Blink-in-JavaScript

Kentaro Hara (haraken@chromium.org)

What is it?

- Blink-in-JavaScript is a mechanism to enable Blink developers to implement DOM features in JavaScript (instead of C++)

Team

- haraken@
- tasak@
- yosin@
- yoicho@
- jochen@
- dcarney@

Agenda

- Concept

- Design

- Implementation
    - The main issue is how to ensure security

Concept

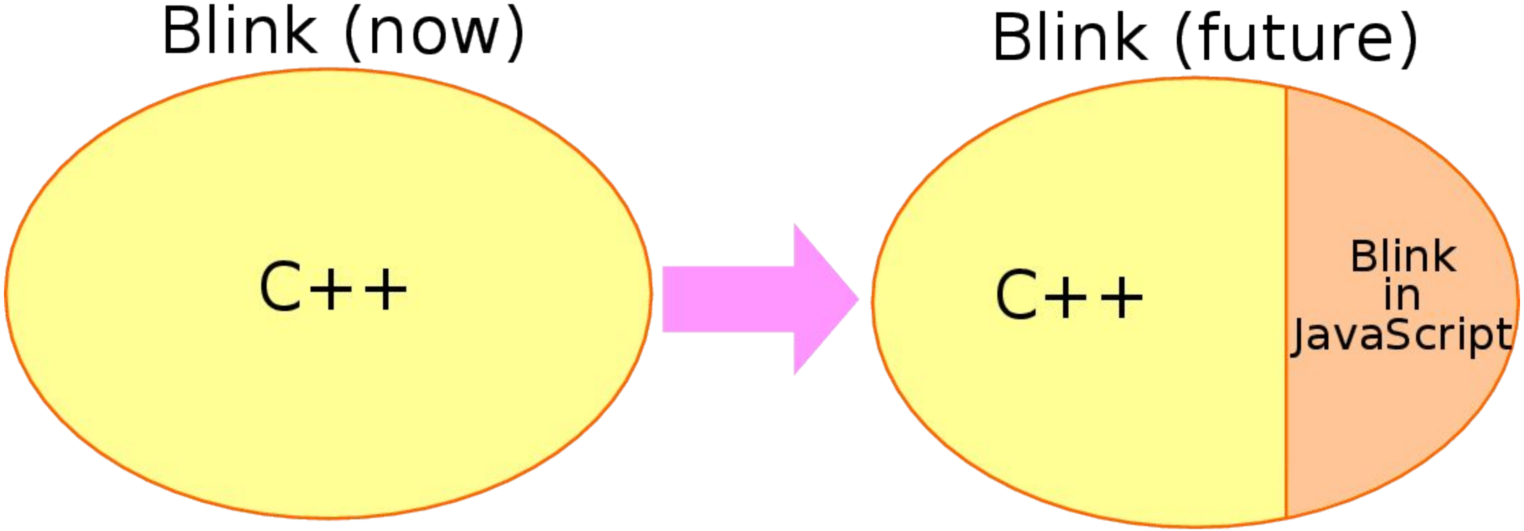
Motivation

- C++ causes a lot of security bugs
- C++ is hard to maintain

- If we can implement more things in JS, we can make Blink more secure and easier to maintain

The basic idea

- Implement only the core part in C++
- Implement other parts in JS on top of existing, web-exposed JS APIs



Targets of Blink-in-JS

- High-level DOM features that can be easily implemented on top of existing JS APIs

- DOM features that are unloved and should be factored out from C++

- DOM features that are going to be deprecated

- DOM features that are going to be implemented in C++ in the near future (i.e., Polyfil)

Targets of Blink-in-JS

- Examples:
    - XSLT
    - Editing’s execCommand()
    - A bunch of editing APIs
    - ScriptRegexp
    - Node.normalize()
    - DOMWindow.atob()/btoa()
    - ...

Example: XSLT

- XSLT adds a lot of complexity to the code base

- We do want to remove it, but can’t because of non-negligible number of users (in enterprise area)

- So let’s factor it out from C++ to JS!

Example: Editing APIs

- Editing APIs have a ton of use-after-free bugs

- Editing APIs can be implemented on top of existing JS APIs

- Most of them are not performance-sensitive

- So let’s move it to JS :)

Summary

- The goal of Blink-in-JS is improving:
    - maintainability
    - security
    - layering of the web architecture

- The goal of Blink-in-JS is NOT improving:
    - performance
    - power
    - memory

Better maintainability

- Maintainability matters

- Simplifying the code base allows us:
    - to make performance improvements
    - to add more important features more quickly

Better layering of the web architecture

- Better layering improves security

- Currently we implement everything in C++, so we need to ensure security for everything…

- If we implement only the core part in C++ and other parts in JS, we just need to ensure security for the C++ part and the JS engine

Wait!

- What about performance/power/memory? Won’t they regress?

Performance & power

- Problem:
    - JS is slower than C++ (and thus consumes more power)

- Solution:
    - Performance-sensitive features are not the target of Blink-in-JS

Memory

- Problem:
    - JITed JS code is 20x~ larger than C++ binary

- Solution:
    - Blink-in-JS is lazily complied (it’s not compiled until the feature is requested)
    - The compiled code is discardable anytime (the code is recompiled when the feature is requested again)

Summary

- Blink-in-JS enables Blink developers to implement DOM features in JS

- The goal is to improve:
    - maintainability
    - security
    - layering of the web architecture

Design

Programming model

- It’s easy; you just need to:
    - add [ImplementedInJS] to DOM attributes/methods in IDL files
    - implement the DOM attributes/methods in JS

- Then, necessary binding code will be auto-generated

Programming model

```
// WindowBase64.idl
interface WindowBase64 {
  [ImplementedInJS] DOMString atob(DOMString str);
};

// WindowBase64.js
installClass(“WindowBase64”, function() {
  return {atob: function atob(str) {
    // Here |this| is equal to |window|.
    return base64Encode(str);
  }};
});
```

Notes

- It’s also possible to use Blink-in-JS (not through IDL but) from inside Blink

- What Blink-in-JS can use is limited to web-exposed JS APIs
    - Future work: Expose internal APIs that are visible only to Blink-in-JS

How it works

- Blink-in-JS is lazily compiled at the first time the DOM attribute/method is accessed

- Blink-in-JS is executed in the same security level as Chrome extensions

Security model

- The problem is that we cannot execute Blink-in-JS in the same “world” (explained later) as user’s JS

- ...because Blink can have confidential information that should not be exposed to user’s JS
    - File names in an <input> element
    - Contents of a clipboard

Security model

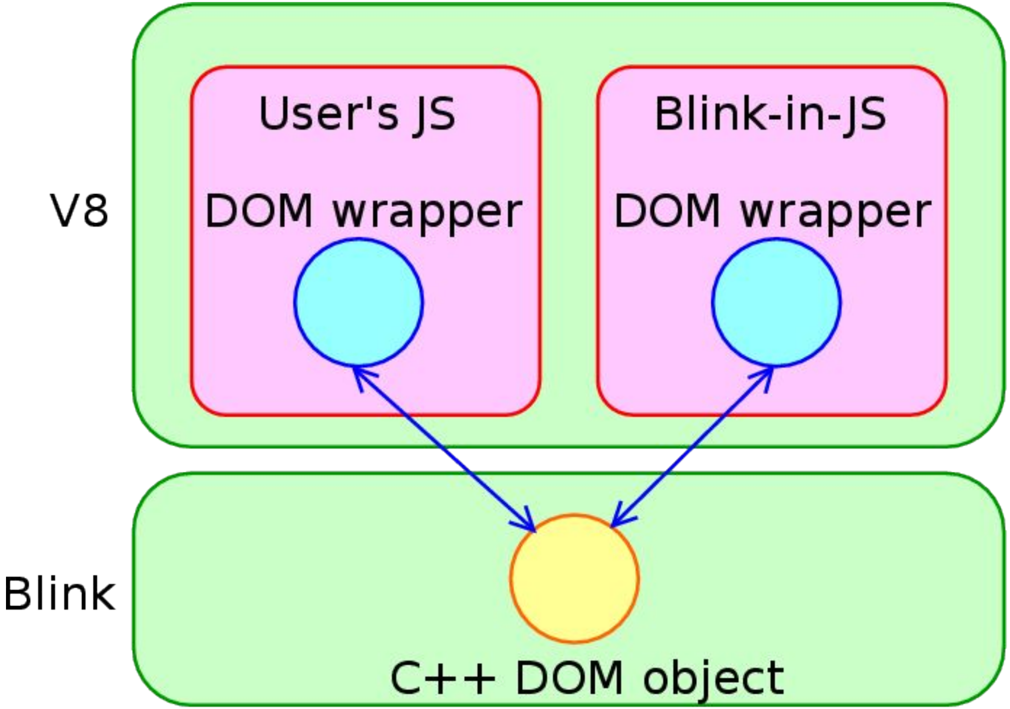
- Requirements:
    - Blink-in-JS and user’s JS need to operate the same C++ DOM objects
    - However, JS objects should not leak between Blink-in-JS and user’s JS

- In short, underlying C++ DOM objects should be shared between Blink-in-JS and user’s JS, but JS objects should be isolated

Security model

- C++ DOM objects are shared, but their DOM wrappers are separated

- ...and thus guarantees that no JS objects leak between Blink-in-JS and user’s JS



Security model

- This is exactly what Chrome extensions are doing
    - using a concept of “world” (explained later)

- So Blink-in-JS uses the same infrastructure and guarantees the same level of JS isolation
    - Blink-in-JS is “a Chrome extension inside Blink”
    - Blink-in-JS switches the world whenever it is entered/exited

Summary

- It’s easy to use Blink-in-JS

- Blink-in-JS is lazily compiled

- Blink-in-JS is executed in the same security level as Chrome extensions

Implementation

(Mostly about how to ensure security)

I mentioned...

- Chrome extensions guarantee security using a concept of “world”

- Blink-in-JS uses the same infrastructure and guarantees the same level of JS isolation as Chrome extensions

However...

- The problem is that the implementation of the “world” is broken :-/
    - JS objects sometimes leak among worlds...

- We must fix it; it’s not only for Blink-in-JS but also for all Chrome extensions

What’s the problem?

- To understand the problem, you need to understand complicated concepts in V8 bindings:
    - Isolate
    - Context
    - World

- I will explain these now :)

Isolate

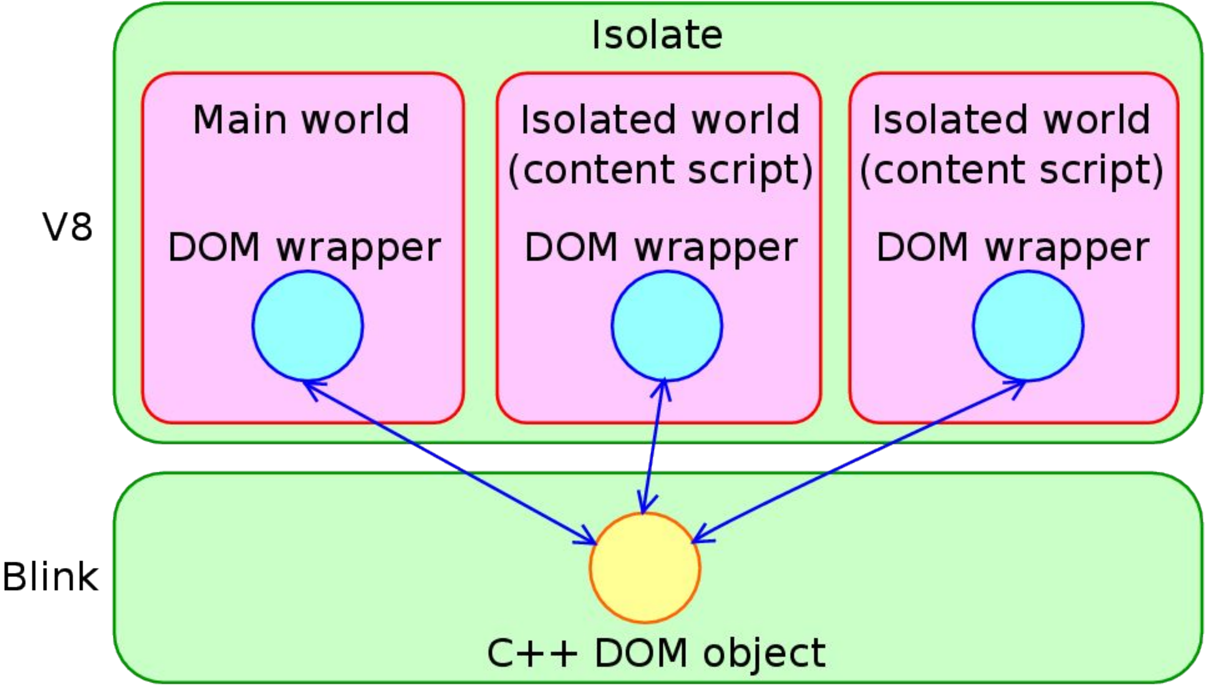
- An isolate is a V8 concept, associated to each thread
    - One isolate is for the main thread
    - One isolate is for each worker thread

Context

- A context is a V8 concept, associated to a global variable scope
    - Roughly speaking, a context corresponds to a window
    - Each frame has its own window and thus its own context
        - e.g., window.foo in an <iframe> is different from window.foo in another <iframe>

World

- A world is a concept to sandbox DOM wrappers among content scripts of Chrome extensions



World

- In one isolate:
    - underlying C++ DOM objects are shared among worlds
    - but the DOM wrappers are separated

- Each world has its own context
    - e.g., Object.prototype is different per world

- Therefore, it is guaranteed that no JS objects leak among worlds

World

- A world is a concept to completely sandbox JS executions except underlying C++ DOM objects

- The current problem is that DOM wrappers can leak among worlds (and thus JS objects can leak among worlds)
    - e.g., A world can access a window object of another world...

Isolate, context, world

- Isolate = Thread

- Context = Global scope (window object)

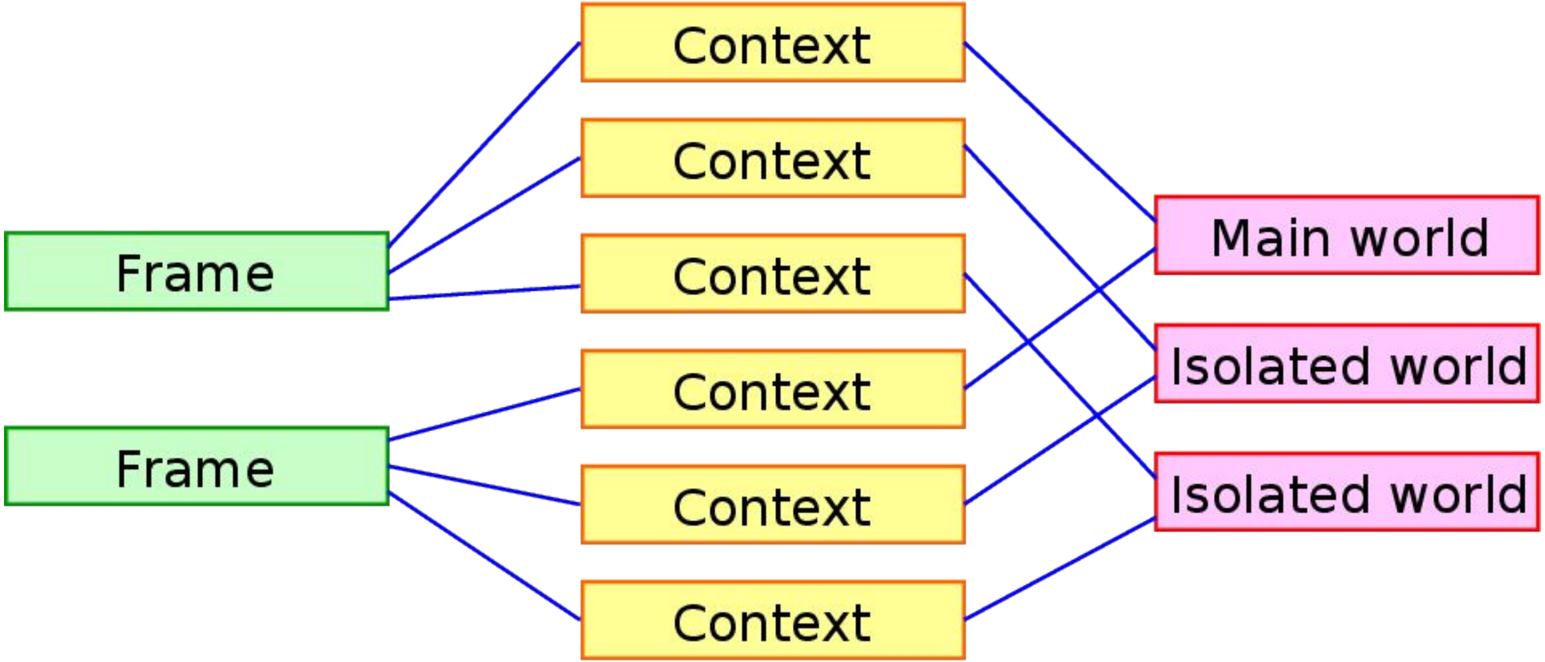
- World = Content script

Isolate, context, world

- Remember that:
    - Each frame has its own context
    - Each world has its own context

- This means that if one isolate has *x* frames and *y* worlds, there are *x**y* contexts involved

Isolate, context, world



- One global scope is needed for each pair of (page frame, content script)

Isolate, context, world

- Whenever you access DOM wrappers (e.g., when you call toV8()), you need to make sure that you are in a correct context

- Otherwise, you will end up returning DOM wrappers of another world, which will lead to cross-world leakage

```
// main.html
<iframe src=”iframe.html”></iframe><script>
var iframe = document.querySelector(“iframe”);
iframe.onload = function () {
  var div = iframe.contentDocument.querySelector(“div”); // The <div> wrapper should
be created in the context associated with the main frame and the current world
  div.onclick = function() { ... } /* This should be invoked in the context that
registered the event handler */
  div.click();
}
</script>

// iframe.html
<div></div><script>
var div = document.querySelector(“div”); // The <div> wrapper should be created in
the context associated with <iframe> and the current world
div.onclick = function() { ... } /* This should be invoked in the context that
registered the event handler */
</script>
```

Anyway, you must be in a correct context

(1) When the event handler is created, you need to get the current context and record it

(2) When the event handler is invoked (sometime later), you need to restore the context, and then invoke the event handler

Revisited: What’s the problem?

- Isolate, context and world are complicated

- People write binding code without understanding it

- People tend to use a current context when they don’t know what context they should use
    - The current context is not always equal to a correct context
    - It can lead to cross-world leakage...

Solutions

Solution 1: Invent a better programming model everyone can understand

Solution 2: Introduce dynamic verifications about cross-world leakage

Solution 1: Better programming model

- There are two cases where binding code is executed
    - Synchronous case: JS calls the binding code and immediately go back to JS
        - e.g., div.firstChild, div.appendChild()
    - Asynchronous case: JS calls the binding code and creates some proxy object, and then later Blink calls back the binding code through the proxy object
        - e.g., Event handlers, Promise

Solution 1: Better programming model

- The synchronous case is no problem

- Because JS is calling you, it’s already guaranteed that you are in a correct context

Solution 1: Better programming model

- The asynchronous case needs special handling

- The basic idea is:

  (1) When JS calls the binding code and creates a proxy object (e.g., V8EventListener), store the current context

  (2) When later Blink calls back the binding code through the proxy object (e.g., V8EventListener::handleEvent()), restore the context before accessing DOM wrappers

Solution 1: Better programming model

```
class V8ProxyObject { // e.g., V8EventListener
  V8ProxyObject() : m_state(ScriptState::current()) { }
  void someCallback() { // Blink calls back later
    if (m_state->contextIsEmpty()) // Context is already gone
      return;
    ScriptState::Scope scope(m_state.get()); // Enter the context
    ...;
  }
  RefPtr<ScriptState> m_state; // ScriptState piggybacks isolate,
context, world and all other information about script execution
};
```

Solution 2: Dynamic verifications

- Introducing ScriptState will fix cross-world leakage

- As a next step, it’s important to verify

- Specifically, we’re going to use:
    - ScriptValue
    - Security tokens

Solution 2: Dynamic verifications

- ScriptValue is a thin wrapper of a V8 value
- When a Blink object holds a V8 value, ScriptValue should be used

```
class V8EventListener {
  ScriptValue m_listenerFunction;
};
```

Solution 2: Dynamic verifications

- Verify that ScriptValue is always accessed from the world from which the ScriptValue is created

    - By doing this, we can verify that no V8 values held by Blink objects leak among worlds

```
class V8EventListener {
  ScriptValue m_listenerFunction;
};
```

Solution 2: Dynamic verifications

- A security token is a V8 concept to detect cross-context access
    - A context can have a security token
    - If a JS object accesses another JS object created from a context that has a different security token, V8 detects the error

```
x = ...; // An object from one context
y = ...; // An object from another context that has a
different security token
x.foo = y; // V8 detects the error and sets undefined
```

Solution 2: Dynamic verifications

- If we set the same security token on all contexts in the same world, V8 detects all cross-world leakage for us

```
x = document.xxx(); // xxx() returns a wrapper of one world
y = document.yyy(); // Assume that yyy() is mis-implemented
and returns a wrapper of another world
x.foo = y; // V8 detects the error and sets undefined
```

Solution 2: Dynamic verifications

- The security token is a perfect way to detect cross-world leakage

- The problem is that the current implementation is not yet perfect
    - We’re making it perfect :)

Summary

- We need to guarantee that no DOM wrappers leak among worlds

- This is not only for Blink-in-JS but also for all Chrome extensions

- We are fixing it by:
    - inventing a better programming model with ScriptState
    - introducing dynamic verifications

Conclusion

Conclusion

- Blink-in-JS enables developers to implement DOM features in JS

- The goal is to improve security, maintainability and layering of the web architecture

- The challenging part is to eliminate all cross-world leakage

Working plan

(1) Refactor confusing infrastructures about isolate, context and world

(2) Introduce ScriptState to the code base and fix all cross-world leakage

(3) Implement dynamic verifications about cross-world leakage

(4) Land the infrastructure of Blink-in-JS

(5) Move XSLT and editing/ to Blink-in-JS

- Now we’re working on (2) and (3)

Thanks!