JavaScriptCore CSI: A Crash Site Investigation Story

Jun 1, 2016

by Mark Lam

When debugging JavaScript bugs, web developers have Web Inspector which provides a debugger and many introspective tools to examine their code with. But when the bug is at a lower level in WebKit's JavaScript engine, JavaScriptCore (JSC), WebKit engineers will need to use a different set of tools to diagnose the issue.

Today, I'll describe some of these tools that WebKit engineers use by telling the story of how we diagnosed a real bug in the JSC virtual machine (VM). This story will take us through ...

- The Crash
  - Preparation Work for Debugging the Crash
  - Inspecting the Bug with a Debugger
- Isolating the Bug
  - Disabling JIT tiers with JSC Options
  - Increasing JIT Predictability
  - Reporting Compiled Functions
  - Filtering Functions to Compile by Bytecode Size
  - Filtering Functions to Compile using a Whitelist
  - Dumping Compiled Functions
- Debugging with Logging
  - A LLInt Probe and Counting VM Entries
  - Who is Modifying VM::calleeSaveRegistersBuffer?

    - Logging the Modifications to VM::calleeSaveRegistersBuffer
    - Printing from JIT code
    - Following the Trail
- Summary

The Crash ∞

We use the WebKit Layout Tests as our primary regression tests for the full WebKit stack. While running this test suite on an AddressSanitizer (ASan) build of WebKit, and we discovered a crash on one of the layout tests:

```
$ ./Tools/Scripts/run-webkit-tests LayoutTests/inspe
```

The crash stack trace is as follows:

```
==94293==ERROR: AddressSanitizer: heap-use-after-fre
READ of size 8 at 0x61e000088a40 thread T0
    #0 0x10f9536a0 in JSC::VM::exception() const (/V
    #1 0x110fce753 in JSC::JITCode::execute(JSC::VM*
    #2 0x110ee9911 in JSC::Interpreter::executeCall(
    #3 0x10fc5d20a in JSC::call(JSC::ExecState*, JSC
    #4 0x10fc5d6c5 in JSC::call(JSC::ExecState*, JSC
    #5 0x10fc5e1ed in JSC::profiledCall(JSC::ExecSta
...
```

From the stack trace, it appears that the crash is in the JSC VM. The original bug has already been fixed in revision 200879. This post will recount how we fixed the bug.

Preparation Work for Debugging the Crash ∞

The first thing we should do in our investigation is to check if this crash still reproduces on the latest tip of tree of

WebKit source. At the time of this investigation, that would be revision 200796.

Since the crash was on an ASan build, we need to build WebKit with the ASan configuration:

```
$ svn co -r 200796 http://svn.webkit.org/repository/
$ cd webkitDir
$ ./Tools/Scripts/set-webkit-configuration --asan
$ ./Tools/Scripts/build-webkit --debug
```

Next, we re-run the test:

```
$ ./Tools/Scripts/run-webkit-tests --debug LayoutTes
```

Fortunately, the crash reproduces reliably.

`run-webkit-tests` is a test harness script that ultimately invokes either the `WebKitTestRunner` (WKTR) or `DumpRenderTree` (DRT) executables to run the tests. WKTR runs on WebKit2, which is a multi-process architecture. DRT runs on WebKit1 (a.k.a. WebKitLegacy), which is single processed. By default, `run-webkit-tests` runs WKTR because WebKit2 is what all modern WebKit browsers should be built on. But for our debugging purpose, it would be simpler to work with DRT instead.

Here, we will try to reproduce the crash by running the test with DRT directly instead of going through the `run-webkit-tests` test harness:

```
$ VM=WebKitBuild/Debug/ && \
DYLD_FRAMEWORK_PATH=$VM \
```

```
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Result: the crash still reproduces. Yay! Now we're ready to dive in and diagnose what went wrong.

Inspecting the Bug with a Debugger ∞

The first thing to do is to see what a debugger (*lldb*) can tell us about the crash:

```
$ VM=WebKitBuild/Debug/ && \
DYLD_FRAMEWORK_PATH=$VM \
lldb $VM/DumpRenderTree -- LayoutTests/inspector/deb
(lldb) run
```

The debugger runs DRT and quickly stops due to a bad memory access here:

```
frame #0: 0x000000010262193e JavaScriptCore`JSC::JIT
   78      } else
   79          entryAddress = addressForCall(MustChec
   80      JSValue result = JSValue::decode(vmEntryTo
-> 81      return vm->exception() ? jsNull() : result
   82  }
   83
```

At line 81 (where the debugger stopped), `jsNull()` is effectively a constant, and `result` should be a variable in a register. The only memory access we see there is the read of `vm->exception()` which accesses the VM's `m_exception` field (see VM.h). Looking at the source code (in JITCode.cpp), we see that `vm` is an argument that is passed into `JITCode::execute()` by its caller:

```
(lldb) up
frame #1: 0x0000000102507b58 JavaScriptCore`JSC::Int
   1017 {
   1018     // Execute the code:
   1019     if (isJSCall)
-> 1020         result = callData.js.functionExecuta
```

Looking at the code for `Interpreter::executeCall()` (in Interpreter.cpp), we see that `vm`'s value was used without triggering a crash before execution reached `JITCode::execute()`. This means `vm` had a valid value previously. `JITCode::execute()` also does not have any code that alters `vm` (note that `vmEntryToJavaScript()` takes a `VM*` not a `VM&`).

This is strange indeed. `vm` should be valid, but is not. Let's look at what the machine code is actually doing at this crash site:

```
(lldb) disassemble
...
   0x10262192e <+910>: callq 0x10376994c ; symbol st
   0x102621933 <+915>: movq 0x80(%rbx), %rax
   0x10262193a <+922>: movq 0x18(%rbx), %rcx
-> 0x10262193e <+926>: movq %rcx, (%rax)
...
```

We see that we're crashing while trying to store to the address in the `%rax` register. The value in `%rax` was computed using register `%rbx` just two instructions before. Let's look at the register values:

```
(lldb) reg read
General Purpose Registers:
```

```
    rax = 0x0000000045e0360e
    rbx = 0x00007fff5fbf6300
    ...
    rbp = 0x00007fff5fbfa230
    rsp = 0x00007fff5fbfa050
    ...
```

Notice that `%rbx` contains a value that is close to the value of the stack pointer, `%rsp`. Comparing their values, we see that `%rbx` (`0x00007fff5fbf6300`) is pointing to a lower address than the stack pointer (`0x00007fff5fbfa050`). Since the stack grows from high to low addresses, that means `%rbx` is pointing to a part of the stack that is not allocated for use by this frame. That explains why ASan flags this memory access as invalid, thereby yielding a crash.

This test run is on X86_64 hardware. Based on X86_64's application binary interface (ABI), the `%rbx` register is a callee save register. For example, let's say we have the following functions:

```
void goo() {
    ...    // Uses %rbx to do its work.
}

void foo() {
    ...    // Sets register %rbx to 42.
    goo(); // Let goo() do some work.
    ...    // %rbx should still be 42.
}
```

Because `%rbx` is a callee save register, the ABI states that function `goo()` must save `%rbx`'s value before using it, and, accordingly, restore the value before returning to its caller. From function `foo()`'s perspective, it doesn't have to save `%rbx` first before calling `goo()` because it is the

callee `goo()`'s responsibility to preserve the value of the register, hence the term *callee save register*.

Searching for `%rbx` in the full disassembled machine code for JITCode::execute(), we see that `%rbx` is set to the stack pointer at the top of the function:

```
0x10262162c <+12>:   pushq  %rbx
...
0x102621638 <+24>:   movq   %rsp, %rbx
```

... and never set again until it is restored at the end of the function:

```
0x102621abe <+1182>: popq   %rbx
```

Throughout the function (including before the crash point), `%rbx` is used to compute stack addresses that are read from and written to. We didn't crash on any of these prior uses of `%rbx`.

Because `%rbx` is a callee save register and there are no other writes to it in this function (between the top of the function and the crash point), we know that some callee of `JITCode::execute()` must have modified `%rbx` and failed to restore it before returning. JSC does have code to save and restore callee save registers in LLInt interpreter and Just-In-Time (JIT) compiler generated code. Since `JITCode::execute()` serves as the entry point to LLInt or JIT generated code, maybe the bug is in LLInt or JIT code.

Isolating the Bug ∞

JSC comes with multiple tiers of execution engines. You may have read about them here. To recap, there are 4 tiers:

- *tier 1*: the LLInt interpreter
- *tier 2*: the Baseline JIT compiler
- *tier 3*: the DFG JIT
- *tier 4*: the FTL JIT (now with our new B3 backend)

One way we can reduce the search area for the bug is by checking to see which of these tiers are required for the bug to reproduce.

Disabling JIT tiers with JSC Options ∞

In JSC's Options.h, you will find a list of options that can be used to configure how the JSC VM behaves at runtime. We can use these options by setting them as environmental variables when we invoke DRT like so:

```
$ VM=WebKitBuild/Debug/ && \
JSC_someOption=someValue \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

To test if any of the JIT tiers have any effect on the bug, we would like to disable each of the tiers and re-test for the bug. For that, we need the following options:

Option	Description	Tiers that can run	Tiers that cannot run
JSC_useJIT=false	Disables all JITs	LLInt	Baseline, DFG, FTL
JSC_useDFGJIT=false	Disables the DFG and above	LLInt, Baseline	DFG, FTL
JSC_useFTLJIT=false	Disables the FTL	LLInt, Baseline, DFG	FTL

Let's start from the lowest tier and move upwards from there i.e. let's only allow JavaScript (JS) code to run with the LLInt interpreter:

```
$ VM=WebKitBuild/Debug/ && \
JSC_useJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Result: the crash does not reproduce. That means the bug must lie in one or more of the JITs. Next, allow up to the baseline JIT tier:

```
$ VM=WebKitBuild/Debug/ && \
JSC_useDFGJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Result: the crash still does not reproduce. Looks like the bug may lie in the DFG or above. Next, allow up to the DFG JIT tier:

```
$ VM=WebKitBuild/Debug/ && \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Result: the crash reproduces. Ah hah! We have a crash when we allow up to the DFG tier to run. So, let's continue to leave the FTL out to simplify our debugging work. The next thing to do is to reduce our search area by compiling only a

minimum set of functions. Hopefully, that minimum set will consist of only 1 function.

Increasing JIT Predictability ∞

But before we do that, let's add one more useful option:

Option	Description
JSC_useConcurrentJIT=false	Disables concurrent compilation.

The DFG and FTL JITs may do their compilations in background threads. We can disable the use of these concurrent background threads, and force all DFG and FTL compilations to be synchronous with the execution of the JS code as follows:

```
$ VM=WebKitBuild/Debug/ && \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Result: the crash still reproduces. Good. The bug is not due to any race conditions with JIT compilation threads. Now, we can proceed with investigating which compiled JS function is triggering the crash.

Reporting Compiled Functions ∞

We can reduce the set of compiled functions by applying compilation filters. But before we can apply a filter, we must first know which functions are being compiled. We can find this out by telling the JITs to report their compile times for each function:

Option	Description
JSC_reportCompileTimes=true	Report all JIT compile times.
JSC_reportBaselineCompiletimes=true	Report only baseline JIT compile times.
JSC_reportDFGCompileTimes=true	Report only DFG and FTL compile times.
JSC_reportFTLCompileTimes=true	Report only FTL compile times.

Since we think the bug lies in the DFG, let's only report the DFG compile times. We do this by using *JSC_reportDFGCompileTimes=true* in combination with *JSC_useFTLJIT=false*:

```
$ VM=WebKitBuild/Debug/ && \
JSC_reportDFGCompileTimes=true \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

If any functions are DFG compiled, we should see logging on `stderr` for each compiled function that looks like the following:

```
Optimized foo#BAbkxs:[0x62d0000eb840->0x62d0000eba60
```

We read this logging like this:

```
Optimized <function name>#<function hash>:[<pointers
```

The `<function name>#<function hash>` together is called the *function signature*. The `<bytecode size of the function>` is the size of the interpreter bytecode that JSC generated for this function. We will be using these below.

Going back to our test case, we actually see no such DFG compile time logs. That's strange. We needed the DFG enabled in order to reproduce the crash, but no function was DFG compiled.

Let's go down a tier to the baseline JIT and see what functions get compiled there:

```
$ VM=WebKitBuild/Debug/ && \
JSC_reportBaselineCompileTimes=true \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

This time around, we get a few compile time logs:

```
Optimized isSymbol#BTGpXV:[0x62d000214100->0x62d0001
Optimized toString#D57Jzo:[0x62d000214dc0->0x62d0001
Optimized endsWith#AfTryh:[0x62d00028e300->0x62d0001
Optimized processDescriptor#DsYIGz:[0x62d00028e0e0->
Optimized createFakeValueDescriptor#BPpnwK:[0x62d000
Optimized processProperties#CgNq2F:[0x62d00028e520->
Optimized isPrimitiveValue#BLrwAH:[0x62d000215420->0
...
Optimized _isHTMLAllCollection#BlkszW:[0x62d00021520
```

```
Optimized _subtype#DYV24q:[0x62d000214320->0x62d0002
Optimized next#EXE83Q:[0x62d000217840->0x62d00023539
Optimized arrayIteratorValueNext#A7WxpW:[0x62d000217
```

The next step is to reduce this set of functions down to a minimum.

Filtering Functions to Compile by Bytecode Size ∞

From the baseline compile times logging, we can see that the functions that were baseline compiled are in a range of bytecode sizes from 48 to 1031.

We can filter functions to compile by limiting the bytecode size range to compile. We can do this using the following options:

Option	Description
JSC_bytecodeRangeToJITCompile=N:M	Only JIT compile functions whose bytecode size is between N and M (inclusive).
JSC_bytecodeRangeToDFGCompile=N:M	Only DFG compile functions whose bytecode size is between N and M (inclusive).
JSC_bytecodeRangeToFTLCompile=N:M	Only FTL compile functions whose bytecode size is between N and M (inclusive).

Let's try starting with the range 1:100:

```
$ VM=WebKitBuild/Debug/ && \
JSC_bytecodeRangeToJITCompile=1:100 \
JSC_reportBaselineCompileTimes=true \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

Now, we only get the following logs:

```
Optimized isSymbol#BTGpXV:[0x62d000214100->0x62d0001
Optimized toString#D57Jzo:[0x62d000214dc0->0x62d0001
Optimized _isHTMLAllCollection#BlkszW:[0x62d00021520
```

... and the crash still reproduces. We're on the right track.

We can continue to filter using reduced bytecode ranges, but let me take this opportunity to introduce you to another way to filter functions to compile ...

Filtering Functions to Compile using a Whitelist ∞

There are times when the minimum set of compiled functions needed to reproduce an issue is more than 1, and the bytecode sizes of those functions may not fit nicely in a contiguous range that excludes all other functions.

For example, let's say we have filtered a list of functions down to 3:

```
foo#Abcdef with bytecode size 10
goo#Bcdefg with bytecode size 50
```

```
hoo#Cdefgh with bytecode size 100
```

... and we want to check if we can reproduce an issue with only `foo` and `hoo`. For that, we can't use the bytecode range filter since the range that contains `foo` and `hoo` (10:100) will also allow `goo` to be compiled. Instead, for such cases, we need to be able to filter by function signature instead:

Option	Description
JSC_jitWhitelist= <whitelist file>	Only JIT compile functions whose signatures are in the whitelist file.
JSC_dfgWhitelist= <whitelist file>	Only DFG compile functions whose signatures are in the whitelist file.

Let's apply this to the list of functions in our investigation. First, we'll create a file with the remaining function signatures we saw from the compile time logs:

```
$ vi whitelist.txt
isSymbol#BTGpXV
// toString#D57Jzo
// _isHTMLAllCollection#BlkszW
```

Each of the signatures must be on their own line in the whitelist file, and each entry must start on the first character of the line. The whitelist mechanism supports C++ style comments that start with `//`. When used, the `//` must also start on the first character of the line. Notice that we commented out the 2nd and 3rd function signatures. This is so that we can test for one signature at a time. We'll start with testing only the 1st one: `isSymbol#BTGpXV`.

Since we're looking at baseline JIT compilations, let's use the *JSC_jitWhitelist* option:

```
$ VM=WebKitBuild/Debug/ && \
JSC_jitWhitelist=whitelist.txt \
JSC_reportBaselineCompileTimes=true \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

... and it still crashes. That's exactly what we're hoping for i.e. we only need to debug one compiled function: `isSymbol#BTGpXV`.

Dumping Compiled Functions ∞

Next, we can dump the compilation artifacts of the compiled function, and see if we spot anything wrong in them. We can do this using the following options:

Option	Description
JSC_dumpDisassembly=true	Dumps disassembly of all JIT compiled functions.
JSC_dumpDFGDisassembly=true	Dumps disassembly of DFG and FTL compiled functions.
JSC_dumpFTLDisassembly=true	Dumps disassembly of FTL compiled functions.
JSC_dumpSourceAtDFGTime=true	Dumps the source of the DFG / FTL compiled functions.
JSC_dumpBytecodeAtDFGTime=true	Dumps the bytecode of the DFG / FTL compiled functions.
JSC_dumpGraphAfterParsing=true	Dumps the DFG graph after parsing the function bytecode at DFG / FTL compilation time.

Option	Description
JSC_dumpGraphAtEachPhase=true	Dumps the DFG graph after each phase of DFG / FTL compilation.

Let's use *JSC_dumpDisassembly=true* to see what is being generated by the baseline JIT for our function `isSymbol#BTGpXV`.

```
$ VM=WebKitBuild/Debug/ && \
JSC_dumpDisassembly=true \
JSC_jitWhitelist=whitelist.txt \
JSC_reportBaselineCompileTimes=true \
JSC_useConcurrentJIT=false \
JSC_useFTLJIT=false \
DYLD_FRAMEWORK_PATH=$VM \
$VM/DumpRenderTree LayoutTests/inspector/debugger/re
```

We get a dump for `isSymbol#BTGpXV` that looks something like this (for brevity, I have shortened this dump by omitting many sections of generated code):

```
Generated Baseline JIT code for isSymbol#BTGpXV:[0x6
   Source: function isSymbol(obj) {return typeof obj
   Code at [0x374dbee009a0, 0x374dbee00ff0):
          0x374dbee009a0: push %rbp
          0x374dbee009a1: mov %rsp, %rbp
          ...
   [   0] enter
          0x374dbee00a5f: mov $0xa, -0x20(%rbp)
          ...
   [   1] log_shadow_chicken_prologue
          0x374dbee00b5e: mov $0x2, 0x24(%rbp)
          ...
   [   2] get_scope         loc3
          0x374dbee00bc9: mov 0x18(%rbp), %rax
          ...
```

```
    [   4] mov               loc4, loc3
           0x374dbee00bd5: mov -0x20(%rbp), %rax
           0x374dbee00bd9: mov %rax, -0x28(%rbp)
    [   7] create_lexical_environment loc5, loc3, Cel
           0x374dbee00bdd: mov $0x8, 0x24(%rbp)
           ...
    [  12] mov               loc3, loc5
           0x374dbee00c21: mov -0x30(%rbp), %rax
           0x374dbee00c25: mov %rax, -0x20(%rbp)
    [  15] put_to_scope      loc5, obj(@id0), arg1, 2
           0x374dbee00c29: mov 0x30(%rbp), %rax
           ...
    [  22] debug             didEnterCallFrame, 0
           ...
    [  25] debug             willExecuteStatement, 0
           ...
    [  28] get_from_scope    loc7, loc5, obj(@id0), 1
           ...
    [  36] typeof            loc8, loc7
           ...
    [  39] stricteq          loc8, loc8, String (atom
           ...
    [  43] debug             willLeaveCallFrame, 0
           ...
    [  46] ret               loc8
           ...
           0x374dbee00e4d: ret
    (End Of Main Path)
    (S) [  39] stricteq          loc8, loc8, String (
           0x374dbee00e4e: mov $0x28, 0x24(%rbp)
           ...
           0x374dbee00ea0: jmp 0x374dbee00dd8
    (End Of Slow Path)
           0x374dbee00ea5: mov $0x62d000287400, %rsi
           ...
           0x374dbee00fee: jmp *%rsi
Optimized isSymbol#BTGpXV:[0x62d000287400->0x62d0001
```

Note: we will also get dumps for lots of other pieces of code that aren't our function (not shown in my excerpt above). This is because *JSC_dumpDisassembly=true* literally dumps all disassembly. The other dumps comes from JIT thunk code that we compile but are not generated by the baseline JIT. We can ignore those for now.

Looking at the dump for `isSymbol#BTGpXV`, we see sections of code generated for each bytecode in the function. It's a lot of code to inspect manually (324 lines in the dump).

We need to reduce the search area further.

Debugging with Logging ∞

Let's take a moment to regroup and think through the facts that we have so far. In the beginning, we saw a callee save register, `%rbx`, corrupted in `JITCode::execute()`. `JITCode::execute()` is where JSC calls into JS code. It does this by calling a thunk named `vmEntryToJavaScript`. Immediately after returning from `vmEntryToJavaScript`, it did an exception check, and that's where it crashed.

Here's the relevant code:

```
JSValue JITCode::execute(VM* vm, ProtoCallFrame* pro
{
    ...
    JSValue result = JSValue::decode(vmEntryToJavaSc
    return vm->exception() ? jsNull() : result;
}
```

`vmEntryToJavaScript` is implemented in LLInt assembly using the `doVMEntry` macro (see LowLevelInterpreter.asm and LowLevelInterpreter64.asm). The JSC VM enters all LLInt or JIT code via `doVMEntry`, and it will exit either via

the end of `doVMEntry` (for normal returns), or via `_handleUncaughtException` (for exits due to uncaught exceptions).

Here's an idea ... we should probe the value of `%rbx` at these entry and exit points. After all, the value of `%rbx` that we want to preserve comes from `JITCode::execute()`, which is the caller of `vmEntryToJavaScript` / `doVMEntry`.

One way to probe the value of `%rbx` is to set a debugger breakpoint at the points of interest in code, and inspect the register value when the debugger breaks. However, this technique is only good when we already know that the bug will manifest within the first few times that the debugger breaks.

But if we're not sure when the bug will manifest, we should probe the value of `%rbx` by logging it at the points of interest instead. This allows us to get a bird's eye view of when the bug manifests. If necessary, we can go back and apply breakpoints thereafter in a more targeted way. Since we don't know yet which VM entry/exit the bug will manifest on, let's do some logging.

A LLInt Probe and Counting VM Entries ∞

For logging from LLInt code (which is hand written assembly), we'll need a probe mechanism that preserves CPU registers, calls to a C++ function to do our logging, and then restores the CPU registers after the call. This way we can insert the probes in the LLInt code without perturbing its operation.

Here's a snippet of LLInt code (for X86_64 on OSX only) that will do the job:

```
macro probe(func)
    # save all the registers that the LLInt may use.
    push a0, a1
    push a2, a3
```

```
    push t0, t1
    push t2, t3
    push t4, t5

    emit "mov %rbx, %rdi" # pass %rbx as arg0 (i.e.
    move sp, a1 # pass the stack pointer as arg1.
    call func # call the probe function.

    # restore all the registers we saved previously.
    pop t5, t4
    pop t3, t2
    pop t1, t0
    pop a3, a2
    pop a1, a0
    end
```

We add this LLInt probe macro at the top of *LowLevelInterpreter64.asm*. Next, we insert probes at various locations in *LowLevelInterpreter.asm*:

```
    _vmEntryToJavaScript:
    ...
    doVMEntry(makeJavaScriptCall, _jsEntryEnterProbe

    ...
    _vmEntryToNative:
    ...
    doVMEntry(makeHostFunctionCall, _hostEnterProbe,
```

... and *LowLevelInterpreter64.asm*:

```
macro doVMEntry(makeCall, entryProbe, exitProbe)
    functionPrologue()
    pushCalleeSaves()
```

```
    probe(entryProbe)
    ...
    probe(exitProbe)

    popCalleeSaves()
    functionEpilogue()

    ret
end

_handleUncaughtException:
    probe(_uncaughtExceptionEnterProbe)
    loadp Callee[cfr], t3
    ...
    probe(_uncaughtExceptionExitProbe)

    popCalleeSaves()
    functionEpilogue()
    ret
```

... and add the corresponding callback functions for these probes at the bottom of LLIntSlowPaths.cpp:

```
extern int vmEntryCount;

extern "C" void jsEntryEnterProbe(void* rbx, void* r
void jsEntryEnterProbe(void* rbx, void* rsp)
{
    dataLog("ENTRY[", vmEntryCount, "] jsEntry ENTER
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}

extern "C" void jsEntryExitProbe(void* rbx, void* rs
void jsEntryExitProbe(void* rbx, void* rsp)
{
    dataLog("ENTRY[", vmEntryCount, "] jsEntry EXIT:
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}
```

```
extern "C" void hostEnterProbe(void* rbx, void* rsp)
void hostEnterProbe(void* rbx, void* rsp)
{
    dataLog("ENTRY[", vmEntryCount, "] host ENTER: r
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}

extern "C" void hostExitProbe(void* rbx, void* rsp);
void hostExitProbe(void* rbx, void* rsp)
{
    dataLog("ENTRY[", vmEntryCount, "] host EXIT: rb
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}

extern "C" void uncaughtExceptionEnterProbe(void* rb
void uncaughtExceptionEnterProbe(void* rbx, void* rs
{
    dataLog("ENTRY[", vmEntryCount, "] uncaughtExcep
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}

extern "C" void uncaughtExceptionExitProbe(void* rbx
void uncaughtExceptionExitProbe(void* rbx, void* rsp
{
    dataLog("ENTRY[", vmEntryCount, "] uncaughtExcep
        RawPointer(rbx), " rsp=", RawPointer(rsp), "
}
```

`dataLog()` (see DataLog.h and DataLog.cpp) is WebKit's equivalent of `printf()` except that:
1. it prints to `stderr`, or to a file.
2. it doesn't take a formatting string. It just takes a variable list of items to print.
3. it knows how to print any WebKit class that implements a `dump()` method (e.g. `ScopeOffset::dump()`) or has an associated `printInternal()` function (e.g. this one for `CodeType`).

We should also add a few lines to `JITCode::execute()` to count how deep we have entered into the VM:

```
int vmEntryCount = 0;
...
JSValue JITCode::execute(VM* vm, ProtoCallFrame* pro
{
    ...
    vmEntryCount++;
    JSValue result = JSValue::decode(vmEntryToJavaSc
    vmEntryCount--;
    return vm->exception() ? jsNull() : result;
}
```

With this, we can now see the values of `%rbx` at each level of VM entry and exit. So, we rebuild WebKit and re-run the test. Here is an excerpt of the resultant logs:

```
...
ENTRY[1] jsEntry ENTER: rbx=0x7fff5fbfa0a0 rsp=0x7ff
...
ENTRY[2] jsEntry ENTER: rbx=0x7fff5fbf5a00 rsp=0x7ff
ENTRY[2] host ENTER: rbx=0x7fff5fbf2cc0 rsp=0x7fff5f
ENTRY[2] host EXIT: rbx=0x7fff5fbf2cc0 rsp=0x7fff5fb
ENTRY[2] jsEntry EXIT: rbx=0x7fff5fbf5a00 rsp=0x7fff
...
ENTRY[2] host ENTER: rbx=0x7fff5fbf2ca0 rsp=0x7fff5f
ENTRY[2] host EXIT: rbx=0x7fff5fbf2ca0 rsp=0x7fff5fb

Optimized isSymbol#BTGpXV:[0x62d000214100->0x62d0001

...
ENTRY[2] host ENTER: rbx=0x7fff5fbf2ca0 rsp=0x7fff5f
ENTRY[2] host EXIT: rbx=0x7fff5fbf2ca0 rsp=0x7fff5fb
...
ENTRY[2] host ENTER: rbx=0x7fff5fbf4860 rsp=0x7fff5f
ENTRY[2] host EXIT: rbx=0x7fff5fbf4860 rsp=0x7fff5fb
...
ENTRY[2] jsEntry ENTER: rbx=0x7fff5fbf6920 rsp=0x7ff
ENTRY[2] jsEntry EXIT: rbx=0x7fff5fbf6920 rsp=0x7fff
ENTRY[2] jsEntry ENTER: rbx=0x7fff5fbf5380 rsp=0x7ff
```

```
...
ENTRY[2] jsEntry EXIT: rbx=0x7fff5fbf5380 rsp=0x7fff

ENTRY[1] uncaughtException ENTER: rbx=0x7fff5fbfa0a0
ENTRY[1] uncaughtException EXIT: rbx=0x7fff5fbf6280
*** CRASHED here ***
```

Looking at the logs from the crash point going backwards, we find the following:
1. The last exit point before we crashed in `JITCode::execute()` is from the `_handleUncaughtException` handler for `ENTRY[1]`.
2. The value of `%rbx` when we entered `_handleUncaughtException` is `0x7fff5fbfa0a0`, which matches the `%rbx` at the last time we entered the VM at `ENTRY[1]`.
3. However, the value of `%rbx` when we exited `_handleUncaughtException` is a different (and incorrect) value, `0x7fff5fbf6280`.

How did the value of `%rbx` get corrupted between the entry to and exit from `_handleUncaughtException`?

Looking in `_handleUncaughtException` (in *LowLevelInterpreter64.asm*), we see that there's a call to a `restoreCalleeSavesFromVMCalleeSavesBuffer()` macro.
`restoreCalleeSavesFromVMCalleeSavesBuffer()` (in *LowLevelInterpreter.asm*) basically copies values from the `VM::calleeSaveRegistersBuffer` buffer to the callee save register. That explains why `%rbx`'s value changed while in `_handleUncaughtException`. What remains unanswered is how that bad value got into the `VM::calleeSaveRegistersBuffer` buffer.

Who is Modifying VM::calleeSaveRegistersBuffer? ∞

A quick search for `calleeSaveRegistersBuffer` in the source code later, and we see that

`VM::calleeSaveRegistersBuffer` is written to in only a few places:

1. The `copyCalleeSavesToVMCalleeSavesBuffer()` macro in *LowLevelInterpreter.asm*.
2. `UnwindFunctor::copyCalleeSavesToVMCalleeSavesBuffer()` in *Interpreter.cpp*.
3. `compileStub()` in *FTLOSRExitCompiler.cpp* emits code to write to the buffer.
4. `AssemblyHelpers::copyCalleeSavesToVMCalleeSavesBuffer()` and `AssemblyHelpers::copyCalleeSavesFromFrameOrRegisterToVMCal` in AssemblyHelpers.h emits code to write to the buffer.

Logging the Modifications to VM::calleeSaveRegistersBuffer ∞

The next thing to do is to log the values that are written to `VM::calleeSaveRegistersBuffer`. First of all, we can rule out the one in *FTLOSRExitCompiler.cpp* because we've disabled the FTL for our test.

For the `copyCalleeSavesToVMCalleeSavesBuffer()` macro in *LowLevelInterpreter.asm*, we can add a LLInt probe like so:

```
macro copyCalleeSavesToVMCalleeSavesBuffer(vm, temp)
    if ARM64 or X86_64 or X86_64_WIN
        leap VM::calleeSaveRegistersBuffer[vm], temp
        if ARM64
            probe(_llintSavingRBXProbe)
            storep csr0, [temp]
            storep csr1, 8[temp]
            ...
```

... and the following at the bottom of *LLIntSlowPath.cpp*:

```
extern "C" void llintSavingRBXProbe(void* rbx, void*
void llintSavingRBXProbe(void* rbx, void* rsp)
{
    dataLog("ENTRY[", vmEntryCount, "] LLInt set the
        RawPointer(rbx), "\n");
}
```

`UnwindFunctor::copyCalleeSavesToVMCalleeSavesBuffer()` is C++ code. So we can just adds some logging like so:

```
extern int vmEntryCount;
...
void copyCalleeSavesToVMCalleeSavesBuffer(StackVisit
{
    ...
    unsigned registerCount = currentCalleeSaves->siz
    for (unsigned i = 0; i < registerCount; i++) {
        RegisterAtOffset currentEntry = currentCalle
        ...
        vm.calleeSaveRegistersBuffer[vmCalleeSavesEn
        // Begin logging.
        auto reg = currentEntry.reg();
        if (reg.isGPR() && reg.gpr() == X86Registers
            void* rbx = reinterpret_cast<void*>(
                vm.calleeSaveRegistersBuffer[vmCalle
            dataLog("ENTRY[", vmEntryCount, "] Unwin
                RawPointer(rbx), "\n");
        }
        // End logging.
    }
    ...
}
```

Printing from JIT code ∞

For `AssemblyHelpers::copyCalleeSavesToVMCalleeSavesBuffer()` and `AssemblyHelpers::copyCalleeSavesFromFrameOrRegisterToVMCallee` we're dealing with JIT generated code.

JSC has its own macro assembler that the JITs use to emit machine instructions for the compiled JS functions. The macro assembler provides emitter functions for generating a machine instruction or a pseudo-instruction made up of a sequence of machine instructions.

The `print` emitter is one such pseudo-instruction emitter for debugging use only:

```
template<typename... Arguments>
void print(Arguments... args);
```

`print` takes a comma separated list of arguments that it will concatenate and print to `stderr`. In addition to printing the usual data types (like `const char*` strings, and `int`s), it also knows how to print the runtime values of CPU registers, memory locations, or dump all registers. See MacroAssemblerPrinter.h for more details.

To use `print`, set `ENABLE_MASM_PROBE` (in Platform.h) to a non-zero value, and include `MacroAssemblerPrinter.h` in your file. Here's how we use it in *AssemblyHelpers.h*:

```
#include "MacroAssemblerPrinter.h"
...
    void copyCalleeSavesToVMCalleeSavesBuffer(const
    {
#if NUMBER_OF_CALLEE_SAVES_REGISTERS > 0
        ...
        for (unsigned i = 0; i < registerCount; i++)
            RegisterAtOffset entry = allCalleeSaves-
            ...
```

```
            if (entry.reg().isGPR()) {
                // Begin logging.
                auto entryGPR = entry.reg().gpr();
                if (entryGPR == X86Registers::ebx) {
                    print("ENTRY[", MemWord(Absolute
                        "] AH::copyCalleeSavesToVMCa
                        entryGPR, "\n");
                }
                // End logging.
                storePtr(entry.reg().gpr(), Address(
            } else
                ...

    void copyCalleeSavesFromFrameOrRegisterToVMCalle
    {
#if NUMBER_OF_CALLEE_SAVES_REGISTERS > 0
        ...
        RegisterAtOffsetList* currentCalleeSaves = c
        ...
        for (unsigned i = 0; i < registerCount; i++)
            RegisterAtOffset vmEntry = allCalleeSave
            ...
            if (vmEntry.reg().isGPR()) {
                GPRReg regToStore;
                if (currentFrameEntry) {
                    // Load calleeSave from stack in
                    regToStore = temp2;
                    loadPtr(Address(framePointerRegi
                } else
                    // Just store callee save direct
                    regToStore = vmEntry.reg().gpr()

                // Begin logging.
                if (vmEntry.reg().gpr() == X86Regist
                    print("ENTRY[", MemWord(Absolute
                        "] AH::copyCalleeSavesFromFr
                        regToStore, "\n");
                }
                // End logging.
                storePtr(regToStore, Address(temp1,
```

```
            } else {
                ...
```

In the above, we're using `print` to log a string that looks like this:

```
ENTRY[0x10cddf280:<0x00000002 2>] AH::copyCalleeSave
```

Note that in `copyCalleeSavesFromFrameOrRegisterToVMCalleeSavesBuffer()`, the `MemWord(AbsoluteAddress(&vmEntryCount))` argument passed to `print` is printed as `0x10cddf280:<0x00000002 2>` instead. `0x10cddf280` is the address `&vmEntryCount`, and `<0x00000002 2>` is the runtime `int` value found at that address at the time that the code generated for `print` was executed.

Similarly, the `regToStore` variable was printed as `ebx:<0x7fff57a75bc0 140734663973824>`. Though `regToStore` is a variable at JIT compilation time, `print` captures its value as the id of the register to print. In this example, that would be the `%rbx` register. It is printed as `ebx` because `ebx` is the id that the macro assembler uses to represent both the `%ebx` register on 32-bit x86 and the `%rbx` register on 64-bit X86_64 ports of WebKit (see the `RegisterID` enum list in X86Assembler.h). `0x7fff57a75bc0` is the value in the `%rbx` register at the time the code generated for `print` was executed.

Following the Trail ∞

With all this logging code added, the interesting parts of the logging output now looks like this:

```
...
ENTRY[0x10a92d280:<0x00000002 2>] AH::copyCalleeSave
```

```
...
ENTRY[0x10a92d280:<0x00000002 2>] AH::copyCalleeSave
...
ENTRY[0x10a92d280:<0x00000002 2>] AH::copyCalleeSave
...
ENTRY[1] uncaughtException ENTER: rbx=0x7fff59f25100
ENTRY[1] uncaughtException EXIT:  rbx=0x7fff59f212e0
ASAN:SIGSEGV
```

It appears that `copyCalleeSavesFromFrameOrRegisterToVMCalleeSavesBuffer()` is the only one who is writing to `VM::calleeSaveRegistersBuffer`, and overwriting the saved value for `%rbx` with `0x7fff59f212e0` (which matches the corrupted value that we restored to `%rbx` in `_handleUncaughtException` later just before the crash point). The logging also shows that the corruption happened more than once (in fact, a lot more than the 3 times I chose to include in the above excerpt of the logs).

A quick search for `copyCalleeSavesFromFrameOrRegisterToVMCalleeSavesBuffer()` reveals that it is only called from 2 functions: `JIT::emitEnterOptimizationCheck()` and `JIT::emitSlow_op_loop_hint()`. Both of which generate code for the baseline JIT.

Adding some `print`s to these, we find that `JIT::emitEnterOptimizationCheck()` is the source of our corruption.

From here, with a bit more logging and code searching, we will uncover the rest of the details needed to piece together the whole story of how the corruption happened. So, here's the story of how the crash came to be:

1. The test code enters the VM 2 levels deep.
2. During the 1st level entry, an uncaught JS exception is thrown. This leads to `%rbx` being saved in

`VM::calleeSaveRegistersBuffer`.

3. The 2nd level entry is for Web Inspector, which is inspecting the thrown exception. During the 2nd level entry, Web Inspector calls `isSymbol#BTGpXV` a lot thereby making it a hot function.
4. Because `isSymbol#BTGpXV` is hot, the VM tries to DFG compile it (via `JIT::emitEnterOptimizationCheck()`), but fails to. This is why the DFG has to be enabled, though no functions are DFG compiled.
5. The code emitted by `JIT::emitEnterOptimizationCheck()` first saves callee save registers to `VM::calleeSaveRegistersBuffer` (via `copyCalleeSavesFromFrameOrRegisterToVMCalleeSavesBuffer()`. Hence, the value `%rbx` in the buffer is overwritten here.
6. When execution returns to `_handleUncaughtException` on the 1st level entry, it copies the now corrupted value in `VM::calleeSaveRegistersBuffer` to `%rbx`, and we later get a crash when we try to use `%rbx`'s value.

In other words, the bug is that the VM needs a `calleeSaveRegistersBuffer` buffer for each level of entry into the VM, but it only provided one that is used by all levels. If you're interested, you can check out the fix here.

Summary ∞

We have seen how we can use the JSC options, in combination with logging to diagnose the root cause of a bug. We saw that the `print` pseudo-instruction is available for doing logging from JIT generated code. These tools enables us to isolate VM bugs to specific tiers of execution engines, to do dumps that let us peek into the artifacts that the JITs generate, and also to easily add logging to do fine grain inspection of registers and values used by JIT compiled code at runtime.

If you're working on a port of JavaScriptCore or are simply interested in exploring how JSC works under the covers, I

hope you'll find these tools helpful in your adventure with JavaScriptCore.

As always, if you encounter any bugs in the tools or have requests for enhancements, please file bugs at bugs.webkit.org. You're also welcome to join the WebKit community and contribute fixes for the bugs or implement the enhancements yourself.

Cheers. ■

Next

Memory Debugging with Web Inspector

Learn more

Previously

Release Notes for Safari Technology Preview 5

Learn more