Pwn2Own 2018 Safari 漏洞利用开发记录系列 Part 2：Apple Safari浏览器漏洞挖掘之旅（下篇）

mss**** / 2018-08-10 08:52:25 / 浏览数 2708 技术文章 技术文章 顶(0) 踩(0)

原文：https://blog.ret2.io/2018/06/13/pwn2own-2018-vulnerability-discovery/

在本文的上篇中，我们为读者详细介绍了在挖掘像浏览器这样高度复杂的攻击对象的过程中，如何通过初步侦查确定合理的攻击目标，选择合适的漏洞挖掘方法，最后，还为

通过语法合成JavaScript代码

我们的fuzzer的JS测试用例的生成，是由语法来驱动的，并得到了定制版本的Mozilla dharma的加持。实际上，这里的定制化，主要是添加了一些方便的方法，使我们能够生成更加"相互依赖"的代码。

我们为dharma提供了两类JS语法：Libraries和Drivers。

Library语法主要有三项用途：

- 定义用于初始化特定JavaScript对象或类型（数组、字符串等）的规则
- 定义为这些对象或类型生成所有有效的成员函数（例如，array.sort()）的规则
- 定义语法或语义相似的规则集

driver语法可以为我们的library语法提供更高级的脚手架。通常情况下，Driver语法首先使用之前利用library定义的规则来初始化一组各不相同的JavaScript变量。之后，它

- 实现JavaScript变量之间的强制交互
- 对这些变量进行随机修改、删除或重新初始化变量
- 根据变量类型调用相应的成员函数

集合允许我们利用driver语法表示更加高级的"指导方针"。例如，下面是我们为JSArray对象编写的一些集合：

```
CallbackMethod :=
   +every+
   +filter+
   +reduce+
   ...

InPlaceMethod :=
   +sort+
   +splice+
   +shift+
   ...
```

如果我们希望一个数组在偶然的情况下随机修改自己，只需在我们正在构建的更大的语法结构中包含一个+JSArray:InPlaceMethod+语句即可。如果没有集合的话，我们将

集合中的每个条目都会生成各个成员函数（或API）的随机选择签名，并且会根据需要触发其他JavaScript结构的生成过程：

```
sort :=
   sort()
   sort(+GenericFunc:TwoArgNumericCmp+)
   ...
```

这使我们在编写driver语法时获得了更大的灵活性。因为，我们只需要为测试用例的生成提供粗略的限制条件，然后让概率决定哪种可能性变为现实。

最终，我们可以使用一个"mega-driver"语法完成大部分的模糊测试。这个语法将初始化我们编写的所有library语法中的变量，然后通过它们完成"混沌式的"生成操作：

```
try { someTypedArray1.reduce(function(acc, cval, c_index, c_array) { try{ c_array[c_index] = c_array.reduce((acc, cval, c_index, c_array) => { try{
    y.lastIndexOf(new Object(), -32760);c_array[c_index] = c_array.filter((arg) => { return (arg == new Object()) }); } catch(e){ } });c_array.reverse()
try { someRegex1[Symbol.search]("WUtazcQunqxEnKAbPkeIfNoQnSpOwQULMUoDVf") } catch (e) { }
try { someSet1.entries() } catch (e) { }
try { someWeakSet1.delete(function() {}) } catch (e) { }
try { Object.getOwnPropertyNames(someWeakSet1) } catch (e) { }
try { someString1.hasOwnProperty("toString") } catch (e) { }
try { for (var element in someObject1) { try{ someObject1[element] = someObject1[element].concat(); } catch(e) { } } } catch (e) { }
try { someTypedArray1 = new Uint16Array(someArrayBuffer1, 114) } catch (e) { }
try { someIntlNumberFormat1.formatToParts(+17064) } catch (e) { }
try { Math.sinh(11582) } catch (e) { }
try { someWeakSet1.add(someTypedArray1) } catch (e) { }
try { someDataView1.getFloat64(250, false) } catch (e) { }
try { Intl.NumberFormat.supportedLocalesOf("fi-FI") } catch (e) { }
try { someArray1[0] = someRegex1.test(String.fromCodePoint(669014) + "prAmHEKKXgdQBgytdTnyQnd") } catch (e) { }
try { someWeakSet1.delete(someObject1) } catch (e) { }
try { Intl.DateTimeFormat.supportedLocalesOf("ar-LB-u-hc-h11-nu-beng") } catch (e) { }
try { Intl.NumberFormat.supportedLocalesOf("es-PA-u-nu-kali") } catch (e) { }
try { for(var index=0; index < 7; index++){ someArray1[index] = someArray1.entries(); } } catch (e) { }
try { for(var index=0; index < 8; index++){ someArray1[index] = someArray1.join("iRq"); } } catch (e) { }
```

这是由我们混沌式的“mega-driver”语法生成的一个测试用例的代码片段

最初，这种语法是作为“控件”使用的，用于收集代码覆盖率并改进底层的library语法。我们的最初的计划是，对运行时功能有很好的了解之后，就利用driver语法构建更有趣

当然，这个计划并未如愿以偿，因为我们在构建library语法时发现了一个可利用的漏洞。

可扩展的模糊测试工具

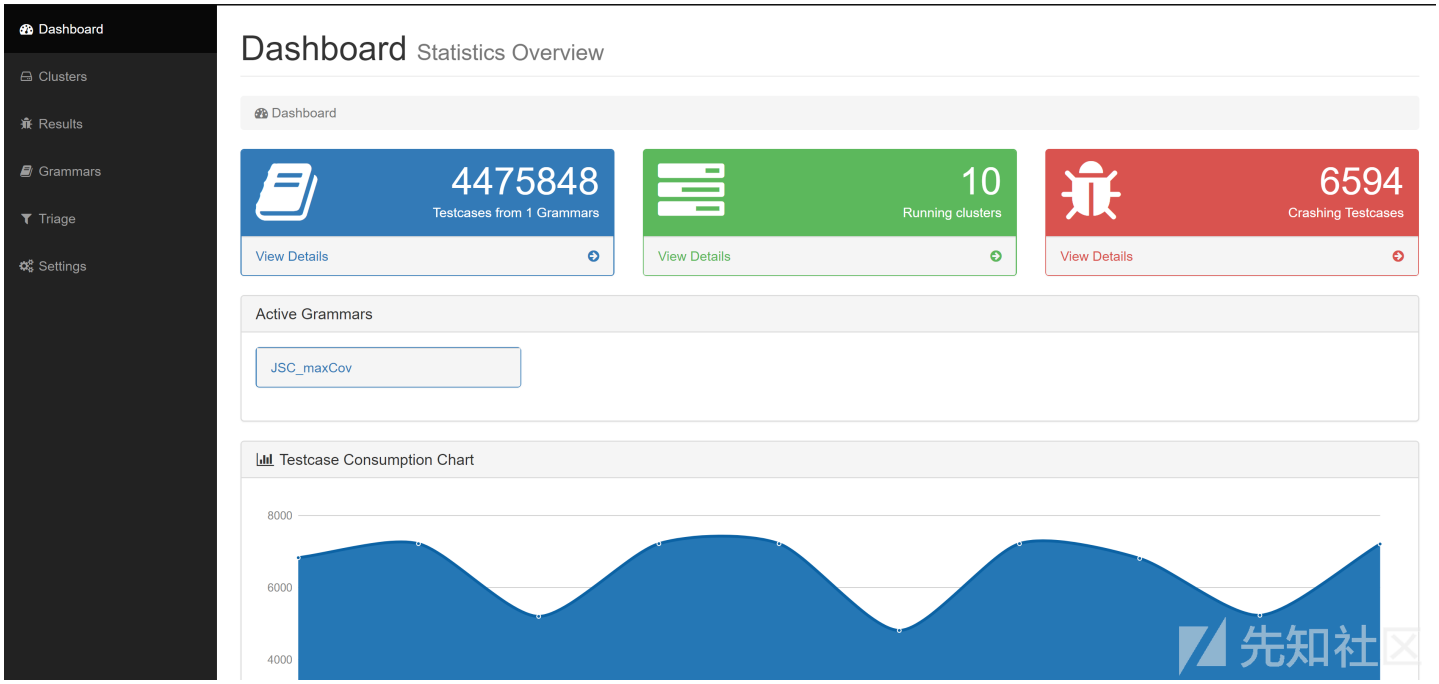
除了构建JS语法以生成更好的测试用例之外，我们还利用Python开发了另一种分布式和可扩展的模糊测试工具。根据传统，该工具的基础架构被设计为使用单个“master”节

master节点负责：

- 管理worker的配置
- 将worker报告的崩溃和测试用例分拣到不同的”桶”中
- 整理worker收集的代码覆盖率指标

因此，worker节点的相应责任包括：

- 生成测试用例
- 利用JSC启动测试用例并报告崩溃
- 通过随机抽样的测试用例收集代码覆盖率数据



master节点提供的fuzzer仪表板

在worker节点上，我们将针对JSC的调试版本进行模糊测试，它们都是利用ASAN编译的，并且运行在Ubuntu 16.04系统上。为了实现这一目的，我们没有对JavaScriptCore代码库进行任何的特殊修改。

Abstract Corpus的覆盖率

我们需要拿出稍许时间，让测试工具借助DynamoRIO对JSC执行的随机抽样收集代码覆盖率。该工具将汇总所有代码覆盖率方面的信息，并将其作为我们的JS语法能够为JS

在master节点上，我们可以利用仪表板自动生成并发布lcov报告，以便快速直观地展示整体代码覆盖率。通过聚合来自worker的所有新覆盖数据，该报告每隔几分钟就更新

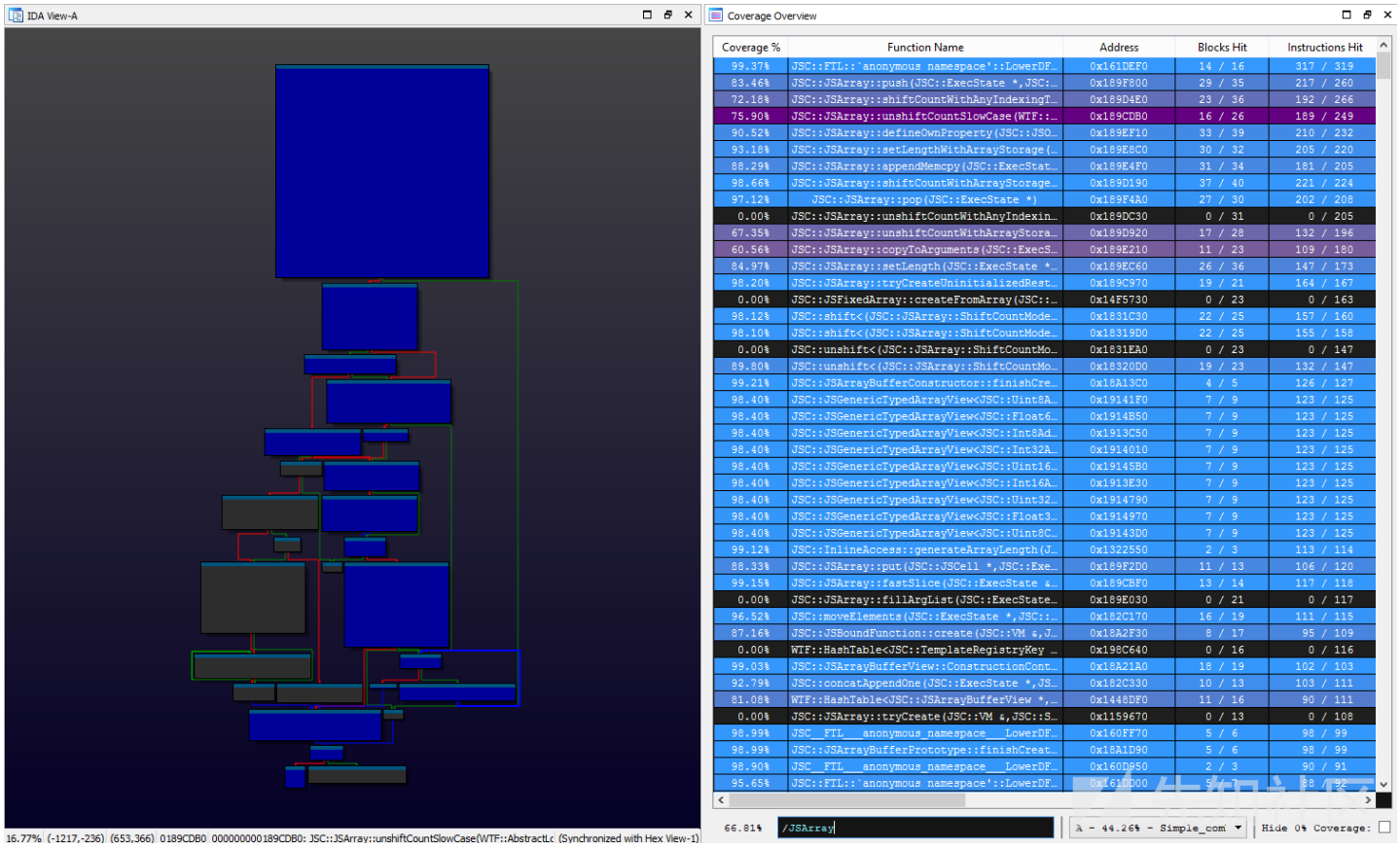
LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - runtime			
Test:	coverage.info	Lines:	20014	33801	59.2 %
Date:	2018-05-26 20:15:02	Functions:	0	0	-

Filename	Line Coverage			Functions	
AbstractModuleRecord.cpp		0.0 %	0 / 252	-	0 / 0
AbstractModuleRecord.h		0.0 %	0 / 14	-	0 / 0
ArgList.cpp		85.1 %	40 / 47	-	0 / 0
ArgList.h		50.9 %	28 / 55	-	0 / 0
ArrayBuffer.cpp		49.8 %	100 / 201	-	0 / 0
ArrayBuffer.h		51.5 %	17 / 33	-	0 / 0
ArrayBufferNeuteringWatchpoint.cpp		81.2 %	13 / 16	-	0 / 0
ArrayBufferNeuteringWatchpoint.h		100.0 %	3 / 3	-	0 / 0
ArrayBufferSharingMode.h		83.3 %	5 / 6	-	0 / 0
ArrayBufferView.cpp		0.0 %	0 / 16	-	0 / 0
ArrayBufferView.h		42.9 %	9 / 21	-	0 / 0
ArrayConstructor.cpp		71.7 %	43 / 60	-	0 / 0
ArrayConstructor.h		93.8 %	15 / 16	-	0 / 0
ArrayConventions.cpp		100.0 %	6 / 6	-	0 / 0
ArrayConventions.h		23.1 %	6 / 26	-	0 / 0
ArrayIteratorPrototype.cpp		100.0 %	5 / 5	-	0 / 0
ArrayIteratorPrototype.h		100.0 %	10 / 10	-	0 / 0
ArrayPrototype.cpp		92.9 %	724 / 779	-	0 / 0

查看master fuzzer节点生成的JSC的lcov报告

有时，我们还可以使用Lighthouse来下载和浏览运行过程中的覆盖率汇总数据。此外，我们还可以让工具收集启用了编译时优化和内联禁用的JSC发布版本的覆盖率情况，以



使用Lighthouse探索最新的JSC代码覆盖率汇总数据

当然，在源代码可用的时候，我们却来评估汇编代码级别的覆盖率似乎不合常理，但“更接近裸机”也有其优点：

- 利用反汇编器对棘手的边缘案例（edge cases）进行“放大”处理要更为自然
- 与lcov生成的静态HTML报告相比，Lighthouse生成的报告更具交互性

- Lighthouse为我们提供了详细的函数级别属性和覆盖率统计信息
- C++模板化代码即使只有一份源代码实现，却可以弄出多个二进制函数

这种工具组合为调整语法提供了宝贵的见解，以确保覆盖边缘案例、非标准操作模式以及JSC运行时中显著的执行差距。

本来，我们打算使用覆盖率来加强和指导源代码的人工审查工作，但是正如语法部分提到的，我们的计划被过早出现的漏洞所打断了。

漏洞的挖掘

当语法"corpus"覆盖了目标（runtime文件夹）中50%以上的代码之后，我们陆续开始看到许多特殊的崩溃。模糊测试在良好的覆盖率可见性的帮助之下，这种情况并不少见

当我们对这些新的崩溃进行分类时，我们注意到了一个特别有趣的callstack：

```
ASAN:SIGSEGV
=================================================================
==29641==ERROR: AddressSanitizer: SEGV on unknown address 0x0000977537dd (pc 0x7f303f347707 bp 0x7ffcc69020c0 sp 0x7ffcc69020c0 T0)
    #0 0x7f303f347706 in WTFCrash (/opt/jsc-debug/lib/libJavaScriptCore.so.1+0x4efa706)
    #1 0x7f303f347742 in WTFCrashWithSecurityImplication (/opt/jsc-debug/lib/libJavaScriptCore.so.1+0x4efa742)
    #2 0x483fc9 in JSC::StructureIDTable::get(unsigned int) (/opt/jsc-debug/bin/jsc+0x483fc9)
    #3 0x4850d3 in JSC::VM::getStructure(unsigned int) (/opt/jsc-debug/bin/jsc+0x4850d3)
    #4 0x48d0d2 in JSC::JSCell::structure(JSC::VM&) const (/opt/jsc-debug/bin/jsc+0x48d0d2)
    #5 0x48d85a in JSC::JSCell::classInfo(JSC::VM&) const (/opt/jsc-debug/bin/jsc+0x48d85a)
    #6 0x48d7ca in JSC::JSCell::inherits(JSC::VM&, JSC::ClassInfo const*) const (/opt/jsc-debug/bin/jsc+0x48d7ca)
    #7 0x49bd41 in JSC::JSObject* JSC::jsCast<JSC::JSObject*, JSC::JSCell>(JSC::JSCell*) (/opt/jsc-debug/bin/jsc+0x49bd41)
    #8 0x48c248 in JSC::asObject(JSC::JSCell*) (/opt/jsc-debug/bin/jsc+0x48c248)
```


WTFCrashWithSecurityImplication（...）会引起所有安全研究人员的注意

在将崩溃测试用例进行最小化之后，我们得到了以下概念证明（PoC）：

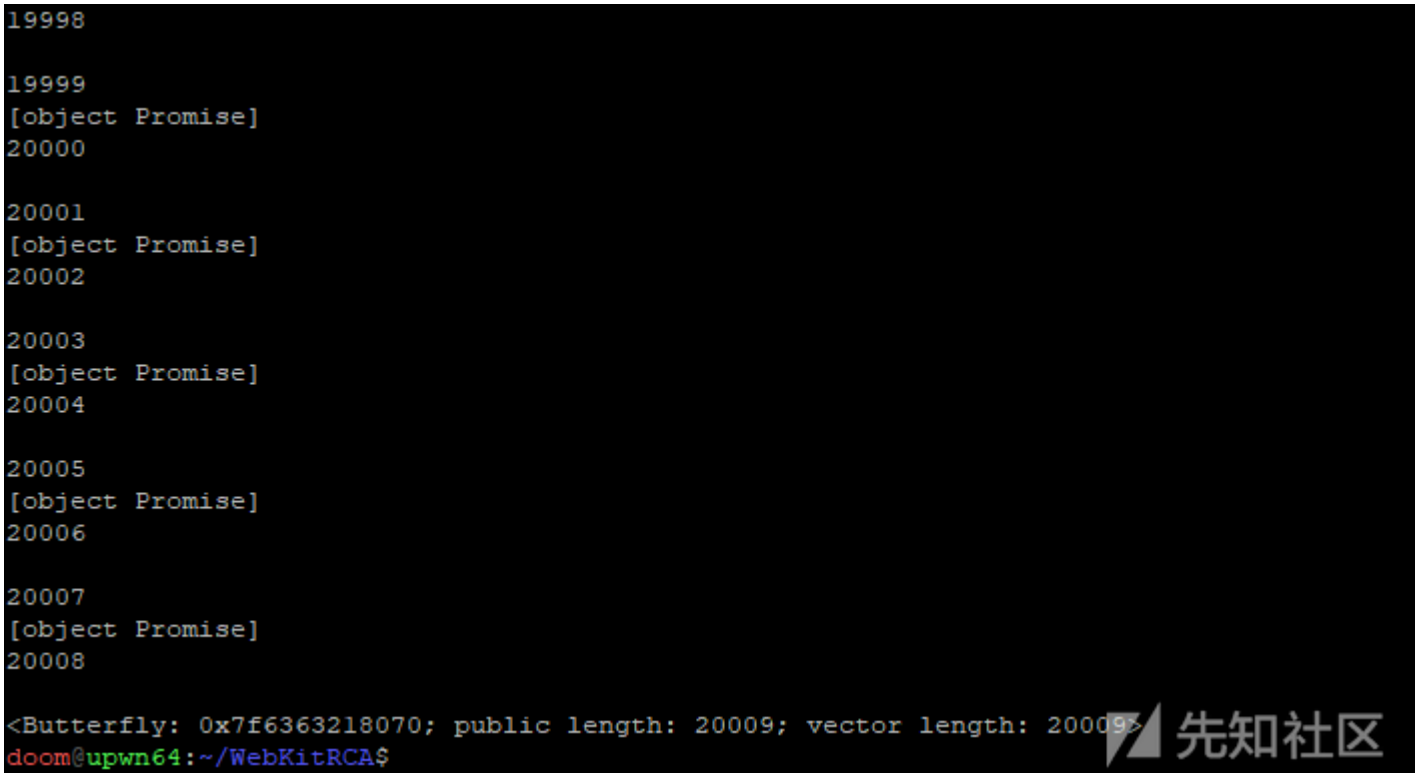
```
// initialize a JSArray (someArray1), with more JSArrays []
var someArray1 = Array(20009);
for (var i = 0; i < someArray1.length; i++) {
   someArray1[i] = [];
}

// ???
for(var index = 0; index < 3; index++) {
   someArray1.map(
       async function(cval, c_index, c_array) {
           c_array.reverse();
       });
}

// print array contents & debug info
for (var i = 0; i < someArray1.length; i++) {
   print(i);
   print(someArray1[i].toString());
}
print(describeArray(someArray1))
```

这个脚本（poc.js）会时断时续地表现出一些异常行为，我们觉得，这些行为是由模糊测试工具捕获的原始JSC崩溃造成的。

通过JSC重复运行测试用例，我们可以观察到一个无法解释的现象，其中一些JSPromise对象会随机存储到someArray1中：



利用JSC的调试版本测试最小化的PoC代码（译者注：动画效果见原文图像）

这是绝不应该发生的。someArray1只应该保存在最小化测试用例第一步中创建的JSArrays中。所以，该现象说明这里发生了一些非常错误的事情。

结束语

在审计软件的安全性能的时候，软件规模一旦达到一定程度，这项任务就会变得非常艰巨。在这篇文章中，我们讨论了一些非常有用的策略，来帮助读者确定研究目标、缩小

在下一篇博文中，我们将详细介绍如何使用高级调试技术，对我们最小化后的可疑测试用例(JSC漏洞)进行根源分析(root cause analysis，RCA)。一旦对这些问题进行了充分的理解，我们可以就其真正的安全影响得出明智的结论。

上一篇：WhatsApp漏洞分析 下一篇：vulnhub|渗透测试lampiao

1. 0 条回复
   - 动动手指，沙发就是你的了！