原文地址:http://drops.wooyun.org/tips/11565

Author:exp-sky@腾讯玄武实验室

0x00 前言

本文源自一次与TK闲聊，期间得知成功绕过CFG的经过与细节(参考：[利用Chakra JIT绕过DEP和CFG])。随即出于对技术的兴趣，也抽出一些时间看了相关的东西，结果发现了另一处绕过CFG的位置。所以这篇文章中提到的思路与技术归根结底是来自TK提示的，在此特别感谢。

关于CFG的分析文章已经有很多了，想要了解的话可以参考我之前在HitCon 2015上的演讲(spartan 0day & exploit)。要说明的是，本文的内容即为我演讲中马赛克的部分，至此通过一次内存写实现edge的任意代码执行方法就全部公开了。

0x01 Chakra调用函数的逻辑

chakra引擎在函数调用时，会根据所调用函数状态的不同进行不同的处理。比如第一次调用的函数、多次调用的函数、DOM接口函数及经过jit编译后的函数。不同的函数类型会有不同的处理流程，而这些不同的处理都会通过`Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >`函数调用`Js::JavascriptFunction::CallFunction<1>`函数来实现。

1.函数的首次调用与多次调用

当调用如下脚本时,`Js::JavascriptFunction::CallFunction<1>`函数会被`Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >`函数调用。

```
#!js
function test(){}

test();
```

如果函数是第一次被调用，则执行流程如下。

```
#!bash
chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >
    |-chakra!Js::JavascriptFunction::CallFunction<1>
        |-chakra!Js::JavascriptFunction::DeferredParsingThunk
            |-chakra!Js::JavascriptFunction::DeferredParse
            |-chakra!NativeCodeGenerator::CheckCodeGenThunk
                |-chakra!Js::InterpreterStackFrame::DelayDynamicInterpreterThunk
                    |-jmp_code
                        |-chakra!Js::InterpreterStackFrame::InterpreterThunk
```

如果再次调用这个函数的话，调用流程如下。

```
#!bash
chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >
    |-chakra!Js::JavascriptFunction::CallFunction<1>
        |-chakra!NativeCodeGenerator::CheckCodeGenThunk
            |-chakra!Js::InterpreterStackFrame::DelayDynamicInterpreterThunk
                |-jmp_code
                    |-chakra!Js::InterpreterStackFrame::InterpreterThunk
```

两次的调用流程大致是相同的，其中主要不同是因为，函数在第一次调用时候需要通过DeferredParsingThunk函数对其进行解析。其实函数只有在第一次调用时才进行进一步的初始化解析操作，这样设计主要是为了效率。而后续调用再直接解释执行。

分析发现，`Js::JavascriptFunction::CallFunction<1>`函数调用的子函数是通过Js::ScriptFunction对象中的数据获得的。后续调用的函数`Js::JavascriptFunction::DeferredParsingThunk`和`NativeCodeGenerator::CheckCodeGenThunk`都存在于`Js::ScriptFunction`对象中。两次调用中`Js::ScriptFunction`对象的变化。

第一次调用时的Js::ScriptFunction对象。

```
#!bash
0:010> u poi(06eaf050 )
chakra!Js::ScriptFunction::`vftable':

0:010> dd 06eaf050
06eaf050  5f695580 06eaf080 00000000 00000000

0:010> dd poi(06eaf050+4)
06eaf080  00000012 00000000 06e26c00 06e1fea0
06eaf090  5f8db3f0 00000000 5fb0b454 00000101
```

```
0:010> u poi(poi(06eaf050+4)+0x10)
chakra!Js::JavascriptFunction::DeferredParsingThunk:
```

第二次调用时的Js::ScriptFunction对象。

```
#!bash
0:010> u poi(06eaf050 )
chakra!Js::ScriptFunction::`vftable':

0:010> dd 06eaf050
06eaf050  5f695580 1ce1a0c0 00000000 00000000

0:010> dd poi(06eaf050+4)
1ce1a0c0  00000012 00000000 06e26c00 06e1fea0
1ce1a0d0  5f8db9e0 00000000 5fb0b454 00000101

0:010> u poi(poi(06eaf050+4)+0x10)
chakra!NativeCodeGenerator::CheckCodeGenThunk:
```

所以函数在第一次调用与后续调用的不同，是通过修改Js::ScriptFunction对象中的函数指针来实现的。

2.函数的jit

接下来我们看一下函数的jit。测试脚本代码如下，多次调用test1函数触发其jit。

```
#!js
function test1(num)
{
    return num + 1 + 2 + 3;
}

//触发jit

test1(1);
```

经过jit的Js::ScriptFunction对象。

```
#!bash
//新的调试，对象内存地址会不同

0:010> u poi(07103050 )
chakra!Js::ScriptFunction::`vftable':

0:010> dd 07103050
07103050  5f695580 1d7280c0 00000000 00000000

0:010> dd poi(07103050+4)
1d7280c0  00000012 00000000 07076c00 071080a0
1d7280d0  0a510600 00000000 5fb0b454 00000101

0:010> u poi(poi(07103050+4)+0x10)              //jit code
0a510600 55              push    ebp
0a510601 8bec            mov     ebp,esp
0a510603 81fc5cc9d005    cmp     esp,5D0C95Ch
0a510609 7f21            jg      0a51062c
0a51060b 6a00            push    0
0a51060d 6a00            push    0
0a51060f 68d0121b04      push    41B12D0h
0a510614 685c090000      push    95Ch
0a510619 e802955b55      call    chakra!ThreadContext::ProbeCurrentStack2 (5fac9b20)
0a51061e 0f1f4000        nop     dword ptr [eax]
0a510622 0f1f4000        nop     dword ptr [eax]
0a510626 0f1f4000        nop     dword ptr [eax]
0a51062a 6690            xchg    ax,ax
0a51062c 6a00            push    0
0a51062e 8d6424ec        lea     esp,[esp-14h]
0a510632 56              push    esi
0a510633 53              push    ebx
0a510634 b8488e0607      mov     eax,7068E48h
0a510639 8038ff          cmp     byte ptr [eax],0FFh
0a51063c 7402            je      0a510640
0a51063e fe00            inc     byte ptr [eax]
0a510640 8b450c          mov     eax,dword ptr [ebp+0Ch]
0a510643 25ffffff08      and     eax,8FFFFFFh
0a510648 0fbaf01b        btr     eax,1Bh
0a51064c 83d802          sbb     eax,2
0a51064f 7c2f            jl      0a510680
0a510651 8b5d14          mov     ebx,dword ptr [ebp+14h] //ebx = num
0a510654 8bc3            mov     eax,ebx        //eax = num (num << 1 & 1)
0a510656 d1f8            sar     eax,1          //eax = num >> 1
0a510658 732f            jae     0a510689
0a51065a 8bf0            mov     esi,eax
```

```
0a51065c 8bc6            mov     eax,esi
0a51065e 40              inc     eax             //num + 1
0a51065f 7040            jo      0a5106a1
0a510661 8bc8            mov     ecx,eax
0a510663 83c102          add     ecx,2           //num + 2
0a510666 7045            jo      0a5106ad
0a510668 8bc1            mov     eax,ecx
0a51066a 83c003          add     eax,3           //num + 3
0a51066d 704a            jo      0a5106b9
0a51066f 8bc8            mov     ecx,eax
0a510671 d1e1            shl     ecx,1           //ecx = num << 1
0a510673 7050            jo      0a5106c5
0a510675 41              inc     ecx             //ecx = num += 1
0a510676 8bd9            mov     ebx,ecx
0a510678 8bc3            mov     eax,ebx
0a51067a 5b              pop     ebx
0a51067b 5e              pop     esi
0a51067c 8be5            mov     esp,ebp
0a51067e 5d              pop     ebp
0a51067f c3              ret
```

`Js::ScriptFunction`对象中原本指向`NativeCodeGenerator::CheckCodeGenThunk`函数的指针，在jit之后变为指向jit code的指针。实现了直接调用函数jit code。

这里简单说明一下，在调用函数传递参数时，是先将参数左移一位，然后将最低位置1之后的值进行传递的（parameter = (num << 1) & 1）。所以其在获取参数之后的第一件事是将其右移1位，获取参数原始的值。至于为什么要这样，我想应该是因为脚本引擎垃圾回收机制导致的，引擎通过最低位来区分对象与数据。

```
#!bash
chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >
    |-chakra!Js::JavascriptFunction::CallFunction<1>
        |-jit code
```

调用jit函数时的调用栈如上所示，这就是chakra引擎调用jit函数的方法。

3.DOM接口函数

最后为了完整性，还有一类函数需要简单介绍一下，就是DOM接口函数，由其它引擎如渲染引擎提供的函数（理论上可以为任何其它引擎）。

```
#!js
document.createElement("button");
```

执行上面脚本则会通过下面的函数调用流程，最后调用到提供接口函数的引擎中。

```
#!bash
chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >
    |-chakra!Js::JavascriptFunction::CallFunction<1>
        |-chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk //调用dom接口函数
            |-dom_interface_function    //EDGEHTML!CFastDOM::CDocument::Trampoline_createElement
```

当调用dom接口函数时，`Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >`函数及后续的处理流程中所使用的Function对象与前面不同，使用的是`Js::JavascriptExternalFunction`对象。然后与前面的函数调用类似，也是通过解析对象内的函数指针，并对其进行调用，最终进入到想要调用的DOM接口函数中。

```
#!bash
0:010> u poi(06f2cea0)
chakra!Js::JavascriptExternalFunction::`vftable':

0:010> dd 06f2cea0
06f2cea0  5f696c4c 06e6f7a0 00000000 00000000

0:010> dd poi(06f2cea0+4)
06e6f7a0  00000012 00000000 06e76c00 06f040a0
06e6f7b0  5f8c6130 00000000 5fb0b454 00000101

0:010> u poi(poi(06f2cea0+4)+0x10)
chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk:
```

这就是chakra引擎对不同类型函数的不同调用的方式。

0x02 漏洞与利用

经过前面对chakra引擎各种调用函数方法的介绍，我们再来看一下本文的重点绕过cfg的漏洞。前面提到的在第一次调用脚本创建的函数时与后续调用此函数会有不同的流程。这里我们再看一下此处的逻辑，调用栈如下。

```
#!bash
//第一次调用
chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >
    |-chakra!Js::JavascriptFunction::CallFunction<1>
        |-chakra!Js::JavascriptFunction::DeferredParsingThunk
            |-chakra!Js::JavascriptFunction::DeferredParse    //获取NativeCodeGenerator::CheckCodeGenThunk函数
            |-chakra!NativeCodeGenerator::CheckCodeGenThunk
                |-chakra!Js::InterpreterStackFrame::DelayDynamicInterpreterThunk
                    |-jmp_code
                        |-chakra!Js::InterpreterStackFrame::InterpreterThunk
```

在前面没有提到的是，上面调用流程中的Js::JavascriptFunction::DeferredParse函数。此函数内部会进行函数解析相关的工作，并且返回NativeCodeGenerator::CheckCodeGenThunk函数的指针，然后在返回Js::JavascriptFunction::DeferredParsingThunk函数后对其进行调用。NativeCodeGenerator::CheckCodeGenThunk函数的指针也是通过解析Js::JavascriptFunction对象获得的。代码如下。

```
#!js
int __cdecl Js::JavascriptFunction::DeferredParsingThunk(struct Js::ScriptFunction *p_script_function)
{
    NativeCodeGenerator_CheckCodeGenThunk = Js::JavascriptFunction::DeferredParse(&p_script_function);
    return NativeCodeGenerator_CheckCodeGenThunk();
}


.text:002AB3F0 push    ebp
.text:002AB3F1 mov     ebp, esp
.text:002AB3F3 lea     eax, [esp+p_script_function]
.text:002AB3F7 push    eax             ; struct Js::ScriptFunction **
.text:002AB3F8 call    Js::JavascriptFunction::DeferredParse
.text:002AB3FD pop     ebp
.text:002AB3FE jmp     eax
```

在这个跳转位置上并没有对eax中的函数指针进行CFG检查。所以可以利用其进行eip劫持。不过还首先还要知道Js::JavascriptFunction::DeferredParse函数返回的NativeCodeGenerator::CheckCodeGenThunk函数指针是如何通过Js::ScriptFunction对象何解析出来的。解析过程如下。

```
#!bash
0:010> u poi(070af050)
chakra!Js::ScriptFunction::`vftable':

0:010> dd 070af050 + 14
070af064  076690e0 5fb11ef4 00000000 00000000

0:010> dd 076690e0 + 10
076690f0  076690e0 04186628 07065f90 00000000

0:010> dd 076690e0 + 28
07669108  07010dc0 000001a8 00000035 00000000

0:010> dd 07010dc0
07010dc0  5f696000 05a452b8 00000000 5f8db9e0

0:010> u 5f8db9e0
chakra!NativeCodeGenerator::CheckCodeGenThunk:
```

如上所述，Js::JavascriptFunction::DeferredParse通过解析Js::ScriptFunction对象获取NativeCodeGenerator::CheckCodeGenThunk函数指针，解析方法简写为[[[Js::ScriptFunction+14]+10]+28]+0c。所以只要伪造此处内存中的数据，即可通过调用函数来间接触发Js::JavascriptFunction::DeferredParse函数的调用，进而劫持eip，具体如下。

```
#!bash
0:010> g
Breakpoint 0 hit
eax=603ba064 ebx=063fba10 ecx=063fba40 edx=063fba40 esi=00000001 edi=058fc6b0
eip=603ba064 esp=058fc414 ebp=058fc454 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
chakra!`dynamic initializer for 'DOMFastPathInfo::getterTable''+0x734:
603ba064 94              xchg    eax,esp
603ba065 c3              ret
```

这样就绕过了cfg，成功劫持了eip。这种方法简单稳定，在获得了内存读写能力时使用是很方便的。此漏洞已于2015年7月25日报告微软。

0x03 修补方案

本文所述的漏洞微软已经修补，修补方案也比较简单就是对此处跳转增加cfg检查。

```
#!bash
.text:002AB460 push    ebp
.text:002AB461 mov     ebp, esp
.text:002AB463 lea     eax, [esp+arg_0]
.text:002AB467 push    eax
```

```
.text:002AB468 call    Js::JavascriptFunction::DeferredParse
.text:002AB46D mov     ecx, eax        ; this
.text:002AB46F call    ds:___guard_check_icall_fptr  //增加cfg检查
.text:002AB475 mov     eax, ecx
.text:002AB477 pop     ebp
.text:002AB478 jmp     eax
.text:00
```

0x04 参考

- 利用Chakra JIT绕过DEP和CFG
- spartan 0day & exploit