THIS IS FOR THE PWNERS: EXPLOITING A WEBKIT 0-DAY IN PLAYSTATION 4

Written by Mehdi Talbi , Quentin Meffre - 10/12/2020 - in Exploit

Despite an active console hacking community, only few public PlayStation 4 exploits have been released. In this post, we will give a walk-through on the exploitation of a 0-day WebKit vulnerability on 6.xx firmware.

The exposed WebKit-based browser is usually the entry point of a full-chain attack: from browser exploitation to kernel exploitation. However, browser engine hardening techniques together with the total absence of debugging capabilities makes it very hard to successfully exploit bugs in the latest PS4 firmware.

In this post, we will introduce the root cause of the bug. The bug provides limited exploitation primitives. However, thanks to a weakness we identified in ASLR mechanism, we were able to make this bug exploitable. We will focus on the exploitation strategy we adopted and how we turned a non-trivial Use-After-Free into a R/W primitive leading to code execution.

ATTACKING THE PS4

The browser is probably the most common entry point to attack the PS4. The browser is based on WebKit and runs in a sandbox. Unlike iOS devices, there is no modern
mitigation such as the Gigacage or the `StructureID` randomization and JIT is not enabled.

A typical exploit chain starts with a WebKit exploit to get code execution in the renderer process followed by a sandbox bypass to run a kernel exploit.

There have been a couple of WebKit vulnerabilities that have been successfully exploited in the past. The `bad-hoist` exploit - by @Fire30_ - is the last known public exploit on the PS4. It exploits the vulnerability CVE-2018-4386 found by @lokihardt and provides read/write primitives. The exploit works on firmware up to 6.72.

Another vulnerability found by @lokihardt (CVE-2018-4441) has been also exploited on the PS4 by @SpecterDev. The exploit provides also read/write primitives and works on firmware 6.20.

For older firmware (< 6.xx), there are some few exploits by @qwertyoruiopz, @SpecterDev, @CTurt, ...

Regarding kernel exploits, @theflow0 released the last exploit in date. The exploit provides read/write primitives in the kernel and is reachable from the WebKit sandbox. The bug is present on firmware up to 7.02 and has been recently combined with the `bad-hoist` exploit to get a full chain on 6.xx firmware.

Finally, there are some few vulnerabilities on BPF that have been discovered and exploited by @qwertyoruiopz.

We strongly encourage readers to take a look at the excellent write-up by @SpecterDev and the blog series on hacking the PS4 by @CTurt.

WEBKIT HEAPS

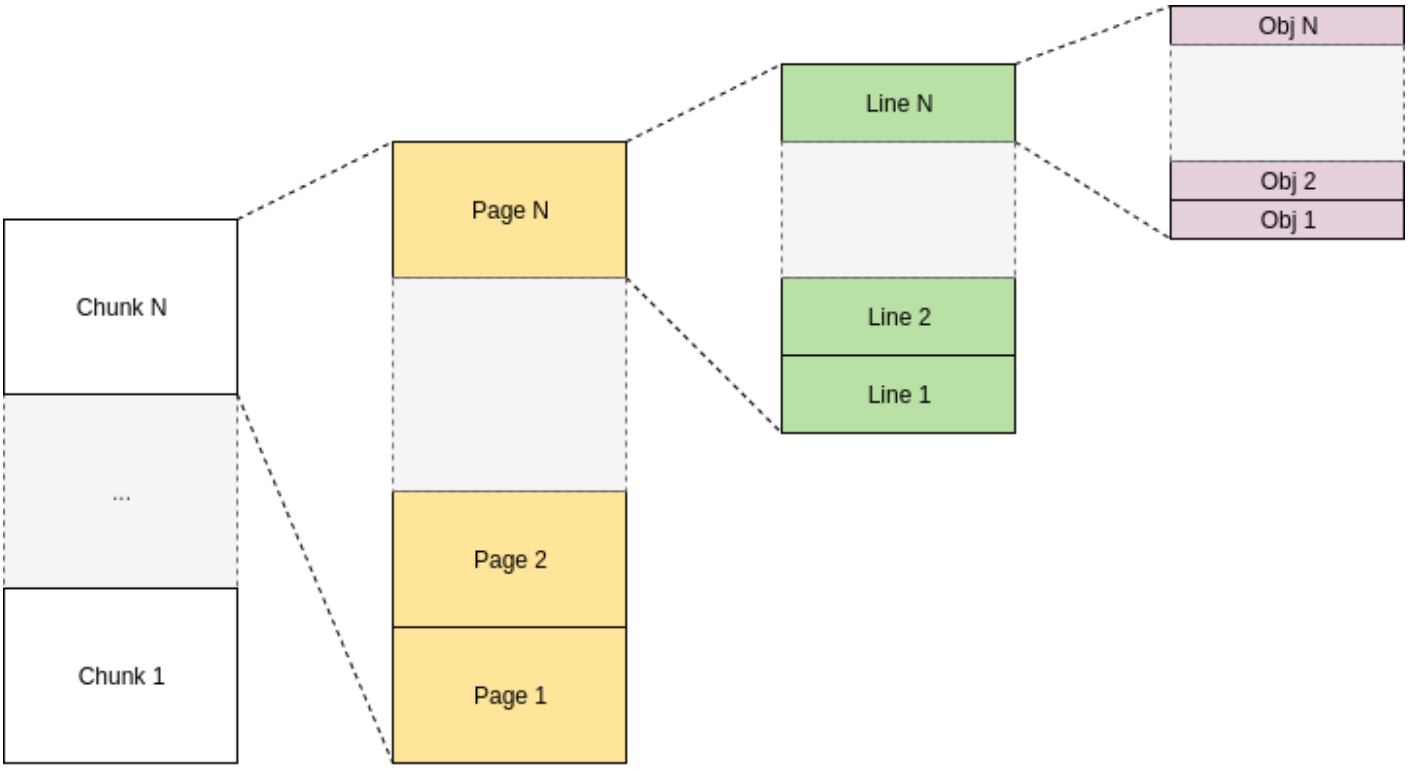
WebKit has many heap allocators:

- FastMalloc is the standard allocator. It is used by many WebKit components ;

- IsoHeap is used by the DOM engine. Its purpose is to sort each allocation using their types to mitigate UAF vulnerability ;
- Garbage Collector is used by the JavaScript engine to allocate JavaScript objects ;
- IsoSubspace is also used by the JavaScript engine. Its purpose is the same as IsoHeap but it is used for a few objects ;
- Gigacage implements mitigations to prevent out-of-bound read/write on specific objects. As mentioned earlier, it is disabled on the PS4.

THE PRIMARY HEAP ALLOCATOR

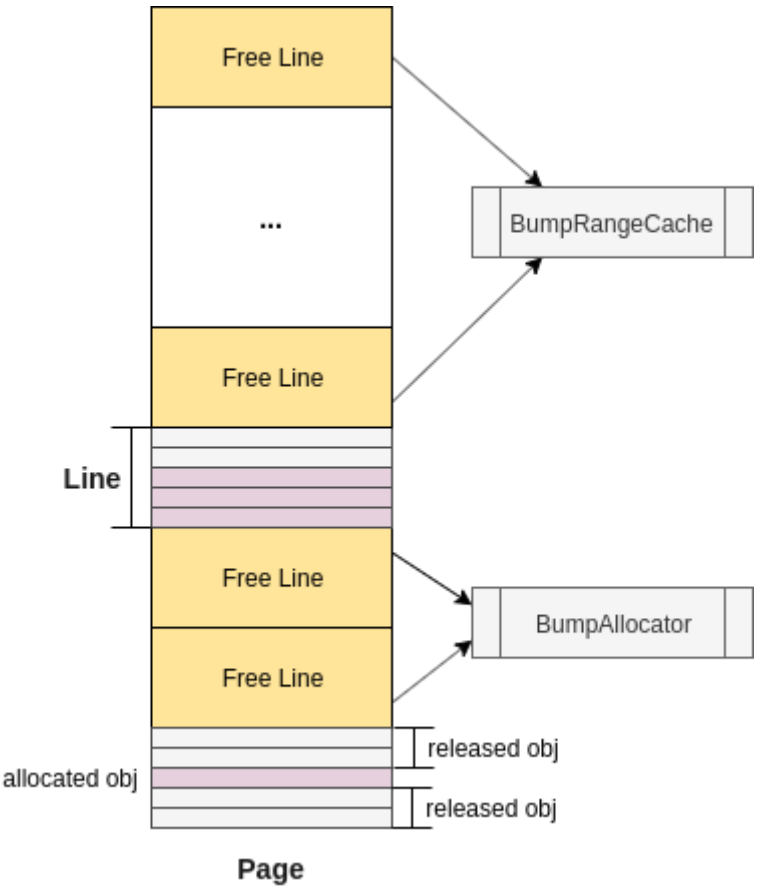
The primary heap is made of chunks that are split into `smallpages` (4 KB). A small page serves allocations for same-sized objects that are stored into `smalllines` (256 bytes).



The primary heap allocator is, in essence, a bump allocator. Objects are allocated using the `fastMalloc` primitive that simply does the following when serving allocations using the fast path.

```
--m_remaining;
char* result = m_ptr;
m_ptr += m_size;
return result;
```

When the bump allocator is running out of available free object slots, fastMalloc's slow path is triggered in order to refill the bump allocator. The allocator is refilled either from the cache (fast path) - namely the `bumpRangeCache` - or from a newly allocated page (slow path). Regarding the slow path, a page is picked up from the cache - namely the `lineCache` or retrieved from the list of free pages maintained for each chunk. In case of a fragmented page (e.g., page allocated from `lineCache`), the contiguous free
lines are used to refill the allocator. The rest of the free lines are used to fill the `bumpRangeCache`.



When an objected is freed, it is not made immediately available for subsequent allocations. It is inserted in a dedicated vector `m_objectLog` that is processed in `Deallocator::processObjectLog` when it reaches its maximal capacity (512) or while refilling the bump allocator through the slow path. The role of the `processObjectLog` is to release a `smallLine` if it's refcount reaches 0 (e.g., all slot objects in the `smallLine` are not used). When a `smallLine` is released, it's associated page is pushed into the `cacheLine`. Note that `smallPages` and `chunks` are also refcounted and therefore released when their respective refcount reaches 0.

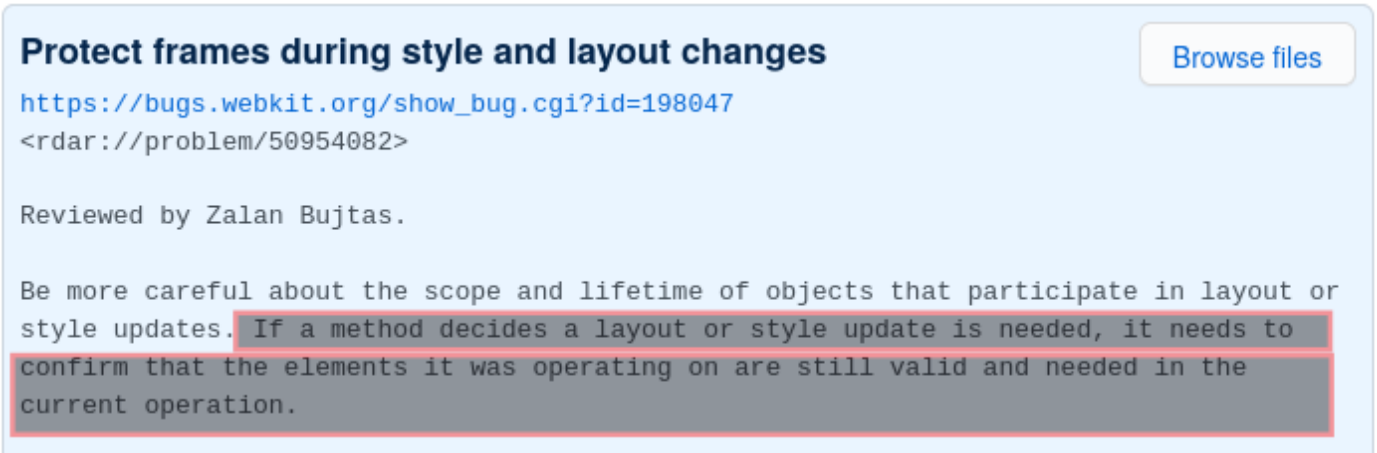
THE BUG

The bug has been triggered by our internal fuzzer. It is present in WebKit DOM engine. More precisely the bug is present in `WebCore::ValidationMessage::buildBubbleTree` method:

```
void ValidationMessage::buildBubbleTree()
{
    /* ... */
    auto weakElement = makeWeakPtr(*m_element); // [1] flawed weak pointer

    document.updateLayout();                    // [2] call user registered JS events

    if (!weakElement || !m_element->renderer())
        return;

    adjustBubblePosition(m_element->renderer()->absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */
}
```

The method `buildBubbleTree` makes a call to update the layout during which all user registered JS handlers are executed. If the `ValidationMessage` is destroyed in a JS callback, this could lead to a Use-After-Free situation when we get back to `buildBubbleTree` code.

The WebKit developers have identified that problems may arise while updating the style or the layout. However, they failed to fix the code due to an extra dereference while making a weak pointer. That is, we can still destroy the `ValidationMessage` instance during a layout update.



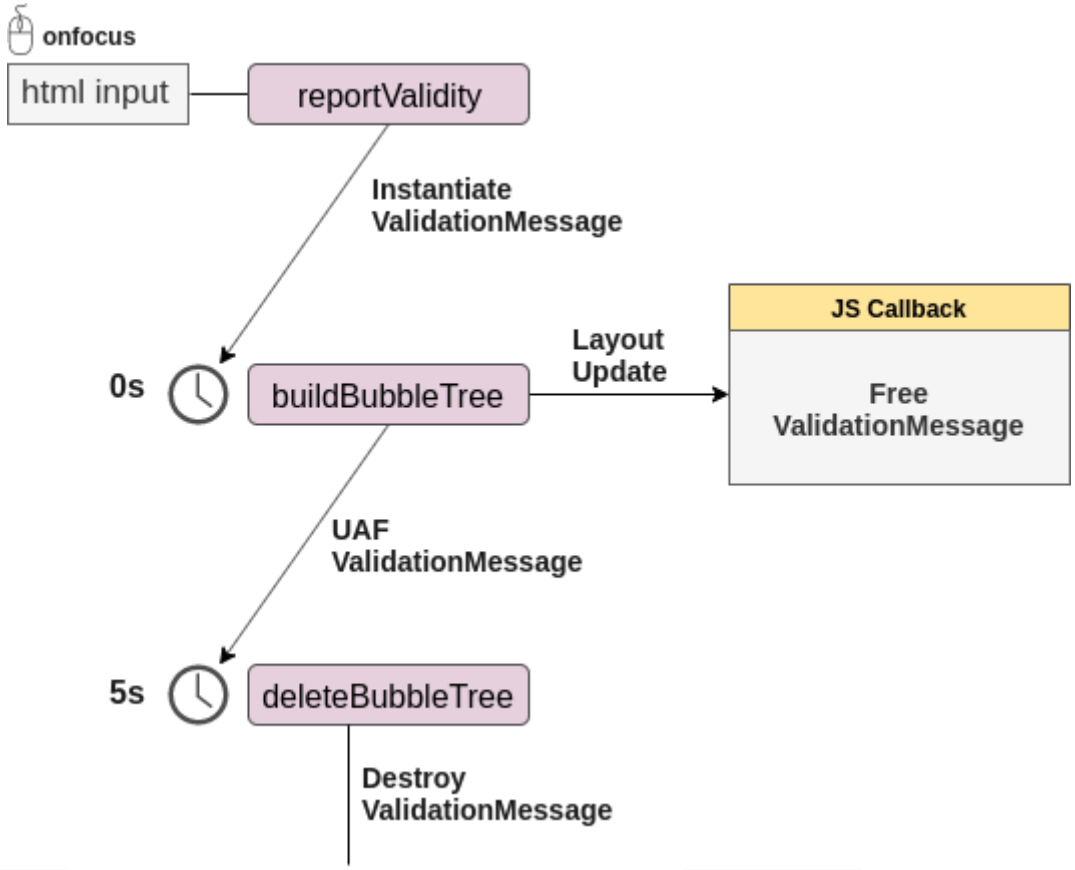
Hereafter is the fix after the reporting of the vulnerability:

```
void ValidationMessage::buildBubbleTree()
{
    /* ... */
-
-    auto weakElement = makeWeakPtr(*m_element);
-
-    document.updateLayout();
-
-    if (!weakElement || !m_element->renderer())
-        return;
-
-    adjustBubblePosition(m_element->renderer()->absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */


+    if (!document.view())
+        return;
+    document.view()->queuePostLayoutCallback([weakThis = makeWeakPtr(*this)] {
+        if (!weakThis)
+            return;
+        weakThis->adjustBubblePosition();
+    });
}
```

TRIGGERING THE BUG

The following figure summarizes the vulnerable path. We can instantiate a `ValidationMessage` object by invoking `reportValidity` on some HTML input field. We register on that input field a JS handler. For instance, we can define a JS

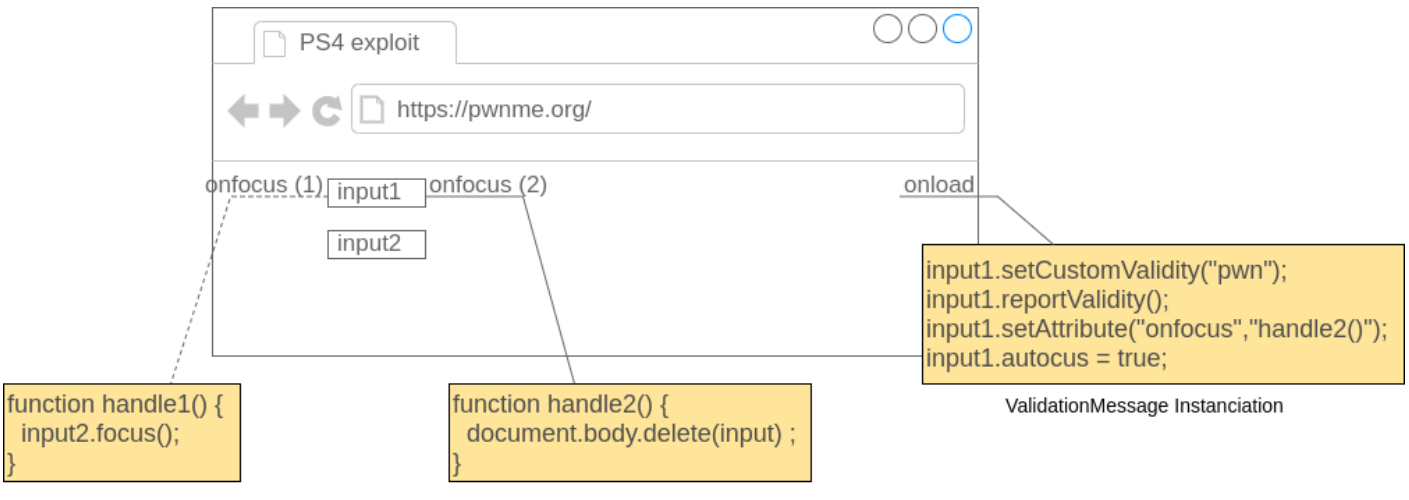
handler that is executed whenever the focus is set on the input field.



The `reportValidity` methods fire-up a timer to call the vulnerable function `buildBubbleTree`. If we set the focus on the HTML input field before timer expiration, `buildBubbleTree` will execute our JS handler during the layout update. If the `ValidationMessage` is destroyed in the JS callback, this leads to a UAF when getting back to `buildBubbleTree` code.

Now, if we manage somehow to survive to early crashes due to invalid access to `ValidationMessage` fields, we end up calling `deleteBubbleTree` that will destroy the `ValidationMessage` instance one more time.

Our first attempt to trigger the bug failed. `reportValidity` sets the focus on the input field which triggers our JS callback too early. As a workaround, we used two input fields: `input1` and `input2`. First, we register on the first input a JS handler on focus event. Then, we invoke `reportVality` on `input1`. This will trigger the execution of our JS handler that will simply set the focus elsewhere (e.g., on `input2`). Finally, before `buildBubbleTree` gets executed, we define a new handler on `input1` that will destroy the
`ValidationMessage` instance.



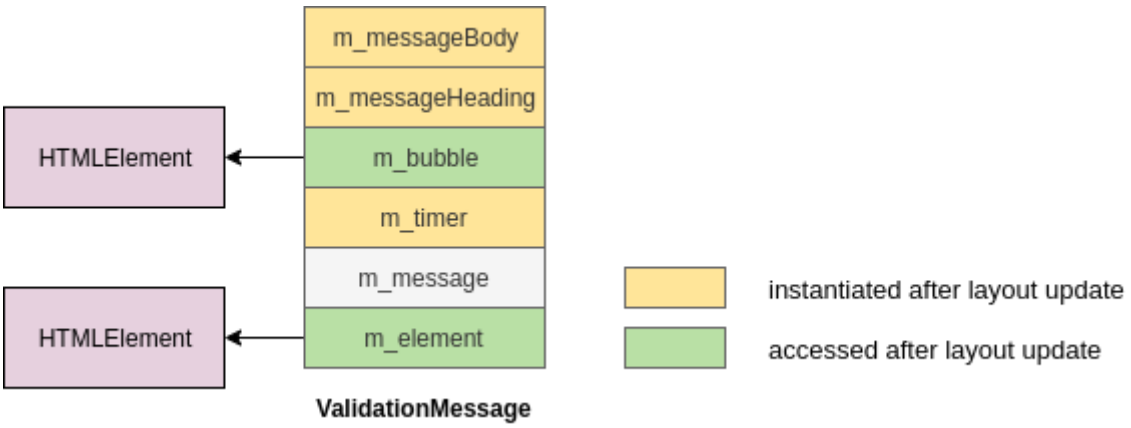
DEBUGGING THE BUG

Triggering the bug on the PS4 leads to a browser crash with zero debugging information. To overcome this limitation, we have two options:

1. Setup a debugging environment as close as possible to the PlayStation environment. That is install a FreeBSD box and build WebKit from sources downloaded from doc.dl.playstation.net. This is helpful but sometimes an exploit which works on our environment does not mean that it will work on the PS4.
2. Debug a 0-day vulnerability with a 1-day vulnerability. For that purpose, we can use the `bad-hoist` exploit as it provides useful primitives such as the read/write primitives but also the classical addrof/fakeobj primitives. However, this exploit is not reliable and works only on firmware 6.xx. Also, running this exploit prior to ours adds some noise on heap shaping.

ANATOMY OF A VULNERABLE OBJECT

The `ValidationMessage` object is created by `reportValidity` and is mainly accessed by `buildBubbleTree`. This object is fastmalloc'ed and is made of the
following fields.



The yellow fields are instantiated (`m_messageBody`, `m_messageHeading`) or gets re-instantiated (`m_timer`) after a layout update. The green fields
(`m_element` and `m_bubble`) each points to a `HTMLElement` instance and are accessed after a layout update.

The destruction of the `ValidationMessage` is done by the `deleteBubbleTree` method:

```
void ValidationMessage::deleteBubbleTree()
{
    if (m_bubble) {
        m_messageHeading = nullptr;
        m_messageBody = nullptr;
        m_element->userAgentShadowRoot()->removeChild(*m_bubble);
        m_bubble = nullptr;
    }
    m_message = String();
}
```

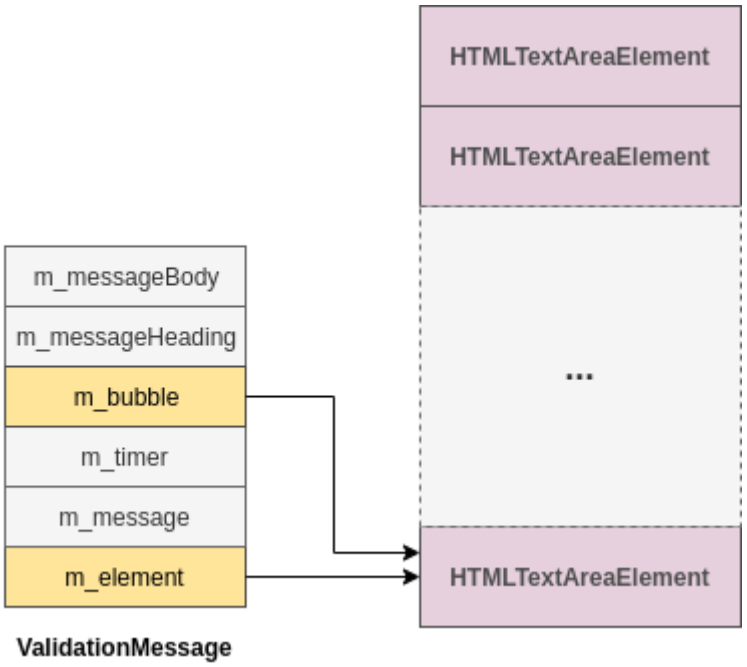
This method clears most of the `ValidationMessage` fields (including `m_bubble`) causing the browser to crash if it is invoked during a layout updating. More precisely, the renderer process crashes in the function `adjustBubblePosition` while attempting to dereference `m_bubble.get()`.

SURVIVING AN (INEVITABLE) CRASH

In order to exploit this vulnerability, we need either a memory leak or an ASLR bypass. It turns out that we can allocate some objects at a predictable location by spraying the heap. However, the exploitation of this weakness requires a prior knowledge on memory mapping. We already have this for firmware 6.xx thanks to the `bad-hoist` exploit. The predicted address is theorically bruteforcable on 7.xx firmware (more on this later).

Here's how to avoid an early crash:

1. Spray `HTMLElement` objects (e.g. `HTMLTextAreaElement`) → Objects allocated at a fixed location.
2. Confuse `ValidationMessage` with a controlled object (e.g. `ArrayBuffer`'s content).
3. Fixe the `m_bubble` and `m_element` values so that they point to the address that we predicted.

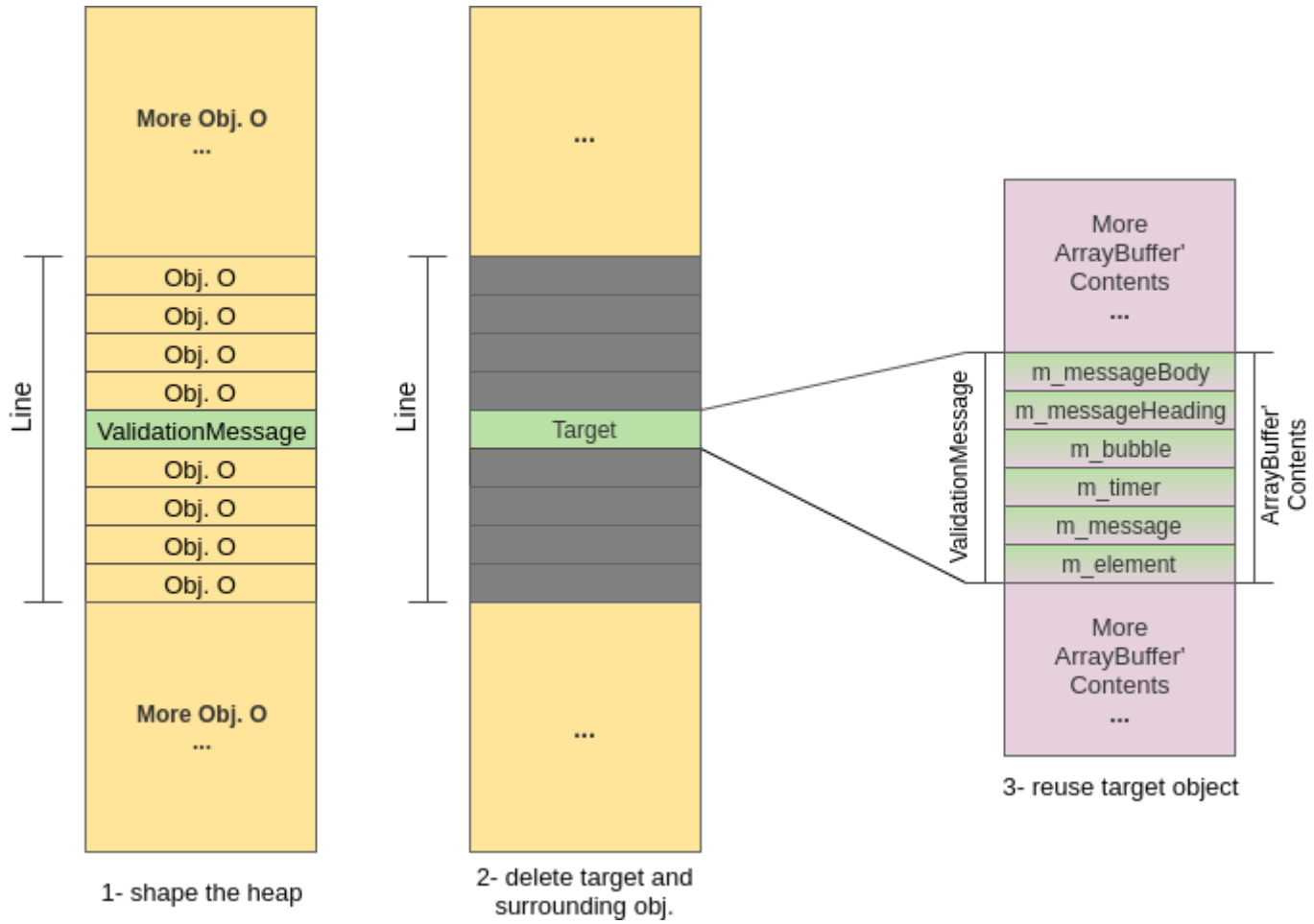


EXPLOITATION STRATEGY

REUSING TARGET OBJECT

In order to reuse our target object, we proceed as depicted in the following figure:

1. We spray the heap with objects `0` - having the same size as `ValidationMessage` (48 bytes) - right before and after the allocation of our target object.
2. We delete the `ValidationMessage` instance as well as the surrounding `0` objects causing the `smallLine` holding those objects to be released and the related `smallPage` to be cached.
3. We spray again the heap with objects having the same size as the target object. In our exploit, we spray with `ArrayBuffer(48)` in order to override `ValidationMessage` content with the `ArrayBuffer`'s content.



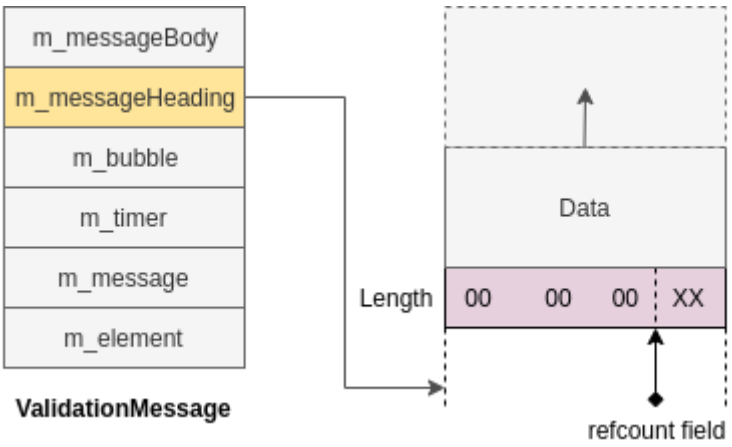
INITIAL MEMORY LEAK

As stated previously, some of the `ValidationMessage` are instantiated after the layout update and therefore their content could be leaked. More precisely, by simply dumping the content of the `ArrayBuffer`, we can leak the value of `m_messageBody`, `m_messageHeading`, and `m_timer`. `m_timer` is particularly interesting because it is fastMalloc'ed and hence allow us to infer the address of objects allocated on the same `smallPage`.

THE ARBITRARY DECREMENT PRIMITIVE

Now, if we use our target object properly, we end up calling the `deleteBubbleTree` method that sets most of the `ValidationMessage` fields to a NULL pointer value. It is important to note that NULL pointer assignment on refcounted classes is overloaded and results in a refcount decrement. This means that we have a refcount decrement on multiple controlled `ValidationMessage`'s fields: `m_messageBody`, `m_messageHeading` and `m_bubble`.

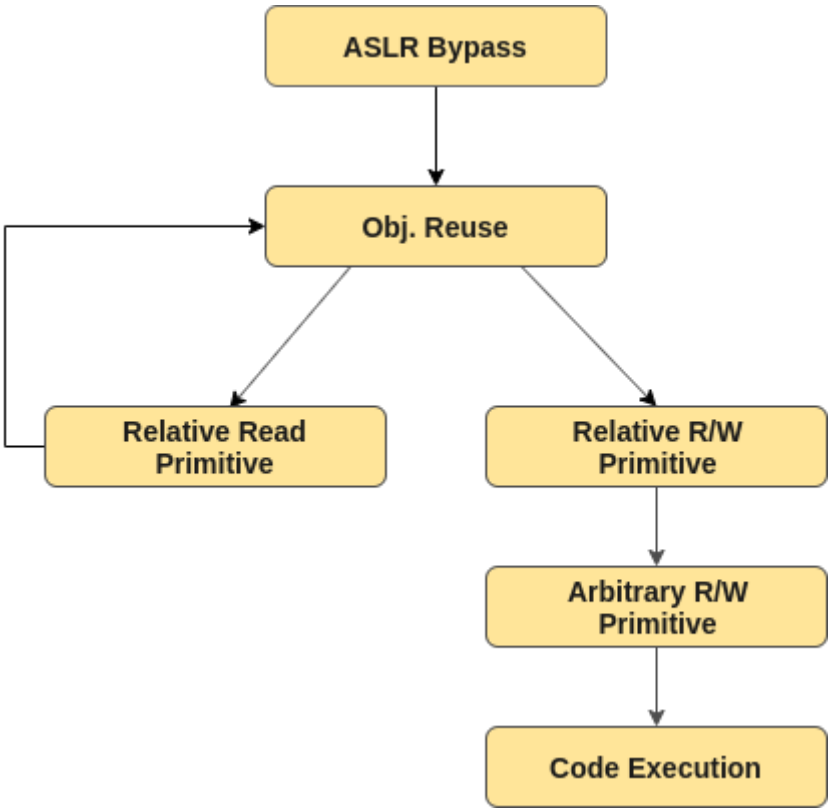
We can exploit the refcount decrement by corrupting for instance the `m_messageHeading` pointer and confusing its `refcount` field with some object's length field. By targeting for example an object with a length and data field such as the `StringImpl` object, the misaligned write on the `length` field would allow us to read beyond the data limit.



In the following, we will exploit the arbitrary decrement primitive twice:

1. The goal of the first run is to set up a relative read primitive in order to leak the address of a `JSArrayBufferView`.
2. The goal of the second run is to set up a relative read/write primitive by corrupting the length field of the previously leaked `JSArrayBufferView` address.

The exploitation workflow is summarized in the following figure:



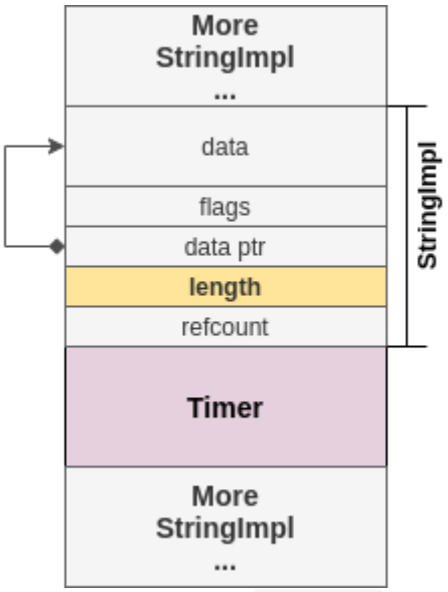
THE RELATIVE READ PRIMITIVE

The goal of this part is to leak the address of a `JSArrayBufferView` object. This object is interesting from our point of view because it has a length field and it allows to read/write arbitrary data into a data buffer. If we can control the length field, then we can read/write beyond the limit of the data buffer. This object is allocated using the Garbage Collector. As we don't have any leak of this heap, we can't reach it using our arbitrary decrement primitive.

`StringImpl` is the JavaScript string representation. This object has a length field and it allows reading data into a buffer. In JavaScript, strings are immutable. This means that we can read but not write into the data buffer after initializing it. The object is allocated using `fastMalloc` and the size of the object is partially under our control. `StringImpl` are therefore a good target to get a relative read primitive.

This is our strategy to overwrite the length of one `StringImpl` object:

1. Spray multiple `StringImpl` objects before and after `m_timer` instantiation. We allocate `StringImpl` such that they have the same size as a timer object in order to ensure that `m_timer` and `StringImpl` reside on the same `smallPage`.
2. Exploit the arbitrary decrement primitive on the length field of one sprayed `StringImpl`.
3. Iterate over the `StringImpl` objects and check which one has a corrupted length (e.g., the one that has a huge length is the right one).



Once we have our relative primitive through a corrupted `StringImpl`'s length field, let's see how we can get `JSArrayBufferView` references on the FastMalloc heap.

`Object.defineProperties` is a JavaScript built in that allows defining JavaScript Object properties.

```
static JSValue defineProperties(ExecState* exec, JSObject* object, JSObject* properties)
{
    Vector<PropertyDescriptor> descriptors;
    MarkedArgumentBuffer markBuffer;

    /* ... */
    JSValue prop = properties->get(exec, propertyNames[i]);
    /* ... */
    PropertyDescriptor descriptor;
    toPropertyDescriptor(exec, prop, descriptor);
    /* ... */
    descriptors.append(descriptor);        // [1] store JSValue reference on fastMalloc
    /* ... */
    markBuffer.append(descriptor.value()); // [2] store one more JSValue reference on fastMalloc
}
```

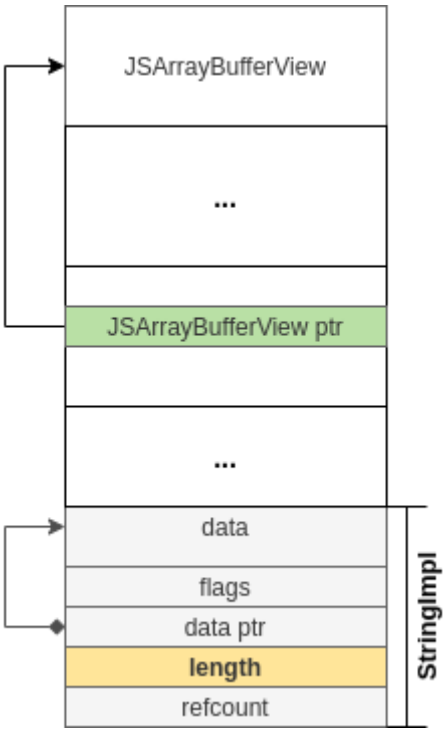
In its implementation, this method uses two objects:

- `Vector<PropertyDescriptor>`
- `MarkedArgumentBuffer`

Both objects have a backing buffer allocated using `fastMalloc` and both objects are used by `defineProperties` to store `JSValue` references. A `JSValue` is used by the JavaScript engine to represent many JavaScript values. This object is interesting for us because it can be a `JSObject` reference (e.g. `JSArrayBufferView`). Thus allowing us to push `JSObject` references on FastMalloc heap. Using our FastMalloc relative read, we can find these references.

This is the technique used by @qwertyoruiopz to get JSObject references on the FastMalloc heap:

1. Allocate multiple `JSArrayBufferView` objects.
2. Push references on FastMalloc heap using `Object.defineProperties.` The references are freed but are not cleared when leaving `Object.defineProperties`. We must be careful to not re-use these allocations to avoid clearing the `JSValue` references.
3. Use the relative read to scan the FastMalloc heap and search for the sprayed JSArrayBufferView references. The targeted reference must be reachable from the relative read.

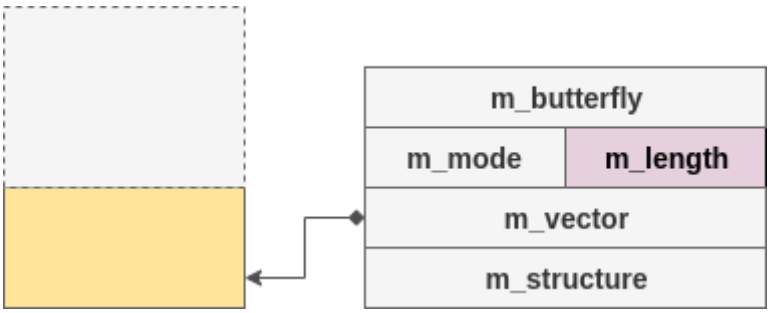


THE RELATIVE READ/WRITE PRIMITIVE

With a `JSArrayBufferView` leak, we can easily get a relative read/write using the following steps:

1. Run the exploit again to re-use the arbitrary decrement primitive.
2. Use the arbitrary decrement to overwrite the `JSArrayBufferView`'s length field. The overwritten `JSArrayBufferView` can be found by checking the length of each sprayed object. The huge one is the right one.

This reference can be used to read/write beyond the limit of the fastMalloc'ed backing buffer.

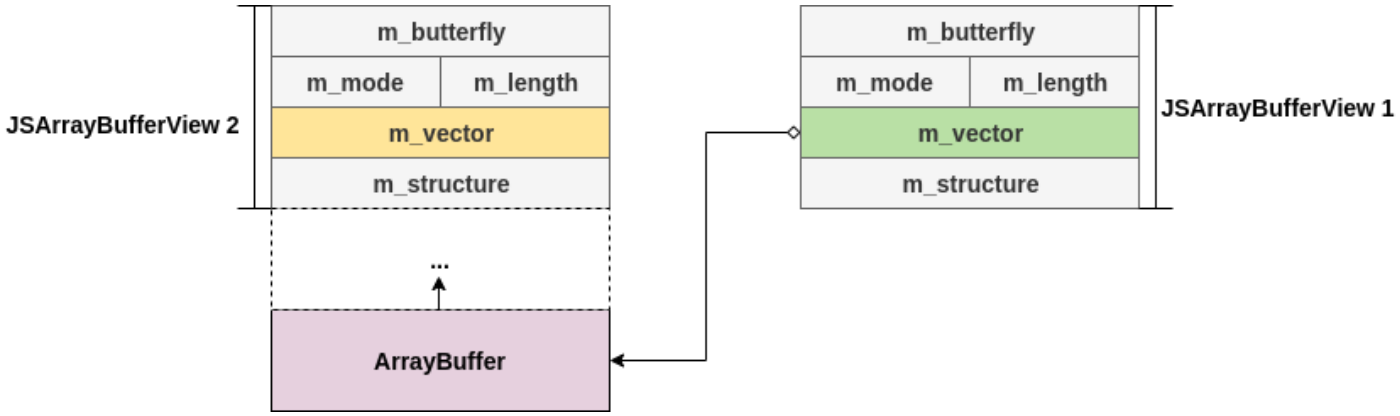


THE ARBITRARY READ/WRITE PRIMITIVE

`JSArrayBufferView` object has a reference field to its data buffer. Corrupting this reference enables an arbitrary read/write primitive.

We still have a relative read, the memory address of the relative read data buffer and we know that the leaked `JSArrayBufferView` is reachable from the relative read data buffer. We can easily find the memory location of the relative read/write backing buffer thanks to the relative read.

Now, we can use the backing buffer reference to reach one of the sprayed `JSArrayBufferView` using the relative read/write and corrupt the backing buffer reference of another `JSArrayBufferView` object. This allows reading and writing arbitrary data at an arbitrary address using the second `JSArrayBufferView` reference as depicted by the following figure:



CODE EXECUTION

PS4 browser doesn't allow allocating RWX memory pages. But with an arbitrary R/W, we can control RIP register. For example, we can overwrite the vtable pointer of one of our previously sprayed `HTMLTextAreaElement` making it points to data that we control. Calling a JavaScript method on `HTMLTextAreaElement` will result in a call to an address under our control. From there, we can do code re-use (like ROP or JOP) to implement the next stage.

The exploit is available at Synacktiv's Github repository.

PORTING THE EXPLOIT ON 7.XX FIRMWARE

We managed to successfully exploit our bug on 6.xx firmware due to a weakness on ASLR implementation that allowed us to predict the address of HTML objects. The predictable address is hardcoded in the exploit and has been identified thanks to the `bad-hoist` exploit. However, without a prior knowledge on memory mapping, the only way to determine the address of our sprayed HTMLElement objects is to brute force this address.

Brute forcing on the PS4 is tedious as the browser requires a user interaction in order to restart. Our idea is to plug a Raspberry Pi that acts as a keyboard on the PS4. Its main goal is to hit enter at periodical time (5s) to restart the browser after the crash. The brute forced address is updated at each attempt and stored in a cookie.