Emit unwinding information for improved debugging of generated code on Win64

Author: paolosev@microsoft.com
Status: Draft
Tracking bug for current state: v8:3598
Created: 2019-1-10 / **Last Updated:** 2019-2-15

LGTMs needed

Name	Write (not) LGTM in this row

Introduction

Debugging TurboFan-generated code in Windows/x64 is difficult because the V8 x64 backend does not emit unwind information on Windows and also because it doesn't emit ABI-compliant stack frames. This breaks the Windows OS stack unwinder, causing several problems:

- Debuggers cannot unwind the stack past V8 dynamically generated frames, so in WinDbg the '*k*' command cannot list full call stacks through V8 frames:

```
(windbg)> k
 # Child-SP          RetAddr           Call Site
00 000000f4`96bfd1f0 0000576e`e0495db1 v8!Builtins_MathRandom+0x62
01 000000f4`96bfd1f8 00000675`843c1171 0x0000576e`e0495db1
02 000000f4`96bfd200 00000000`00000018 0x00000675`843c1171
03 000000f4`96bfd208 000000f4`96bfd240 0x18
04 000000f4`96bfd210 000000f4`96bfd218 0x000000f4`96bfd240
05 000000f4`96bfd218 00000675`8439fd09 0x000000f4`96bfd218
06 000000f4`96bfd220 00000000`00000000 0x00000675`8439fd09
```

- Performance tools like Windows Performance Analyzer, which also rely on the OS stack unwinder to gather CPU usage samples, cannot collect call stack data through V8 frames.

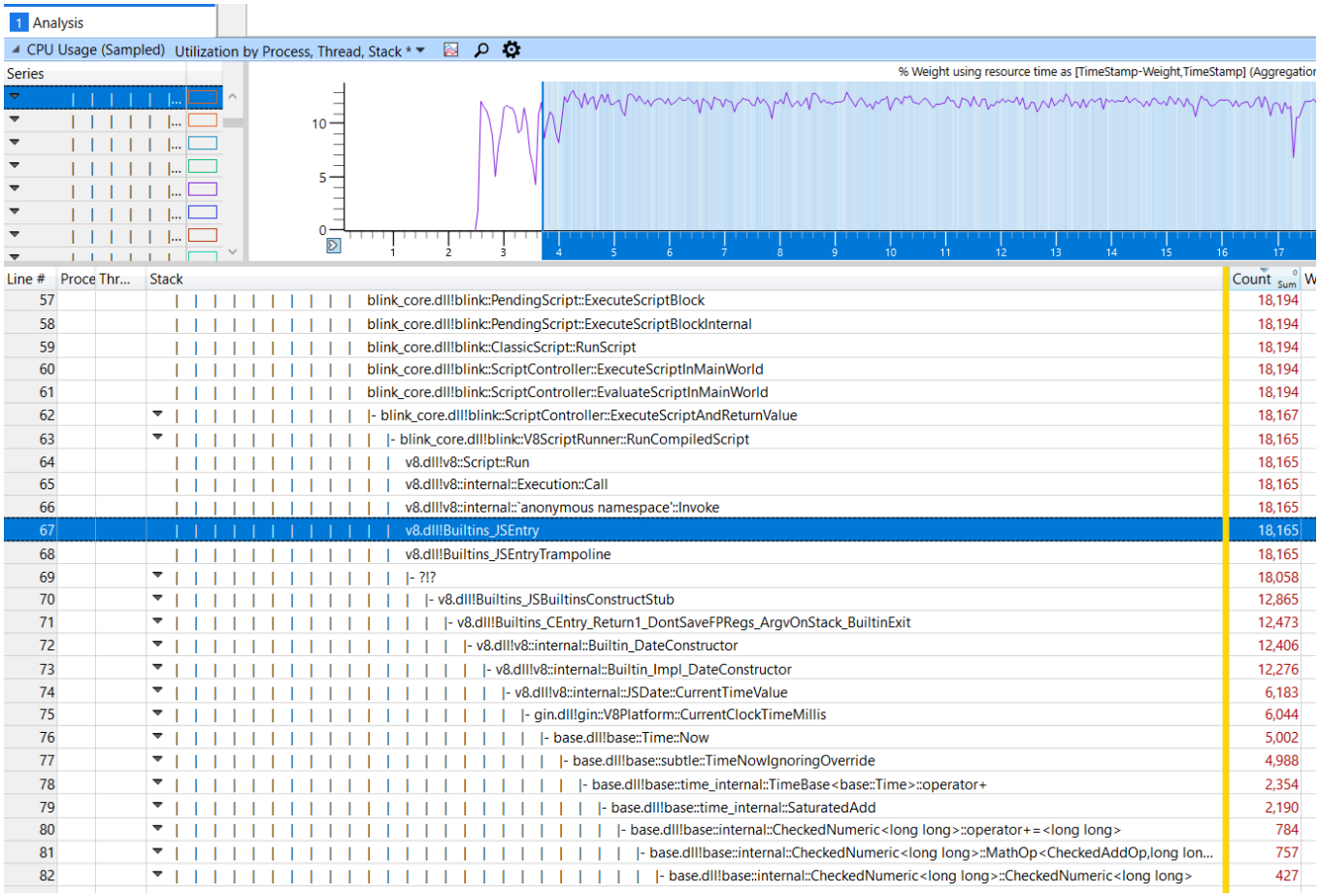
- Win64 SEH (structured exception handling) aborts in the presence of an exception and does not call the unhandled exception handler. This issue has been fixed by installing in Crashpad and Breakpad a custom unwind callback for the entire code range of generated code, which however is not effective for embedded code.

This document describes how unwind information could be emitted to enable the Windows stack walker to work correctly with V8-generated code, both for code dynamically jitted at runtime and for CSA/Torque builtins code that is precompiled and embedded in the v8 binaries (and for which an effort is ongoing to improve the debugging experience).

This is how V8 stack frames appear in a call stack, with the correct unwind data:

```
(windbg)> k
 # Child-SP          RetAddr           Call Site
00 000000ce`719f6c50 00007ff9`6ef11d9e v8!v8::internal::MathRandom::RefillCache+0x31
01 000000ce`719f6d50 00007ff9`6ed743ce v8!Builtins_MathRandom+0x13e
[E:/src/out/x64DbgCB\..\..\v8\src\math-random.cc @ 36]
02 000000ce`719f6db8 00007ff9`6ed743ce v8!Builtins_InterpreterEntryTrampoline+0x34e
03 000000ce`719f6e18 00007ff9`6ed69fc4 v8!Builtins_InterpreterEntryTrampoline+0x34e
04 000000ce`719f6e80 00007ff9`6ed69b62 v8!Builtins_JSEntryTrampoline+0x64
05 000000ce`719f6ea8 00007ff9`6e03c1d8 v8!Builtins_JSEntry+0xe2
06 000000ce`719f6fc0 00007ff9`6e0399b1 v8!v8::internal::GeneratedCode<v8:...
07 000000ce`719f7060 00007ff9`6e038cf4 v8!v8::internal::`anonymous namespace'::Invoke+0xa51
08 000000ce`719f73f0 00007ff9`6e038a8c v8!v8::internal::`anonymous
namespace'::CallInternal+0x244
09 000000ce`719f7530 00007ff9`6d6ba909 v8!v8::internal::Execution::Call+0xbc
0a 000000ce`719f75e0 00007ff9`698bca9b v8!v8::Script::Run+0x3b9
0b 000000ce`719f7840 00007ff9`697e3822
blink_core!blink::V8ScriptRunner::RunCompiledScript+0x74b
[...]
4e 000000ce`719ffaf0 00007ff9`f7aaa251 KERNEL32!BaseThreadInitThunk+0x14
4f 000000ce`719ffb20 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

And this is an example of a WPA (windows performance analyzer) trace of Chromium, when we emit unwind data for V8:



We see that now the OS can fully unwind the call stack also across V8 frames. Interestingly, the call stack can also display function names for all embedded built-ins functions, because V8 already decorates the inline assembly code for built-ins with the necessary annotations to generate the right DWARF/PDB symbol data. However, the jitted frames in the stack won't have any named symbols associated with them (they appear as *"?!?"* in WPA), and generating such symbols is outside the scope of this document.

Stack walking on Win64

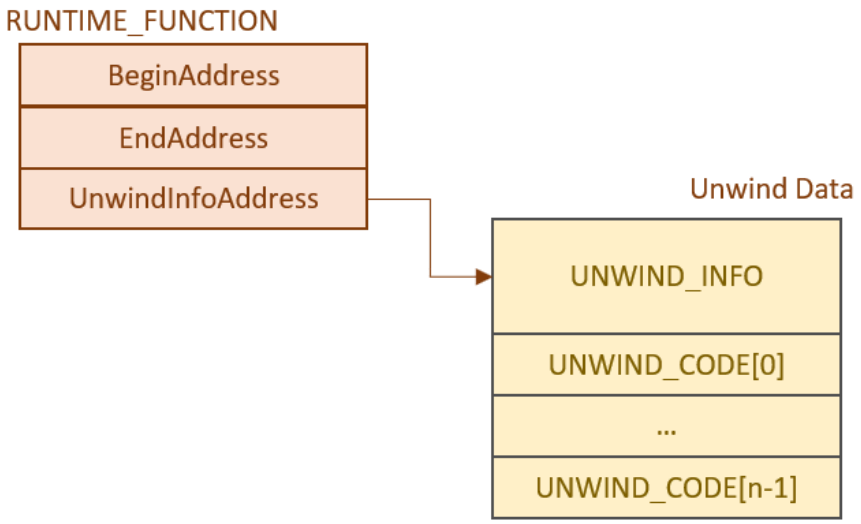
On the x64 architecture, Windows stack walking requires that the correct unwind data be generated and registered for exception handling and debugger support.
Normally this work is done by the compiler and linker, which store unwind data into the .pdata and .xdata sections of the PE executable. But for dynamically generated code we can register unwind data at runtime by calling `RtlAddGrowableFunctionTable` for each function jitted by V8, passing a `RUNTIME_FUNCTION` (or PDATA) object, so declared:

```
struct RUNTIME_FUNCTION {
  DWORD BeginAddress;
```

```
  DWORD EndAddress;
  DWORD UnwindInfoAddress;
};
```

Here, `BeginAddress` and `EndAddress` define the address range of a generated function, and `UnwindInfoAddress` points to the unwind data (or XDATA) for that function. Note that `RUNTIME_FUNCTION` contains RVAs, 32 bits wide virtual addresses relative to the load address of the module that contains the code. For dynamically generated code, the load address is the location of the first instruction in the function.

XDATA is encoded as an `UNWIND_INFO` struct followed by a sequence of `UNWIND_CODE`s, which specify how, given an instruction pointer, the instructions in the prologue of the current function should be reversed in order to walk from the current to the previous stack frame.



Furthermore, the Windows unwinder expects that the code of a function respects the Win64 calling conventions. A function prologue should be limited to these operations:

- saving parameter registers to their shadow space on the stack
- pushing nonvolatile registers
- moving nonvolatile registers to locations on the stack
- decrementing `RSP` by a constant
- establishing a frame pointer in a nonvolatile register other than `RSP`.\

The epilogue should do the inverse operations of the prologue.
The stack pointer `RSP` usually does not change between the prologue and the epilogue of a function, but a frame pointer can be used when it is necessary to allocate additional memory on the stack in the function (like with *_alloca()*).

Unwinding V8 non-ABI-standard frames

Currently the stack frames for code generated by V8 do not follow the Win64 calling conventions and have usually the following prologue and epilogue:

```
  0x55                    push rbp                // PROLOGUE
  0x48 0x89, 0xe5         movq rbp,rsp
  …                     [ push registers ]
                          subq rsp, n

                          [...]

                          movq rsp,rbp            // EPILOGUE
                          pop rbp
                          ret N
```

But it turns out that, even if the Win64 ABI is not respected, it is still possible to generate unwind info that allows the Windows stack walker to correctly unwind those stack frames. The following XDATA works for every V8-generated function:

```
typedef union _UNWIND_CODE { // from Windows SDK ehdata.h
  struct {
    unsigned char CodeOffset;
    unsigned char UnwindOp : 4;
    unsigned char OpInfo : 4;
  };
  unsigned short FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;

typedef struct _UNWIND_INFO { // from Windows SDK ehdata.h
  unsigned char Version : 3;
  unsigned char Flags : 5;
  unsigned char SizeOfProlog;
  unsigned char CountOfCodes;
  unsigned char FrameRegister : 4;
  unsigned char FrameOffset : 4;
  /*  UNWIND_CODE UnwindCode[CountOfCodes];
   *  union {
   *      OPTIONAL unsigned long ExceptionHandler;
   *      OPTIONAL unsigned long FunctionEntry;
   *  };
   *  OPTIONAL unsigned long ExceptionData[];
   */
} UNWIND_INFO, *PUNWIND_INFO;

struct V8_BASE_EXPORT V8UnwindData {
 UNWIND_INFO unwind_info;
 UNWIND_CODE unwind_codes[2];

 V8UnwindData() {
   unwind_info.Version = 1;
```

```
    unwind_info.Flags = 0;
    unwind_info.SizeOfProlog = 4;    // length of 'push rbp - movq rbp, rsp'
    unwind_info.CountOfCodes = 2;
    unwind_info.FrameRegister = 5;   // rbp
    unwind_info.FrameOffset = 0;

    unwind_codes[0].CodeOffset = 4; // IP offset after 'movq rbp, rsp'
    unwind_codes[0].UnwindOp = UWOP_SET_FPREG;
    unwind_codes[0].OpInfo = 0;

    unwind_codes[1].CodeOffset = 1; // IP offset after 'push rbp'
    unwind_codes[1].UnwindOp = UWOP_PUSH_NONVOL;
    unwind_codes[1].OpInfo = 5;      // rbp
  }
};

struct CodeRangeUnwindingRecord {
  RUNTIME_FUNCTION runtime_function;
  V8UnwindData unwind_info;
  unsigned long exception_handler;
  unsigned long exception_thunk[12];
  void* dynamic_table;
};
```

How does this work? Given a current thread context and instruction pointer, [Win64 stack unwinder]:

1. First checks whether the current instruction is in a function epilogue. To do this it looks for various sequence of instruction, and it recognizes as a valid epilogue the end of a V8 epilogue:

   ```
   pop rbp
   ret N
   ```

   Therefore the frame can be correctly unwound when the instruction pointer is in one of the last two instructions.

2. If the current instruction is not in the epilogue, the OS tries to “undo” the instructions that were executed in the function prologue. Information about the work done by these instructions is encoded in the array of `UNWIND_CODE`s which are normally generated by the compiler and linker. Normally a compiler should emit these XDATA for a V8 prologue:

   ```
   push RBP           UWOP_PUSH_NONVOL (RBP)
   movq RBP, RSP       UWOP_IGNORE
   ```

```
push RSI           UWOP_PUSH_NONVOL (RSI)
push RDI           UWOP_PUSH_NONVOL (RDI)
sub RSP, n          UWOP_ALLOC_SMALL (n)
```

But we know that this is not standard, so it would not work.
However, if we generate these two `UNWIND_CODE`s for the first two instructions in the prologue:

```
UWOP_SET_FPREG         (reverted as: RSP = RBP)
UWOP_PUSH_NONVOL(RBP)  (reverted as: RBP = *RSP; RSP += 8)
```

Then the OS can unwind *almost* correctly the prologue: it can retrieve the correct values of `RSP`, `RBP`, `RIP` for the caller frame, for each instruction in and after the prologue. What it cannot do is to recover the value of non-volatile registers, but this should not generally be a problem if our goal is only to walk the stack (unless a non-volatile register happens to be used as frame pointer in one of the non-V8 generated frames in the call stack).

Interestingly, the unwind info is the same for each function that starts with `push RBP`, `movq RBP, RSP` and does not depend on other details of each particular function, like the function length.

Registering unwinding information

There are three kinds of dynamically generated code we need to consider:

1. Embedded builtins, compiled at snapshot build-time
2. Functions dynamically generated at run-time
3. WASM code compiled at run-time

Jitted functions

For jitted functions, the `src/eh-frame` classes, and the x64 `UnwindingInfoWriter` classes already know how to gather eh_frame information, and the `Code` class already provides an (optional) space for storing unwinding info. We could reuse this area to store both PDATA (a `RUNTIME_FUNCTION` object) and the XDATA (a struct equivalent to `V8UnwindData` above), writing a Win64 specialization of the `UnwindingInfoWriter`.

But there is a much simpler possible implementation. Chromium already has code that registers PDATA/XDATA not for stack unwinding but for exception handling, in order to solve the issue with Win64 SEH mentioned above. Class `gin::IsolateHolder` allows embedders to register callbacks to be notified when a memory code-range is allocated or deallocated and CrashPad (*src\components\crash\content\app\crashpad_win.cc*) registers a callback that calls

`RtlAddFunctionTable` to register a single `RUNTIME_FUNCTION` object that spans the whole code-range and whose `UNWIND_INFO` only contains exception-handling data. The function `Isolate::GetCodeRange()` was added for this reason to the v8.h and the `RUNTIME_FUNCTION` and `UNWIND_INFO` objects are allocated in the first page of the code range, which for this reason is reserved and marked as writeable.
Here we can leverage the fact that the stack unwinding data is the same for every function jitted in this code range, and so we can reuse this single PDATA/XDATA entry even for stack unwinding and not only for exception handling. This is the equivalent of telling the Windows kernel that there is a single huge jitted function and that the stack unwinding for any instruction in this address space is determined by the same unwinding codes. Note that `RtlAddGrowableFunctionTable` should be used in place of `RtlAddFunctionTable` to register PDATA that also contains unwind info; but `RtlAddGrowableFunctionTable` is only available in Windows 8 and above, and so it is not possible to make stack unwinding work on Windows 7.

It could be possible to make these changes directly in CrashPad, modifying the `UNWIND_INFO` class declared in *crashpad_win.cc*, to also pass the `UNWIND_CODE`s described early. But that would mean that potentially every V8 embedder should be modified to enable stack unwinding on Win64, and, what's worse, to do so embedders would depend on the knowledge of internal implementation details of V8, such as the format of stack frames.

A better solution is to add the logic to emit unwind info inside V8. We should register the PDATA after a code range has been allocated for an Isolate (in `Isolate::Init`) and unregister it when an Isolate is being deleted. And since multiple PDATA entries should not be registered for the same address range, we need to make a small change to class `gin::IsolateHolder` to check whether unwind info is already being registered by V8. For this reason we add a new function to the V8 API (`v8.h`):

```
bool Isolate::ShouldInstallFunctionTableCallbacksForStackUnwinding()
```

which should be used by any embedder that calls `Isolate::GetCodeRange()` to generate unwind info for an Isolate code range.

But what happens to the Crashpad exception handler then? To be sure that the Crashpad exception handler will also be called when an exception happens in some V8 code with a stack that for some reason cannot be completely unwound, we can add another (Windows-specific?) function to the V8 API:

```
typedef int (*UnhandledExceptionCallback)(
    _EXCEPTION_POINTERS* exception_pointers);
void SetUnhandledExceptionCallback(
      UnhandledExceptionCallback unhandled_exception_callback);
```

that embedders can call to pass an exception handler for all V8-generated (or embedded) code.

We rely on the fact that an `UNWIND_INFO`, after the sequence of `UNWIND_CODE`s, can also optionally specify the RVA of an exception handler that will be called every time an exception happens in that code region, as described in the struct `CodeRangeUnwindingRecord` above.

The only disadvantage of registering a single `RUNTIME_FUNCTION` entry to cover all the functions inside a whole isolate code-range is that it is a bit of a cheat and stack walking becomes slightly imprecise. More precisely, we have a small problem when a stack walk happens when the instruction pointer is exactly at the beginning of a prologue. Since `UNWIND_CODE`s are associated to a `CodeOffset` calculated from the beginning of the function, and the logic in OS stack walker ignores `UNWIND_CODE`s for prologue instructions that have not been executed, if we have a V8 prologue like:

```
Offset  Instruction          Unwind code              Reverted as:
0x00    push rbp             UWOP_SET_FPREG           RSP = RBP
0x01    mov rbp, esp         UWOP_PUSH_NONVOL(RBP)    RBP = *RSP; RSP += 8
0x04   {rest of the function}
```

and `RIP` points to one the first two instructions, what happens while unwinding is that these two `UNWIND_CODE`s will not be ignored, and since `RBP` still points to the previous frame, then the previous frame gets skipped in the stack walk.

There can be also problems *with off-heap trampolines*, when calling an embedded builtin: if the caller frame doesn't use `RBP` as frame pointer and the instruction pointer happens to be in one of the two instructions of an off-heap trampoline (`mov r10, target - jmp r10`), then the stack cannot be unwound.

There is no problem, instead, when stack walking happens in a function epilogue because epilogue unwinding does not use the registered `UNWIND_CODE`s.

WASM

In Chromium, class `gin::IsolateHolder` calls the `RegisterNonABICompliantCodeRange` callback only for the Isolate code range for jitted code (that is returned by `v8::Isolate::GetCodeRange()`). This is not where WASM code gets compiled; for every WASM module, a 1GB code range gets allocated in `WasmCodeManager::NewNativeModule()`.

To be able to stack-walk wasm-compiled frames, we need to register PDATA/XDATA also for WASM code ranges. Analogously to the case of Isolates, the first page of a WASM executable space on Win64 needs to be reserved and temporarily marked as writeable, to contain the `RUNTIME_FUNCTION` and `UNWIND_INFO` objects we use to register this code range. Then, analogously to what we do in class `Isolate::Init`, we can modify `WasmCodeManager::NewNativeModule` to register the PDATA for this code range after a

new WASM module has been created, and `WasmCodeManager::FreeNativeModule` to unregister it while the module is being deleted.

Embedded builtins

Stack unwinding is more complicated for builtin code. Unfortunately, not all builtins start with the `push RBP`, `movq RBP, RSP` sequence; this is currently true for 641 builtins, and we can assume that this will be true only for the TFJ builtins (with JavaScript linkage), but it is generally not true for other kind of builtins, especially the ASM builtins directly written in platform-dependent assembly.

We cannot make many assumption on the code generated for builtins. There are builtins, like `CallVarargs`, which don't even allocate a stack frame, and just end with a jump.
In many builtins (like `ArgumentsAdaptorTrampoline` below) there is a non-standard prologue, with code that checks the validity of status and arguments, dispatches the execution or calls `Abort()` in case of errors. There are then usually one or more `push RBP; movq RBP, RSP` sequences that create a stack frame before calling other functions. Finally there is a `movq rsp,rbp; pop rbp` sequence to close the frame, followed by a non-standard epilogue, with code that ends up returning to the caller or jumping to some other function:.
There is even at least one builtin, `StringEqual`, which generates an RBP frame but where the '`movq rsp,rbp; pop rbp`' sequence precedes the '`push RBP; movq RBP, RSP`' in the code.

All this code is clearly too "un-standard" for the Windows stack unwinder. In other words, there is no way to generate a set of `UNWIND_CODE`s that faithfully track the behaviour of these functions. For this reason, even though the Clang GAS assembler supports the generation of pdata/xdata for stack unwinding through .seh directives, this cannot be a viable option..

However, it turns out that the simple `V8UnwindData` XDATA described before still works in almost all cases if we follow this rules:

- Given the code of a builtin function that contains **n** sequences '`push RBP; movq RBP, RSP`', generate (n+1) PDATA entries.
  - The first entry covers the instructions from the beginning of the function, to the first '`push RBP; movq...`' excluded.
  - The second entry covers the instructions from the first '`push RBP`', to the next '`push RBP`', if present.
  - And so on, with a final entry that covers code from the last '`push RBP`' to the end of the function.
- All these PDATA entries refer to the same `V8UnwindData` described early.
- For functions that don't have any '`push RBP`' generates a single PDATA entry that covers the whole function.

```
kind = BUILTIN
name = ArgumentsAdaptorTrampoline
...
Instructions (size = 295)
000001A3916B3960     0  4881fbffff0000 cmpq rbx,0xffff
000001A3916B3967     7  0f84e7000000   jz 000001A3916B3A54  (ArgumentsAdaptorTrampoline)
000001A3916B396D     d  483bc3         cmpq rax,rbx
000001A3916B3970    10  0f8c4b000000   jl 000001A3916B39C1  (ArgumentsAdaptorTrampoline)
000001A3916B3976    16  55             push rbp
000001A3916B3977    17  4889e5         movq rbp,rsp
...
000001A3916B39C1    61  55             push rbp
000001A3916B39C2    62  4889e5         movq rbp,rsp
...
000001A3916B3A44    e4  488be5         movq rsp,rbp
000001A3916B3A47    e7  5d             pop rbp
...
000001A3916B3A53    f3  c3             retl
000001A3916B3A54    f4  488b4f2f       movq rcx,[rdi+0x2f]
...
000001A3916B3A6E   10e  498b8ccd602f0000 movq rcx,[r13+rcx*8+0x2f60]
000001A3916B3A76   116  ffe1           jmp rcx
…
000001A3916B3A86   126  cc             int3l
```

For example, for the builtin `ArgumentsAdaptorTrampoline` above we would emit three PDATA entries, all pointing to the same `UNWIND_INFO`:

1. `RUNTIME_FUNCTION (offset 0x00 to 0x15)`
2. `RUNTIME_FUNCTION (offset 0x16 to 0x60)`
3. `RUNTIME_FUNCTION (offset 0x61 to 0x126)`

We need to modify the x64 `Assembler` class (*src/x64/assembler-x64.h*) to keep track of the generation of '`push RBP; movq RBP, RSP`' when the assembler is used for the code generation of builtins. We need to pass this information to the `EmbeddedFileWriter` that writes the *embedded.cc* file that will also contain directives for the .pdata/.xdata sections, as we'll see later.

How does this work?

- When we are inside a '`push rbp … pop rbp`' region the XDATA states that the current frame can be unwound reverting the `movq RBP, RSP` and then `push RBP` instructions, and this works because `RBP` actually points to the location in the stack that contains the previous `RBP` frame pointer, allowing the OS unwinder to find the caller frame.
- When we are outside a '`push rbp … pop rbp`' region, the OS unwinder will still try to find the previous frame pointer getting `RSP` from `RBP`. In some cases this might fail, because the register `RBP` is being used for some other purposes. But this is very likely to happen only when the builtin function is at the top of the call stack, not when it is in

the middle, because builtins almost always seem to create an RBP stack frame before calling other functions.

Early tests indicate that this solution works surprisingly well; WPR traces of Chromium show that the stack cannot be completely unwound only for 0.1-0.2% of the samples that contain V8 functions (they appear in WPA as stack frames that start with an unknown “?!?” V8 frame as topmost function).
It should be noted finally that the unwinding a function with the “wrong” unwind codes is not a dangerous operation: the Windows OS unwinder can also run in kernel mode and it’s very robust having to assume that the status of each registry can be corrupted or invalid at any time.

Unwind data generation

- Usually the linker generates right PDATA for all functions it links into an executable, according to the function prologue code, and stores them into the “.pdata” section of the executable. In practice, this is a sequence of `RUNTIME_FUNCTION` objects, sorted by `BeginAddress`. But in our case *`embedded.cc`* does not contain code to compile but only assembly binaries; consequently, no PDATA are generated for builtin code. In fact, the .pdata section (dumped with dumpbin.exe /PDATA) shows a big gap that corresponds to that region in the executable:

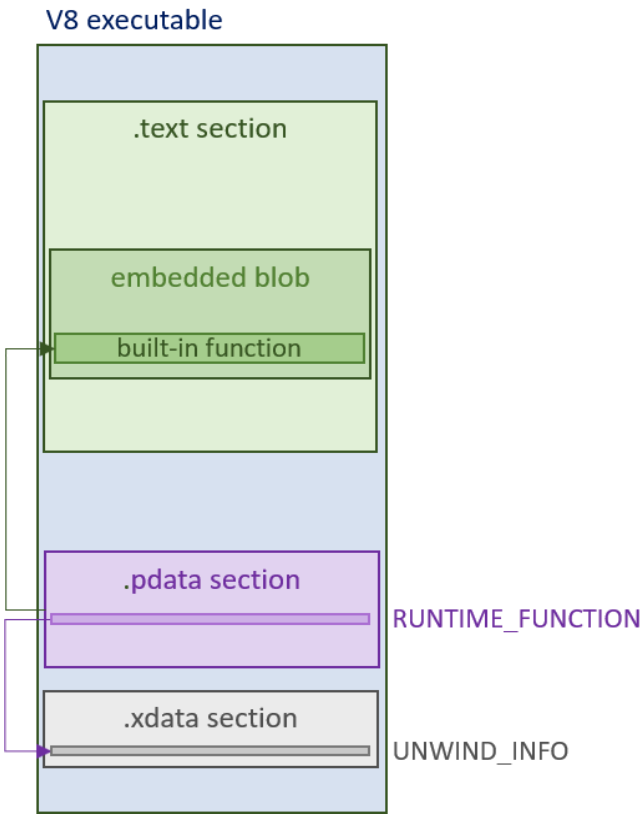
```
001C041C 018973E0 018973F9 0243C560  ??1?$_Vector_alloc@U?$_Vec_...
001C0428 01897400 0189749B 0243C568  ?_Free_proxy@?$_Vector_alloc@U?$_Vec_base...
001C0434 018974A0 018976A4 0243C570  ?SetSnapshotFromFile@internal@v8@@YAXP...
001C044C 01D63080 01D63130 0243C580  ?RoundUpToPowerOfTwo32@bits@base@v8...
001C0458 01D63130 01D631F5 0243C588  ?RoundUpToPowerOfTwo64@bits@base@v8...
```

To fix this issue we just need to add the right assembly directives into `embedded.cc` to register a `RUNTIME_FUNCTION` entry for each built-in function:

```
// pdata for all the code in the embedded blob.
.section .pdata
  .rva v8_Default_embedded_blob_data_ + .BeginAddressBuiltin_0
  .rva v8_Default_embedded_blob_data_ + .EndAddressBuiltin_0
  .rva Builtins_UnwindInfo
  .rva v8_Default_embedded_blob_data_ + .BeginAddressBuiltin_1
  .rva v8_Default_embedded_blob_data_ + .EndAddressBuiltin_1
  .rva Builtins_UnwindInfo
  [...]
  .rva v8_Default_embedded_blob_data_ + .BeginAddressBuiltin_N-1
  .rva v8_Default_embedded_blob_data_ + .EndAddressBuiltin_N-1
  .rva Builtins_UnwindInfo
```

In the same file, we need to emit a single `UNWIND_INFO` entry, since the XDATA is the same for each built-in function:

```
// xdata for all the code in the embedded blob.
.section .xdata
Builtins_UnwindInfo:
  .byte 0x1,0x4,0x2,0x5,0x4,0x3,0x1,0x50
```



Implementation

The following CLs contains the changes required to emit unwind info for V8 code in Win64:

- (Chromium) - Enable Win64 stack walking for jitted frames
- (V8) - Enable Win64 stack walking

Conclusions

This solution has several advantages:

- It is very simple; we don't have to deal with code motion (due to GC) or code deletion.
- It causes no impact in V8 runtime performances.
- It requires minimal or no changes in the embedders.

The only disadvantage is that there can be a small imprecision in the data collected by performance tools:

- Since we register a single `RUNTIME_FUNCTION` to cover the whole code region of an Isolate or a WASM, stack walking can be imprecise when the instruction pointer is exactly at the beginning of the prologue or in an off-heap trampoline.
- Since builtins functions do not follow the x64 Windows ABI, stack walking can fail in a few (statistically infrequent) cases.

This seems like a reasonable price to pay for a very simple implementation.