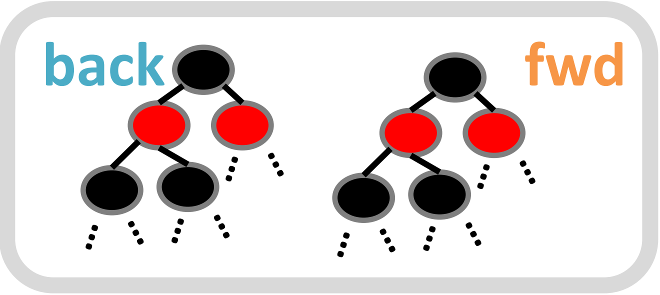
Tracking Object Relationships

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

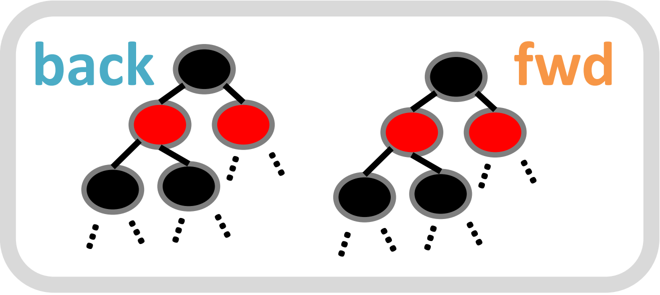
```
doc->child = body;
trace(&doc->child, body);
```



Shadow obj. of Doc



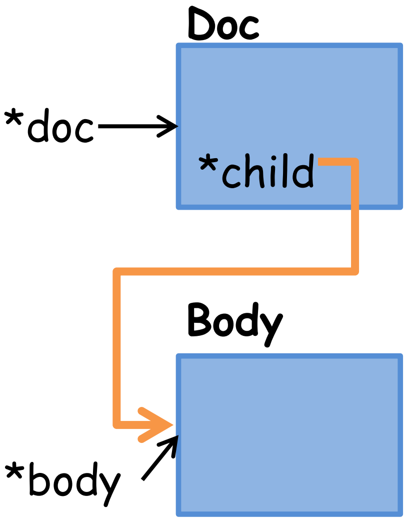
Shadow obj. of Body

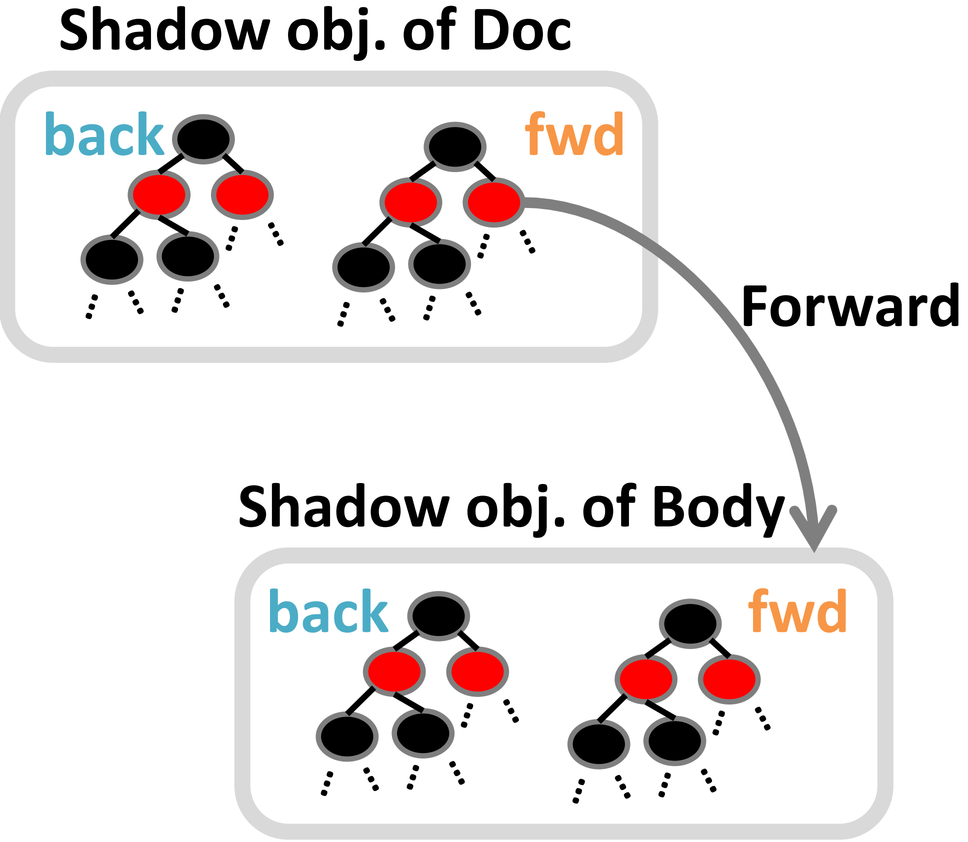


Tracking Object Relationships

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

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

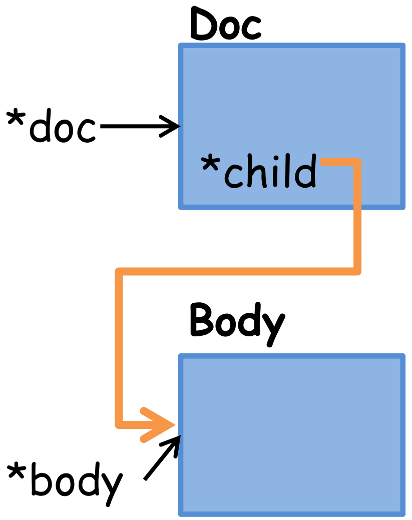


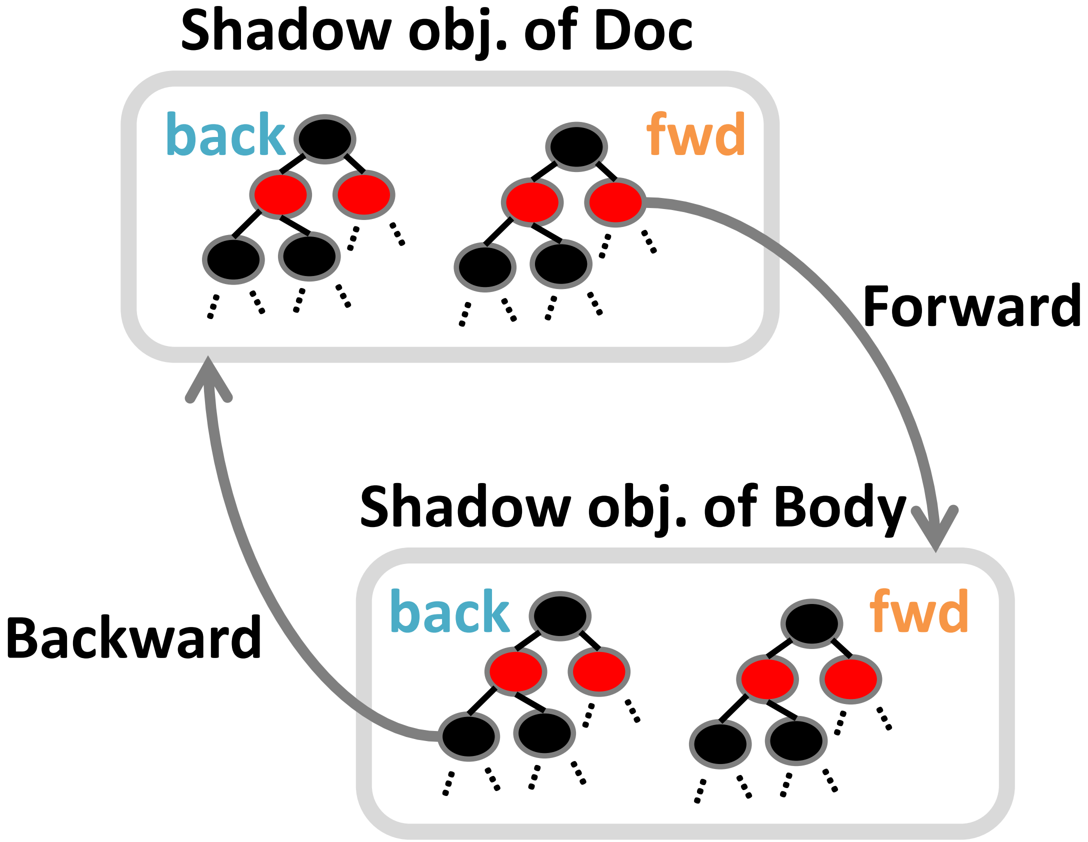


Tracking Object Relationships

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

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



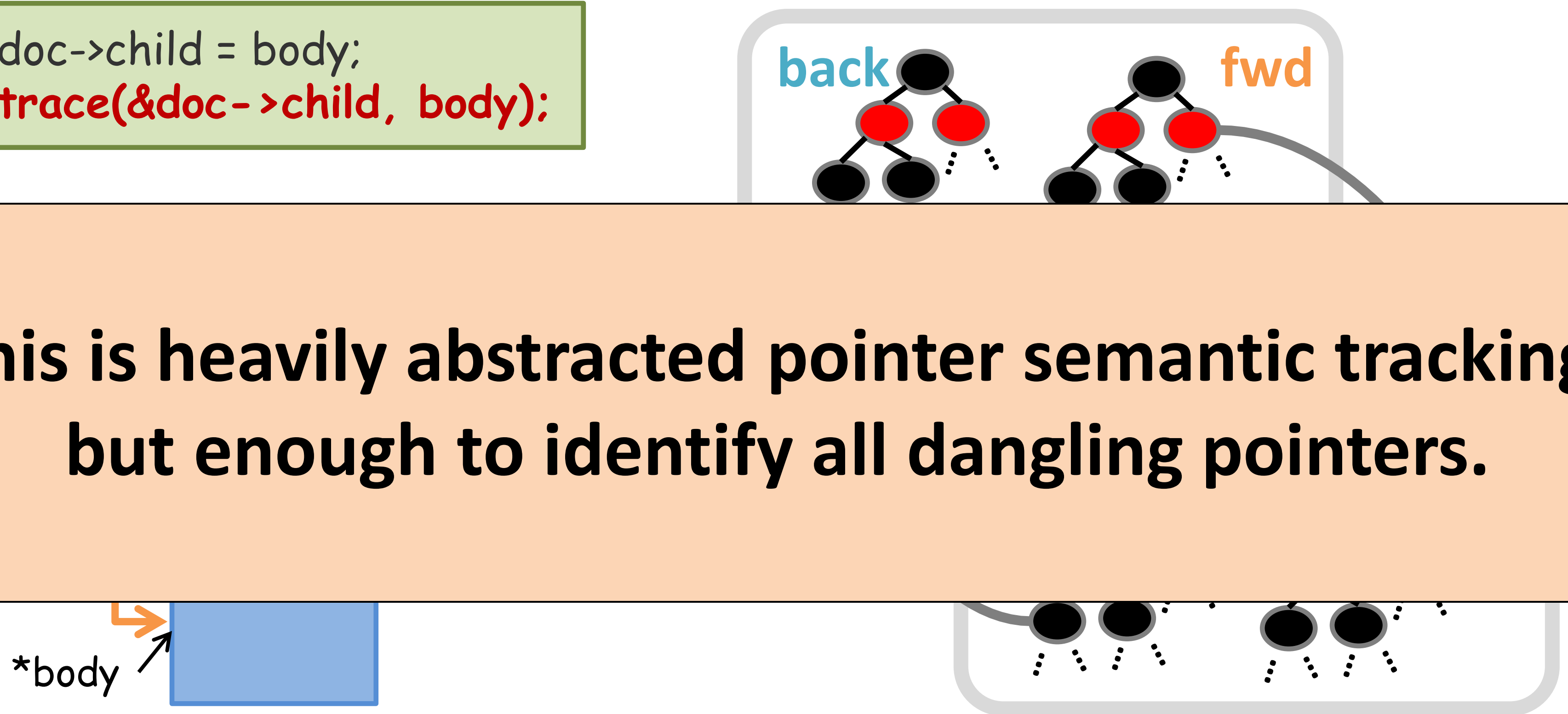


Tracking Object Relationships

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

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

Shadow obj. of Doc



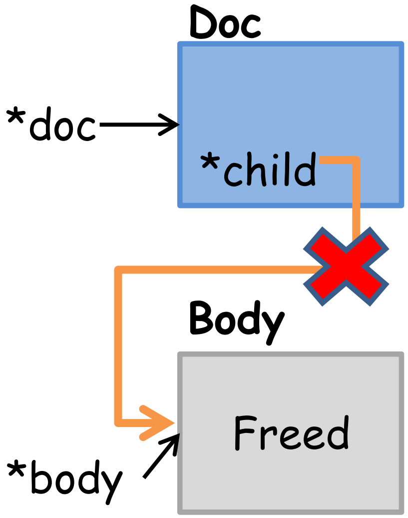
This is heavily abstracted pointer semantic tracking, but enough to identify all dangling pointers.

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
  - Immediately eliminate dangling pointers

```
delete body;
```

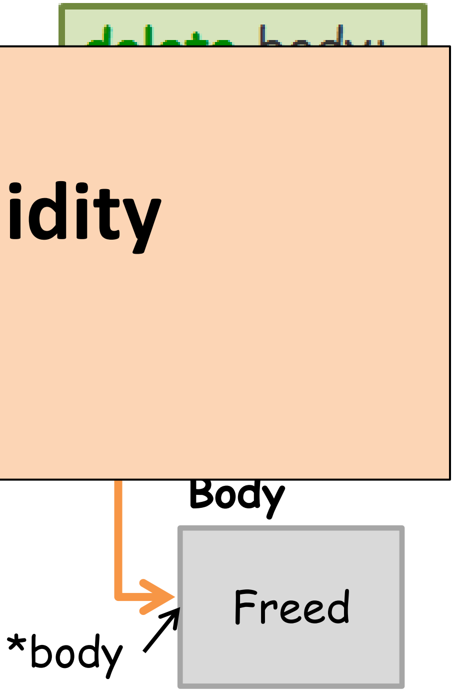
- Using nullified pointers later will turn into safe-null dereference.



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
  - Immediately eliminate dangling pointers

No need to check the pointer validity at the time of use!



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
  - JavaScript benchmarks
    - 4.8% overheads
  - Rendering benchmarks
    - 53.1% overheads
  - A page loading time for the Alexa top 100 websites
    - 7% increased load time

Conclusion

- Presented DangNull, which detects use-after-free in runtime
- Applications
  - Use-after-free prevention for end-users
  - Debugging use-after-free vulnerability
  - Backend new use-after-free vulnerability finding

Demo

- Running Chromium browser (version 29.0.1547.65)
  - Hardened using DangNull
    - 140k/16,831k (0.8%) instructions were instrumented
  - Testing use-after-free exploit (PoC)
    - CVE-2013-2909: Heap-use-after-free in WebCore::RenderBlock::determineStartPosition

Backup slides

Interception / Instrumentation of DangNull

Allocate objects

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

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

Propagate pointers

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

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

Free an object

```
delete body;
```

```
delete body;
```

Use a dangling pointer

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

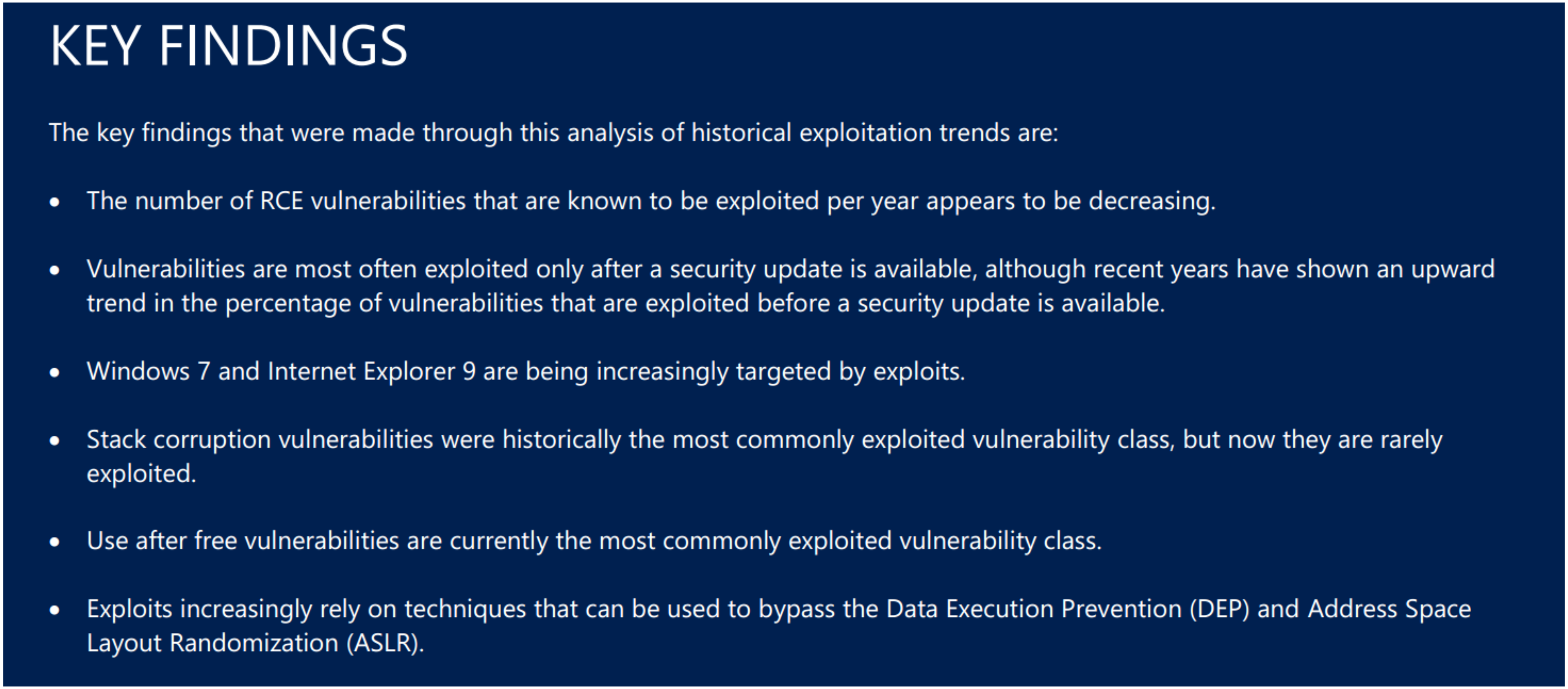
```
if (doc->child)
   doc->child->getAlign();
```

Use-after-free and dangling pointers

- Use-after-free != dangling pointer
  - Use-after-free happens iif a dangling pointer is used.

- Dangling pointers
  - A pointer points to the freed memory region
  - No data semantics
- Benign dangling pointers
  - Never dereferenced dangling pointers
- Unsafe dangling pointers
  - Dereferenced dangling pointers

Emerging Threat: Use-after-free



KEY FINDINGS

The key findings that were made through this analysis of historical exploitation trends are:

- The number of RCE vulnerabilities that are known to be exploited per year appears to be decreasing.
- Vulnerabilities are most often exploited only after a security update is available, although recent years have shown an upward trend in the percentage of vulnerabilities that are exploited before a security update is available.
- Windows 7 and Internet Explorer 9 are being increasingly targeted by exploits.
- Stack corruption vulnerabilities were historically the most commonly exploited vulnerability class, but now they are rarely exploited.
- Use after free vulnerabilities are currently the most commonly exploited vulnerability class.
- Exploits increasingly rely on techniques that can be used to bypass the Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR).

Software Vulnerability Exploitation Trends, Microsoft, 2013

Emerging Threat: Use-after-free

KEY FINDINGS

The key findings that were made through this analysis of historical exploitation trends are:

- The number of RCE vulnerabilities that are known to be exploited per year appears to be decreasing.
- Vulnerabilities are most often exploited only after a security update is available, although recent years have shown an upward trend in the percentage of vulnerabilities that are exploited before a security update is available.
- Windows 7 and Internet Explorer 9 are being increasingly targeted by exploits.
- exploited vulnerability class, but now they are rarely
- Use after fre
- Exploits incre
  Layout Randomization (ASLR).

- Use after free vulnerabilities are currently the most commonly exploited vulnerability class.

Software Vulnerability Exploitation Trends, Microsoft, 2013