软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(2):247−265 [doi: 10.13328/j.cnki.jos.005626]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

攻击网页浏览器:面向脚本代码块的 ROP Gadget 注入*

袁平海[1,2], 曾庆凯[1,2], 张云剑[1,2], 刘 尧[1,2]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 即时编译机制(just-in-time compilation)改善了网页浏览器执行 JavaScript 脚本的性能,同时也为攻击者向浏览器进程注入恶意代码提供了便利.借助即时编译器,攻击者可以将脚本中的整型常数放置到动态代码缓存区,以便注入二进制恶意代码片段(称为 gadget).通过常数致盲等去毒化处理,基于常数的注入已经得到有效遏制.证实了不使用常数转而通过填充脚本代码块也能实施 gadget 注入,并实现图灵完备的计算功能.在编译一段给定的脚本代码时,即时编译器生成的动态代码中通常存在着一些固定的机器指令序列.这些指令序列的存在性不受常数致盲和地址空间布局随机化等安全机制的影响,同时,这些指令序列中可能蕴涵着攻击者期望的 gadget.在实施攻击时,攻击者可以汇集特定的脚本代码块来构造一个攻击脚本,再借助即时编译器来注入 gadget.在 x86-64 架构上评估了这种注入攻击在 SpiderMonkey 和 GoogleV8 这两个开源即时编译引擎上的可行性.通过给这两个引擎输入大量的 JavaScript 脚本,可以得到较为丰富的动态代码块.在这些动态代码块上的统计分析结果表明,这两个引擎生成的动态代码中都存在图灵完备的 gadget 集合.在实际攻击场景中,攻击者可以利用的脚本集合完全包含且远远多于实验用的脚本.因此,攻击者可以采用该方法注入需要的 gadget,以便构造出实现任意功能的 ROP(return-oriented programming)代码.

关键词: 网页浏览器;即时编译机制;即时返回导向编程;ROP(return-oriented programming) gadget 注入;图灵完备计算

中图法分类号: TP311

中文引用格式: 袁平海,曾庆凯,张云剑,刘尧.攻击网页浏览器:面向脚本代码块的 ROP Gadget 注入.软件学报,2020,31(2):247−265. http://www.jos.org.cn/1000-9825/5626.htm

英文引用格式: Yuan PH, Zeng QK, Zhang YJ, Liu Y. Attacking Web browser: ROP gadget injection by using JavaScript code blocks. Ruan Jian Xue Bao/Journal of Software, 2020,31(2):247−265 (in Chinese). http://www.jos.org.cn/1000-9825/5626.htm

Attacking Web Browser: ROP Gadget Injection by Using JavaScript Code Blocks

YUAN Ping-Hai[1,2], ZENG Qing-Kai[1,2], ZHANG Yun-Jian[1,2], LIU Yao[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Modern Web browsers introduce just-in-time (JIT) compilation mechanism to improve their performance on executing JavaScript applications. However, this mechanism has already been abused by attackers to inject malicious code. For instance, as JIT compilers may place JavaScript integers into code-cache in the form of operands of machine instructions, attackers can inject return-oriented programming (ROP) gadgets by crafting JavaScript integers. Fortunately, integer-based injection attacks have already been mitigated by techniques such as constant blinding. This work demonstrates that attackers can also inject ROP gadgets by using JavaScript

* 基金项目: 国家自然科学基金(61572248, 61431008, 61321491); 国家科技支撑计划(2012BAK26B01)

Foundation item: National Natural Science Foundation of China (61572248, 61431008, 61321491); National Key Technology Research and Development Program of China (2012BAK26B01)

收稿时间: 2017-06-21; 修改时间: 2017-09-05; 采用时间: 2018-07-17

code blocks instead of integer values. The idea of this injection scheme is based on the observation that the dynamic code generated by JIT compilers for a given JavaScript code snippet always has some immutable machine instruction sequences. The existence of these sequences is not affected by security mechanisms including constant blinding and address randomization. Moreover, these instruction sequences may contain ROP gadgets needed by attackers. Therefore, attackers can use JavaScript code blocks to obtain these gadgets in their attacks. The proposed injection scheme on SpiderMonkey and GoogleV8 is evaluated by running on x86-64 architecture. These two JIT engines are fed with JavaScript applications from well-known benchmarks and got a great many of dynamic code blocks. Statistical results show that Turing-complete sets of gadgets can be got in these code blocks. In real word attack senarios, the available JavaScript applications can be used by an adversary contain and are far more than those from benchmarks. Therefore, an adversary can apply the proposed scheme to inject gadgets for constructing ROP code to conduct arbitrary computation.

Key words: Web browser; just-in-time compilation; just-in-time return-oriented programming; ROP (return-oriented programming) gadget injection; Turing-complete computation

网页浏览器程序与我们的日常生活联系非常紧密,它们被广泛地安装到流行的电子设备上,包含个人电脑、智能手机、智能电视等.借助浏览器,网络用户可以在线收发电子邮件,查阅个人信息,完成网上银行转账等服务.浏览器的普遍流行,也让它们成为一个首要的攻击目标.攻击者通过实施漏洞利用攻击,能够截取或窃取浏览器收发存放的用户隐私信息.

在主流操作系统上,实施漏洞利用攻击并不容易.栈保护机制,如 stack canary[1],能够有效地缓解常见的基于栈的缓冲区溢出攻击.数据执行保护机制(data execution prevention,简称 DEP)[2,3]通过配置进程的内存页面属性,使得一个页面不能同时具有可写和可执行属性,能够有效地遏制传统的代码注入攻击(即 shellcode 注入攻击).当前,流行的攻击方法都通过复用受害进程空间中已有的代码来绕过 DEP 机制.典型地,攻击者可以利用返回导向编程(return-oriented programming,简称 ROP)技术[4]来构造恶意代码.幸运的是,地址空间布局随机化(address space layout randomization,简称 ASLR)[5]能够有效地缓解代码复用攻击.ASLR 将程序模块加载到一个随机选择的地址区域,让代码片段在内存中的地址拥有不确定性,增加了攻击者构造 ROP 代码的难度.在 Linux 系统上,现代浏览器程序都被编译成位置无关程序(position-independent executable)来支持 ASLR 的应用.

但是,浏览器提供的脚本执行环境,为绕过这些保护机制提供了新的可能.当浏览器程序中存在信息泄漏漏洞时(事实上,信息泄漏漏洞在浏览器这类大型软件中依然普遍存在.以 Google Chrome 为例,从 2016 年至今,已经暴露了 8 个信息泄露漏洞.最近的 CVE-2017-5021 漏洞,会在浏览器渲染精心构造的 HTML 网页时触发,可以让远程攻击者发动越界内存读操作),攻击者可以发动即时返回导向编程攻击(just-in-time return-oriented programming,简称 JIT-ROP)[6].这种攻击是在程序各模块加载完成后动态进行的,它通过运行脚本来查找静态代码(执行前已编译好的代码)中的可复用代码片段(称为 gadget),并将它们串接起来构造 ROP 代码.因此,JIT-ROP 能够有效地绕过 ASLR.有两类方法可以缓解 JIT-ROP 攻击:一种是通过去除代码页上的可读属性来防范静态代码信息泄露[7−10],从而提高 JIT-ROP 的攻击门槛;另一种是采用无 gadget 的保护机制(gadget-free)[11]来有效地根除 ROP 攻击,包括 JIT-ROP 攻击.Gadget-free 通过保护程序模块中的合法间接控制流指令免于滥用,同时消除截断指令中存在的间接控制流指令(以 x86 架构为例,该架构提供了一个高密度的变长指令集.在一个合法指令的中间开始进行解释,通常能够得到语义完全不同的功能指令.本文称这种指令为截断指令.从 ROP 攻击的视角来看,截断指令和正常指令没有任何区别),以此来遏制 ROP 攻击.然而,gadget-free 在动态代码上很可能会引入很大的性能开销,从而限制了它的实用性.因此,当前 JIT-ROP 攻击放弃在静态代码中查找 gadget,转而在动态代码中探索获取 gadget 的可能性.

近年来,攻击者通过滥用浏览器中引入的即时编译机制(just-in-time compilation)来注入恶意代码.现代网页浏览器为了提高执行 JavaScript 脚本的性能,会动态地将脚本编译成机器代码,然后再执行(现在的主流浏览器都有自己的即时编译引擎,Firefox 中有 SpiderMonkey,Chrome 中有 V8,WindowsEdge 中有 Chakra).浏览器为了支持动态代码生成技术,不仅会提供即时编译器,也会提供可写可执行的代码缓存区.依据这个事实,一种可行的攻击方法是直接向代码缓存区中注入 shellcode.典型地,攻击者可以发动即时喷洒攻击(just-in-time spraying)[12].现在,即时编译引擎通过将动态代码及其数据分割到不同的内存页,并在数据页上实施 DEP 来缓解

这种攻击.同时,Zhang 等人[13]提出了一个在动态代码页上严格实施 DEP 的方案,能够有效地抵抗这种攻击.另一种可行的攻击方法是通过控制脚本中的整型常数来注入 gadget[14],当即时编译器处理这些常数时,会把它们作为机器指令的操作数放置到代码缓存区中.脚本中的一些常数所对应的十六进制编码可以是攻击者期望的 gadget 的编码.这种恶意代码注入方法被称为基于常数的 gadget 注入.另外,Maisuradze 等人[15]证实,隐式常数(特指动态代码中的直接控制流转移指令的操作数)也可以被控制,并用于 gadget 注入.

基于显式常数的 gadget 注入可以用常数致盲的方法进行有效的防御[16].以该机制在 GoogleV8 中的实现为例,在生成的机器代码过程中,它会用一对随机数来替换原来的常数,原常数可以通过随机数还原出来.常数致盲让攻击者难以在代码缓存区见到自己期望的 gadget 的编码.对于那些可能逃避常数致盲的案例,可以通过对原脚本代码做多态变形来校正[17].另外,为了防范基于隐式常数的注入,可以参考 G-free[11]的实现方法来确保隐式常数不能被解码成 gadget.

本工作证实:不使用显式常数和隐式常数,转而通过使用脚本代码块,也能实施 gadget 注入,并实现图灵完备的计算功能.我们发现,在即时编译器为一个给定的脚本代码块生成的动态代码中,通常存在一些固定的指令序列,这些序列的存在性不受常数致盲和地址空间布局随机化等安全机制的影响.而且在这些固定的指令序列中,可能存在着攻击者期望的 gadget.在实施攻击时,攻击者可以汇集特定的脚本代码块来构造攻击脚本,再借助即时编译器来注入自己期望的 gadget.

本文在 x86-64 架构上评估了这种注入攻击在 SpiderMonkey 和 GoogleV8 这两个开源即时编译引擎上的可行性.将 Octane 等基准测试程序集中的 JavaScript 脚本作为这两个引擎的输入,可以分别得到 8 422 个和 10 846 个动态代码片段.我们分析并统计了各类 gagdget 在这些代码片段中的存在性,结果显示,在每个引擎生成的动态代码片段中,都能找到图灵完备的 gadget 集合.在实际的攻击场景中,攻击者可以利用的 JavaScript 脚本集合包含且远远大于上述测试集,因此,攻击者总是可以注入自己需要的 gadget 来构造实现任意恶意功能的 ROP 代码.我们的分析系统可以维护脚本代码块与 gadget 之间的对应关系,并记录下了生成当前 gadget 所属动态代码块的编译器代号.基于这些信息,攻击者可以选定特定的脚本代码块,并通过设定该代码块被调用的次数来触发相应的编译器,以便将脚本转化成机器代码,注入期望的 gadget.

1 背景知识与威胁模型

本节简单阐述 ROP 攻击原理,即时编译机制及其滥用.同时,为了评估面向脚本代码块的 gadget 注入是否能够构造出实现任意功能的 ROP 代码,本节也会论述图灵完备的概念.最后会描述本工作采用的威胁模型.

1.1 ROP攻击原理

由于 DEP[2,3]安全机制在主流系统上的广泛配置,攻击者不能以输入数据的方式向受害进程注入并执行 shellcode.现在流行的攻击都利用受害进程中已有的代码来构造恶意代码.ROP[4]是一种典型的代码复用技术,它通过串接以 *ret* 指令结束的 gadget 来构造恶意功能代码.ROP 攻击代码的有效载荷(payload)是由 gadget 的地址和 gadget 需要的参数组成的.在发动攻击时,payload 会被填充到栈帧上(通常是一个伪造的栈帧);每个 gadget 在执行完后,会通过其后的 *ret* 指令来调用下一个 gadget,所有的 gadget 依次执行,实现恶意功能.需要强调的是,使用 *ret* 指令来串接 gadget 只是其中一种可行的方式,用间接 *call* 和间接 *jmp* 指令也可以串接 gadget[18,19].攻击者可以用这些指令结束的代码片段来构造攻击代码.

图 1 为一个真实的 ROP 攻击示例.该示例调用 *system* 函数执行“/bin/sh”这条命令,接着调用 *exit* 函数以结束该进程.假定 payload 已被攻击者使用缓冲区溢出漏洞填充到堆栈上,并且原始函数的返回地址在图中被 0x40067f 覆盖.当原始程序执行 *ret* 指令进行返回后,ROP 代码中第 1 个 gadget 将会执行,即 0x40067f 所表示的代码段“*pop*%*rdi*;*ret*”.寄存器 *rdi* 中的值在该 gadget 执行结束后将会指向字符串“/bin/sh”,而 *ret* 指令将获取并执行“*jmpq**0x200ae2(%*rip*)”这个 gadget.此时库函数 *system* 将会被调用.我们知道,在 x86-64 体系结构上,*system* 函数的功能是执行其存放在寄存器 *rdi* 中参数对应的命令.因此,此时“/bin/sh”命令将被执行.该命令执行后,将会执行代码片段“*pop*%*rdi*;*ret*”,最后,ROP 代码将会调用库函数 *exit*(0x21)结束进程.

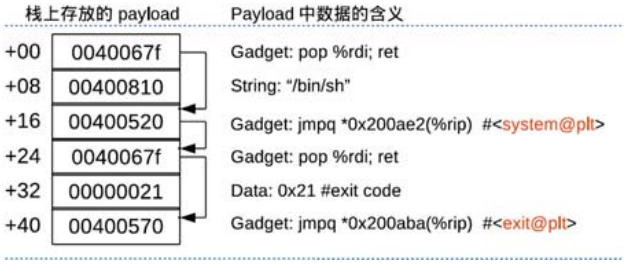


Fig.1 A ROP instance, the stack grows downwards

图 1 ROP 攻击实例,栈向地址空间生长

值得指出的是,ASLR[2]增加了实施 ROP 攻击的难度.但是,浏览器提供的网页脚本执行环境可以让攻击者发动 JIT-ROP 攻击[6]来绕过 ASLR.JIT-ROP 首先利用浏览器程序中存在的内存信息泄露漏洞来暴露地址空间中的静态代码块,然后在这些代码中搜寻 gadget 来构造 ROP 代码.如前所述,一些防止信息泄露的方案可以有效地防范静态代码信息泄露.同时,gadget-free 的安全机制可以有效地保护静态代码免于 ROP 攻击.现在,攻击者开始滥用浏览器中引入的即时编译机制来注入 gadget.

1.2 即时编译原理与滥用

为了提高执行 JavaScript 脚本的性能,现代浏览器都引入了即时编译机制.引入即时编译机制后,浏览器会将执行频率高的 JavaScript 脚本代码编译成机器代码后再执行.虽然执行机器代码要比解释执行(网页浏览器首先将 Javascript 编译成字节码,然后对字节码进行解释执行.结构上,解释器是一个大的 switch...case 语句块)同等语义的字节码更快,但是整个编译过程会引入开销.因此,浏览器会根据性能增益来决定是否对脚本代码进行编译以及是否需要对编译的代码做优化.通常,这个决定与脚本代码块的执行次数直接相关.以 SpiderMonkey 为例,它是以函数为单位进行编译的.默认情况下,当一个函数(或函数中的循环)执行 10 次后,会用基线编译器(开发代号为 Baseline)实施即时编译.该编译器会用同等语义的机器代码(通常会调用外部辅助函数)来替换每个字节码.当该函数的执行变得频繁(达到 1 100 次)时,会使用具有优化功能的编译器(开发代号为 Ion)对脚本进行再编译.这次编译会依据脚本函数的抽象语法树来生成性能更优的机器代码.为了支持即时编译机制,浏览器除了提供数个编译器以外,还必须提供可写可执行的动态代码缓存区.编译器会将自己生成的动态代码放置在该区域,以便后续加以执行.

但是,即时编译机制也为攻击者注入 gadget 提供了便利.典型地,攻击者可以在脚本中以整型常数的方式来注入自定义的 gadget.经过编译器后,这些常数会成为机器指令中的操作数,被放置于代码缓存区中.表 1 给出了 SpiderMonkey 的基线编译器对常数的编译结果.对左侧给出的 JavaScript 语句,SpiderMonkey 的基线编译器会将其编译成右侧所示的机器指令.

Table 1 Interger values in javscripts and their existences in JIT'ed code

表 1 Javascripts 中的整数值及其对应的动态代码编译结果

JavaScript 脚本语句	机器指令
var a=0xc358^0xc39448;	movabsq $0xfff880000000c358, %rbx movabsq $0xfff8800000c39448, %rcx movabsq $0xffffffffffffffff, %rdi movq 0x0(%rdi), %rdi call *0x0(%rdi)

显而易见,脚本中的常数已经出现在代码中.

注意,在机器指令的立即数操作数中,0xfff88000 为 SpiderMoneky 定义的数据类型标签,即 JSVAL_SHIFTED_TAG_INT32.从 ROP 攻击的角度来看,这些常数的十六进制编码可以被解释成攻击者期望的 gadget.例如,常数"0xc358"可以被解释成"pop rax;ret";"0xc39448"可以被解释成"xchg rax,rsp;ret".使用不同的常数可以

得到不同的 gadget;基于常数,攻击者可以很容易地注入自定义的 gadget.

在注入 gadget 后,攻击者还需获取 gadget 的位置才能构造 ROP 链,这样才能最终攻陷受害程序.Athanasakis 等人[14]描述了一个有效的攻击流程:攻击者可以首先获取一些先验知识,包括浏览器对 JavaScript 对象的内存布局以及即时编译器生成代码的确定性.在实施攻击时,先在脚本中构造函数对象,而且函数中包含期望的常数.基于即时编译器实现 gadget 注入后,攻击者可以通过函数对象定位函数所在的代码缓存区,进而得到 gadget 所在的内存地址.

1.3 图灵完备功能

ROP 攻击代码能够实现任意功能的等价描述,是它能够实现图灵完备功能.本节根据主流硬件架构的特点,定义出具体的图灵完备功能.最精简的图灵完备指令集由一条指令构成[20],如下所示的左半部分为该指令的定义,即 *subleq a,b,c*;而右半部分为对应的伪代码,即"*Mem*[*b*]=*Mem*[*b*]–*Mem*[*a*]; if (*Mem*[*b*]≤0) goto *c*"指令首先执行减法运算,即 *b* 指向的内存单元的值减去 *a* 对应的值.若减法操作后的结果小于或等于 0,那么控制流将会跳转到 *c* 指向的代码.尽管 subleq 指令功能强大,不过几乎所有的硬件架构都不支持,因此我们需要将其转换为等效的实际指令集.将该指令所包含的原子操作分开来看,我们可以获得以下 3 条指令.

1. 内存中的减法运算:*Mem*[*b*]=*Mem*[*b*]–*Mem*[*a*]
2. 比较 0 运算:*Mem*[*b*]≤0
3. 条件跳转:if (TURE) goto *c*

观察发现,第 1 条指令中减法运算中的两个操作数均为内存操作数,但常规硬件架构(例如 x86)不支持减法运算中的操作数同时为内存操作数.为此,我们需要增加指令以实现内存读取和写入操作.此外,还要添加实现系统调用功能的指令来支持输入/输出操作.综上,我们增添以下 3 条指令.

4. 加载内存内容到寄存器中:*loadReg*,*Mem*[*addr*]
5. 将寄存器内容存储至内存单元:*storeReg*,*Mem*[*addr*]
6. 实现系统调用功能的指令:*syscall*

这 6 条指令几乎存在于所有架构中.

1.4 威胁模型

本节定义攻击发生的条件,并且假设攻击者拥有的能力.这些假设与最近的攻防研究一致[14−17].

1.4.1 防御技术

根据安全防御机制的部署现状,我们假设操作系统和目标程序中施加了如下的安全机制.

1. 数据执行保护:目标操作系统中部署了 DEP 机制,即通过设置硬件的 NX(non-executable)位,让一个代码页不能同时具有写入和执行权限.该机制被应用到静态代码页上.
2. 地址空间随机化:目标操作系统配置了 ASLR 保护机制,这样,代码模块会被加载到一个随机选择的地址空间.除非拥有一个内存泄漏漏洞,否则攻击者就不能预测代码的位置.
3. 信息泄露防御机制和 gadget-free 安全机制:系统可能配置了预防代码泄露的防御机制,或实施 gadget-free 安全机制,这些安全机制确保攻击者不再能够复用静态代码段中的代码片段.
4. 常数致盲(constant blinding):为了防御基于常数的 gadget 注入,即时编译器中引入了常数致盲安全机制.由于该机制的实施,我们认为,攻击者不再能够通过常数来注入自定义的 gadget.
5. 其他保护机制:在静态代码上可能还实施了其他保护机制,如栈返回地址保护(stack canary)[21]和控制流完整性保护(control-flow integrity,简称 CFI)[22].这些机制能够增加攻击者劫持控制流的难度,但是并不能根除发动攻击的可能性.

1.4.2 威胁模型

我们现在枚举攻击发生的条件和攻击者的能力.

1. 内存泄漏漏洞:假定浏览器程序有一个内存泄漏漏洞,攻击者可以反复地利用它来泄漏可读的内存空

间.但只能读取数据,不能读取代码.

2. 控制流劫持:我们认为,浏览器程序有一个控制流劫持漏洞,可以让攻击者将控制流导向任何地方.值得注意的是,仅有这个漏洞不能发动攻击,因为攻击者还缺少可利用的 gadget.
3. 即时编译环境:根据当前主流浏览器的发展现状,我们认为,浏览器不但提供了一个脚本执行环境,而且还支持即时编译机制.同时,假设攻击者可以构造任意的 JavaScript 攻击脚本.当受害用户用浏览器访问攻击者控制的网页时,这个假设是非常正常的.

综上所述,在这种威胁模型中,攻击者可以利用内存泄漏漏洞和浏览器提供的脚本执行环境发动 JIT-ROP 攻击.同时,由于浏览器支持即时编译,攻击者可以借助即时编译器来注入 gadget.

2 面向脚本代码块的 Gadget 注入

本节对相关原理进行阐述.同时,为了评估该方法能否注入图灵完备的 gadget 以便构造出实现任意功能的 ROP 攻击代码,本节也会对获取动态代码的过程以及在动态代码中搜寻 gadget 的算法进行描述.

2.1 注入原理

浏览器中的即时编译器的应用场景与 gcc 等非即时编译器的应用场景相比,至少有两个显著的不同点.

1) 首先,后者的输入是本地用户控制的,但前者的输入可以被远程攻击者控制;
2) 后者的编译输出不会在当前编译进程的环境下运行,但前者的输入则通常会立即投入到当前进程中运行.

这两个特点决定了攻击者可以借助即时编译器向浏览器进程注入恶意代码.

另外,即时编译器与 gcc 这类编译器有其他差别.为了防范基于常数的 gadget 注入,即时编译器可能会引入常数致盲和地址空间布局随机化等安全机制.这些机制导致即时编译器每一次输出的编译结果有随机性.尽管对同一个脚本代码块编译后得到的结果总是语义等价的,但是机器指令序列不会完全相同.Gcc 通常不会引入这种随机性,因此对同一个代码块编译后得到的结果不但语义等价,而且机器指令序列也相同.

即使如此,我们也发现,在即时编译器为一个给定的脚本代码片段生成的机器代码中,通常存在一些固定不变的机器指令片段,而且这些指令片段中可能蕴含着攻击者期望的 gadget.攻击者可以利用这些 gadget 来构造 ROP 攻击代码.在实施攻击时,攻击者可以汇集特定的脚本代码块来构造一个攻击脚本,并依据需要触发的编译器代号来设置脚本代码块被调用执行的次数,以达到注入 gadget 的目的.此即面向脚本代码块的 gadget 注入.

在实际应用中,实施面向脚本代码块的 gadget 注入比基于常数的注入要复杂.因为即时编译器生成的代码结构很单一,特别是在基线编译器生成的动态代码中,可以找到的 gadget 类型通常非常少.如果攻击者只是简单地向即时编译器提供一个 JavaScript 脚本,然后在生成的动态代码中查找 gadget 来构造 ROP 攻击,那么这种攻击方法几乎是不可行的.为此,攻击者需要获取丰富的先验知识,了解 gadget 在动态代码中的存在性.

2.2 获取动态代码块

为了研究 gadget 在动态代码中的存在性,有两种方法可以使用.

- 第 1 种方法是在源代码级别分析即时编译引擎生成动态代码的算法,以便知道在何种条件下才能生成攻击者需要的 gadget.以 SpiderMonkey 为例,需要了解:(1) 基线编译器将字节码转换成机器代码的算法;(2) 优化编译器将抽象语法树转换成机器代码的算法;(3) 机器代码中保存和恢复寄存器信息的过程;(4) 解释执行环境到基线执行环境的上下文切换;(5) 虚拟机内部与外部的控制流切换.考虑到 SpiderMonkey 还需要在各个阶段设置异常处理机制,这种方法要了解的细节会很多.
- 另一种方法是以黑盒测试的思想来掌握脚本代码块与 gadget 之间的映射关系.向即时编译器输入大量的 JavaScript 脚本,可以得到丰富的动态代码块;然后,在动态代码块中查找 gadget 的存在性.基于这些映射信息,攻击者就可以汇集相应的脚本代码块来构造攻击脚本.虽然这种方法相对简单些,但攻击者也需要对即时编译器的架构有足够的了解,以便在引擎的源代码中实施代码插桩,将动态生成的机器

代码块保存到磁盘文件中,以便事后分析.

两种方法各有利弊.第 1 种方法有利于攻击者定制需要汇集的脚本代码块,但由于编译引擎非常复杂,要在源代码级别了解即时编译引擎的算法并不是一个简单的工作,而且对于那些从截断指令开始的 gadget,这种方法并不能观察到.第 2 种方法相对简单且通用,制定的算法可以在任意一个即时引擎上使用,但是这种方法对输入的脚本有依赖关系,输入的脚本不一定能够暴露出攻击者期望的 gadget.

本工作采用第 2 种方法,这是因为:

1) 本工作侧重于对面向脚本代码块的 gadget 注入做定性研究,即评估该方法能否注入图灵完备的 gadget 集合.
2) 这种方法具有通用性,可用于不同的即时编译引擎,在 SpiderMonkey 和 GoogleV8 上都适用.
3) 总是存在一些潜在的方法可以改善输入脚本质量,以便暴露攻击者期望的 gadget.

为了得到较丰富的动态代码片段,我们选择 Octane 等常见的基准测试集中的 JavaScript 脚本作为输入,喂给即时编译引擎.这些测试集通常是用于测试即时编译引擎的性能,本文使用它们则是为了得到动态代码片段.在实际的攻击场合,攻击者可以利用的脚本集合包含且远远大于这些测试集.因此,本文做的评估是充分性的,即如果这些测试集对应的动态代码中存在图灵完备的 gadget 集合,那么攻击者总是可以注入图灵完备的 gadget,以便构造实现任意功能的 ROP 代码.

为了将动态代码保存到磁盘文件以便事后分析,我们会对即时编译引擎中的每一个编译器进行插桩.虽然每个编译器的具体实现不同,但都会采用如下过程:申请临时的数据缓存区,并在该区域生成原始代码;依据实际代码大小申请动态代码缓存区,将原始代码及重定位信息等存储到该区域;对动态代码进行链接;将动态代码关联到一个特定的类实例,并返回该实例让外部引用即投入运行.由于动态代码的链接过程会修正代码块中与地址相关的指令,这些指令通常引用了动态数据对象或其他动态代码块.因此,插桩点设置在动态代码编译好链接完成之后、投入运行之前.

尽管攻击者期望得到的是 gadget,但他能控制的只是脚本代码.因此在整个分析系统中,需要记录脚本代码块与 gadget 之间的关联关系.为此,在存储动态代码块到文件的过程中,除了需要保存动态代码的内容,还需要保存脚本文件名以及被编译的脚本代码在脚本文件中的起始行号.因为即时编译器是以脚本函数为单位进行编译的,所以这两个信息足够将 gadget 与脚本函数关联起来.另外,我们还会保存动态代码块的大小及其在进程空间中的起始虚拟地址.为了滤除脚本中的常数构成的 gadget,我们还会保存该动态代码块所对应的字节码,以便查找这些脚本常数.

2.3 在动态代码块中查找Gadget

2.3.1 Gadget 的类型

我们先定义 gadget 的类型,再定义 gadget 的查找算法.与 Q[23]和 pyrop[24]类似,我们按行为语义定义了本工作用到的各类经典 gadget,见表 2($M[addr]$表示访问 *addr* 指向的内存单元,$\diamond_b$表示二元算术逻辑运算).注意,在 ROP 代码执行流下的“语义定义”与正常代码下不同.以 *NopG* 为例,它被定义为“无操作”是指除了栈指针寄存器(stack-pointer register,简称 SP)和指令指针寄存器(instruction-pointer register,简称 IP)之外,它不更改内存的状态和其他寄存器的内容.它的行为等同于执行 *ret* 指令.表 2 中 *AddrR*,*InR* 和 *OutR* 涉及到的寄存器默认不包括 IP 和 SP 这两个寄存器.*ArithG* 表示对两个寄存器执行算术逻辑运算的 gadget.*ArithConstG* 代表的 gadget 会对寄存器操作数和常量执行算术和逻辑运算;*ArithLoadG* 代表的 gadget 首先会对寄存器操作数和内存操作数执行算术和逻辑运算,然后将最终的结果加载到寄存器.*ArithStoreG* 代表的 gadget 会将最终结果存储在内存.

对比 Q 和 pyrop 中的内容,我们在此处添加了另外 4 种 gadget 类型,分别是 *RegJumpG*、*RegJumpModifyPayloadG*、*MemJumpG*、*MemJumpModifyPayloadG*.一般来说,间接 *call* 指令和间接 *jmp* 指令来用作结尾的代码片段将会映射到这些 gadget 类型.

通常,以间接 *call* 和间接 *jmp* 指令结束的代码片段会被映射到这几类 gadget.(*mem*)*RegJumpModifyPayloadG* 会对 payload 的内容进行重写,这是有循环功能的 ROP 代码应该极力避免的.消除这种副作用很简单,只要在运

行这类 gadget 之前执行一条“*ret* $4”即可.

Table 2 Gadgets that can be found by our algorithm

表 2 我们的算法能够找到的 gadget 类型

Gadget 类型	输入/输出寄存器	参数	行为语义定义	实例
NopG	无	无	无操作	*Ret*
RetNG	无	*value*	$SP_{old}=SP_{new}+value$	*ret* $4
MoveRegG	*InR*, *OutR*	无	*OutR=InR*	*mov eax*, *edx*; *ret*
LoadConstG	*OutR*, *value*	无	*OutR=value*	*mov ebp*, 0x21; *ret*
LoadMemG	*AddrR*, *OutR*	*offset*	*OutR=M*[*AddrR+offset*]	*pop ebp*; *ret*
StoreMemG	*AddrR*, *InR*	*offset*	*M*[*AddrR+offset*]=*InR*	*mov* [*ebx*+0x40], *eax*; *ret*
ArithG	InR_1, InR_2, *OutR*	$\diamond_b$	$OutR=InR_1 \diamond_b InR_2$	*add eax*, *ebx*; *ret*
ArithConstG	*InR*, *OutR*	*value*, $\diamond_b$	$OutR=InR \diamond_b value$	*add eax*, 0x20; *ret*
ArithLoadG	*AddrR*, *OutR*	*offset*, $\diamond_b$	$OutR \diamond_b=M[AddrR+offset]$	*add eax*, [*ebx*+0x40]; *ret*
ArithStoreG	*AddrR*, *InR*	*offset*, $\diamond_b$	$M[AddrR+offset] \diamond_b=InR$	*add* [*ebx*+0x40], *eax*; *ret*
RegJumpG	*AddrR*	*offset*	*IP=AddrR+offset*	*jmp eax*
RegJumpMod-ifyPayloadG	*AddrR*	*offset*	*IP=AddrR+offset*	e.g.1: push *ebx*; *ret*, e.g.2: *call eax*
MemJumpG	*AddrR*	*offset*	*IP*=[*AddrR+offset*]	*jmp* [*eax*]
MemJumpModifyPayloadG	*AddrR*	*offset*	*IP*=[*AddrR+offset*]	*call* [*eax*]

考虑到ROP代码实现条件转移的困难性,会为它提供两个构造方案,以便增加构造它的成功率.根据我们已有的知识,至少有两个通用方案可以实现条件转移.

- 一个方案是 Shacham 等人[4]设计的,其核心内容是把条件标志的值传递到栈指针寄存器上.他们分 3 个步骤实现了此功能:(1) 执行一些操作来设置或清除特定的标志;(2) 将寄存器 *EFLAGS* 上的标志传递给通用寄存器,以隔离标志;(3) 使用该标志生成所需的增量(清除该标志时,增量为 0),然后将此增量添加到栈指针寄存器 *SP* 中,以便根据需要修改 *SP* 值,实现 ROP 代码的条件转移.为了应用 Shacham 实现条件转移,我们依据相关工作给出的实现细节[4,25],在表 3 中定义了该算法可能额外需要用到的几种 gadget(不属于经典 gadget).
- 另一个实现方案是基于我们以前的工作[26].该工作定义了一种称为 if-gadget 的复合 gadget.它以条件控制流指令开始,它的两个分支的开头都是可重用的经典 gadget.这两个 gadget 将分别从两个存储单元中获取下一个 gadget 的地址,有时候存储单元中的内容可能早已加载到了寄存器中.因此,if-gadget 可以帮助 ROP 代码实现条件转移逻辑.表 4 中右栏给出了一个 if-gadget 实例,它以 0x8048408 处的 *je* 指令开始.该 if-gadget 的机器代码是由左栏中的 C 代码在编译后生成.该代码表示当函数指针不为空时,调用子函数.

Table 3 Gadget types that may be used by Shacham's method for achieving conditional jumps

表 3 Shacham 方法实现条件转移可能需要用到的 gadget 类型

Gadget 类型	语义	示例
lahfG	将状态标志加载到 AH 寄存器中	*lahf*; *ret*
pushfG	将 EFLAGS 寄存器的内容保存到栈上	*pushfg*; *ret*
negG	对操作数的内容取反	*negrax*; *ret*

Table 4 A snippet with an if-gadget

表 4 包含 if-gadget 的片段

C 语言源代码	汇编代码

typedef void (*ft)(void);
void demo(ft fun)
{
if (fun) fun();
}
	080483ff ⟨demo⟩:
	80483ff: sub $0xc,%esp
	8048402: mov 0x10(%esp), %eax
	8048406: test %eax, %eax
	8048408: je 804840c
	804840a: call *%eax
	804840c: add $0xc, %esp
	804840f: ret

2.3.2 Gadget 的查找算法

每个 gadget 不仅必须满足对应的语义定义,而且有一些特定的约束.因此,gadget 有 4 个与 Q[23]中定义类似的属性要求.

1. 功能性.每种类型的 gadget 都是通过语义分析获得的,并且定义中的最弱的先决条件必须满足,因此每个 gadget 都应具有表 2 中定义的功能.
2. 控制保持.gadget 以间接 call 指令,间接 jmp 指令或 ret 指令结束,使得控制流能够从上一个 gadget 传递到下一个 gadget.
3. 已知的副作用.gadget 没有未知的副作用,不能重写不想重写的任何内存内容.
4. 常量堆栈偏移.gadget 在执行后需要将堆栈指针增加某个常数值.

找到表 2 中定义的 gadget 后,进一步搜索 if-gadgets.为了找到 if-gadgets,我们需要在目标代码块中对所有的条件判断指令的起始地址以及它们分支的目标地址进行记录.

为了确保找到的 gadget 不是脚本中的常数引入的,也不是隐式常数所引入的,而且其存在性不受常数致盲和地址空间布局随机化的影响,我们会根据相关攻击原理[14,15,17],按照如下条件对找到的 gadget 进行过滤:

1) 对脚本中的常数在动态代码中的出现,保证其十六进制序列的任意字节不会被任何 gadget 利用;
2) 同时,对常数致盲用到的随机数以及可能受地址空间布局随机化影响的操作数,保证其十六进制序列的任意字节不会被任何 gadget 利用;
3) 任何 gadget 都不会以直接控制流转移指令的操作数所覆盖的字节结束.

2.4 图灵完备的ROP代码构造方案

为了评估面向脚本代码块的注入方法能否得到图灵完备的 gadget 集合,还需要制定一个由 gadget 去实现图灵基本功能的方案.现代计算机架构提供了丰富的指令集,有助于程序使用简洁的机器指令来实现给定的功能,这已足以实现图灵完备的功能.

因此,受益于这样的指令集,构造基于 gadget 的 ROP 功能也非常方便.x86 架构上的 SUB 指令可直接实现图灵基本功能中定义的“减法运算”,不过做加法运算时,虽然可以用类似“sub(x,sub(0,y))”的方式来实现,但是我们可以直接用 add 指令实现.对其他算术逻辑运算也是如此.因此,“减法运算”这种基本功能可用 *ArithG*、*ArithConstG*、*ArithLoadG* 和 *ArithStoreG* 这几类 gadget 来实现.

“比较操作”在 x86 架构上也有多种实现方法.除了常用的 cmp 指令与 test 指令以外,加法指令 add 与减法指令 sub 也可用于和 0 比较的操作,因为它们能够修改一些条件标志位.

“条件转移”功能可以用第 2.3.1 节所述的两种方案来实现.但是,借助于 if-gadget 的实现更加简洁.通常,将 if-gadget 与常见的经典 gadget 结合使用可以实现条件转移逻辑.

在 x86 架构上因为 mov 指令支持操作数中可以有一个内存操作数,因此可以使用 MOV 指令将内存内容加载到寄存器中或将寄存器内容存储到内存单元.因此,一些算术逻辑运算指令如果也支持一个内存操作数,那么该指令也就可以用来实现相同的功能.或者相反地,存储寄存器的内容到内存中.所以,*LoadMemG* 和 *ArithLoadG* 可以实现“加载内存内容到寄存器”的功能;同时,*StoreMemG* 和 *ArithStoreG* 可以实现“存储寄存器内容到内存”的功能.

用户态代码若要实现系统调用的功能,我们可通过 INT、SYSENTER、SYSCALL 等特殊指令,但是某些系统库中大量的封装函数也能实现系统调用功能,特别是 Linux 系统中的 libc 库.因此,我们在构造 ROP 代码时倾向于用库函数实现系统调用功能,因为即使系统上部署了某些安全机制(如地址空间布局随机化),gadget 也能将控制流传递给这些函数,实现系统调用功能.

3 分析系统的实现

本节阐述整个分析系统的实现,包括即时编译器插桩、动态代码块存储以及 gadget 的查找.

3.1 SpiderMoneky架构分析与插桩

SpiderMoneky 即时编译引擎被用于 Firefox 浏览器中.除了解释器及其字节码编译器外,它还有一个基线编译器和一个优化编译器,开发代号分别为 *Baseline* 和 *Ion*.*Baseline* 简单地用语义等价的机器指令序列来实现每个脚本字节码的功能;*Ion* 则会将脚本的抽象语法树转换成中间代码,经过优化处理后再生成机器码.

在源代码中,*Baseline* 编译器生成机器代码的算法很直接,主要过程封装在类 *BaselineCompiler* 中.*Ion* 生成机器代码的算法被封装在多个类中,其中,类 *CodeGenerator* 是最直接相关的类.为了在机器代码生成好并链接完成之后,以及在投入运行之前将动态代码块存储下来,我们在表 5 所列的两个函数中进行代码插桩.

Table 5 SpiderMoneky functions that are instrumented for dumping JIT'ed code

表 5 在 SpiderMoneky 函数中用于转储动态代码的插桩点

编译器代号	代码插桩之文件路径	代码插桩之函数名
BaselineCompiler	./jit/BaselineCompiler.cpp	*BaselineCompiler*::*compile*(·)
Ion	./jit/CodeGenerator.cpp	*CodeGenerator*::*link*(·)

在 SpiderMoneky 中,动态代码块的信息主要通过类 *JitCode* 来表示;每一个动态代码块都对应一个 *JitCode* 实例.通过调用类 *JitCode* 的成员函数 *raw*(·)和 *bufferSize*(·),可以得到动态代码块的起始虚拟地址和代码块大小.同时,可以调用其成员函数 *kind*(·)来查询该动态代码块是由哪一个即时编译器生成的.每一个被编译的脚本函数都是通过类 *JSScript* 来表示的,通过该类的成员函数 *filename*(·)和 *lineno*(·),可以得到脚本文件名和脚本函数在脚本文件中的起始行号.另外,通过 *JSScript* 类的实例,也可以得到脚本函数的字节码.在被插桩的函数中,我们可以获取当前被处理的脚本函数所对应的 *JitCode* 类实例和 *JSScript* 类实例.

同时,SpiderMoneky 的源代码中包含了一个名为 *js* 的宿主程序(hostprogram).通过该宿主程序,可以调用该即时编译引擎来处理 JavaScript 脚本.

3.2 GoogleV8架构分析与插桩

GoogleV8 即时编译引擎被用于 Chrome 浏览器中.除了解释器及其字节码编译器外,它有 3 个即时编译器,开发代号分别为 *Full-codegen*、*Crankshaft* 和 *TurboFan*.这 3 个编译器按优化深度递增.*Full-codegen* 简单地用功能等价的机器指令序列来实现每个字节码的语义;*Crankshaft* 和 *TurboFan* 则会将脚本的抽象语法树转换成中间代码,做优化处理后再生成机器码.与 *Crankshaft* 相比,编译器 *TurboFan* 拥有更多的特性和功能,运行速度更快.*TurboFan* 会是 GoogleV8 下一代的优化编译器.

在源代码中,与这 3 个编译器相关的数据结构和函数非常多,但封装机器代码生成算法的最直接相关类分别为 *FullCodeGenerator*、*LChunk* 和 *CodeGenerator*.为了在机器代码生成好并链接完成之后,以及在投入运行之前将动态代码块存储下来,我们在表 6 列出的 3 个函数中进行代码插桩.

Table 6 GoogleV8 functions that are instrumented for dumping JIT'ed code

表 6 在 GoogleV8 函数中用于转储动态代码的插桩点

编译器代号	代码插桩之文件路径	代码插桩之函数名
Full-codegen	./full-codegen/full-codegen.cc	*FullCodeGenerator*::*MakeCode*(·)
Crankshaft	./crankshaft/lithium.cc	*LChunk*::*Codegen*(·)
TurboFan	./compiler/code-generator.cc	*CodeGenerator*::*GenerateCode*(·)

在 GoogleV8 中,动态代码块是通过类 *Code* 来表示的;每一个动态代码块都对应一个 *Code* 类的实例.通过调用该类的成员函数 *instruction_start*(·)和 *instruction_size*(·),可以得到动态代码块的起始虚拟地址和代码块大小.同时,通过调用其成员函数 *is_crankshafted*(·)和 *is_turbofanned*(·),可以查询到当前代码块是由哪一个即时编译器生成的.每一个被编译的脚本函数都是通过类 *Script* 来表示的.可以通过该类的成员函数 *name*(·)和 *line_offset*(·)来得到脚本文件名和被编译函数在脚本文件中的起始行号;同时,通过成员函数 *source*(·)可以得到脚本函数的源代码.在被插桩的函数中,我们可以获取当前被处理的脚本函数所对应的 *Code* 类实例和 *Script* 类实例.

GoogleV8 的源代码中包含了一个名为 d8 的宿主程序.通过该宿主程序,可以调用该即时编译引擎来处理

JavaScript 脚本.

3.3 动态代码块的存储

动态代码块被存储到磁盘文件中,以便后续离线分析.每一个动态代码块都对应一个磁盘转储文件,这些文件被称为 dump 文件.这些文件是有格式的,图 2 的左侧给出了文件头部结构,文件头部的每个域的含义与其命名是一致的.字段“jshash”“type”和“lineno”分别表示当前被编译的 JavaScript 代码块的字节码所对应的 hash 值,编译器类型和被编译脚本代码块在文件中的起始行号.如下文所描述的,“jshash”值会成为转储文件名的一部分.为了避免 hash 值的碰撞,我们取用 *sha*256 值.为了避免不必要的冗余,同一块动态代码只会保留它的一份转储文件,插桩代码在转储之前会做该检查.字段“jscodeoft”和“jslen”分别表示字节码在 dump 文件中的位置和长度.字段“jitcodeoft”和“jitlen”分别表示动态代码在 dump 文件中的位置和长度.字段“startVA”表示动态代码在内存中的虚拟地址.字段“scriptFileName”表示脚本文件名.图 2 的右侧给出了一个 dump 文件的部分内容作为演示实例,该 dump 文件是在 SpiderMonkey 下生成的.根据示例中的内容可知,该动态代码块是由 *Ion* 编译器(*type*==1)所生成的;动态代码块在文件偏移 0x1b0 处开始,大小为 0xce0 个字节.同时,该动态代码块所对应的脚本函数在“richard.js”脚本文件中,开始行号为 0x2ca.

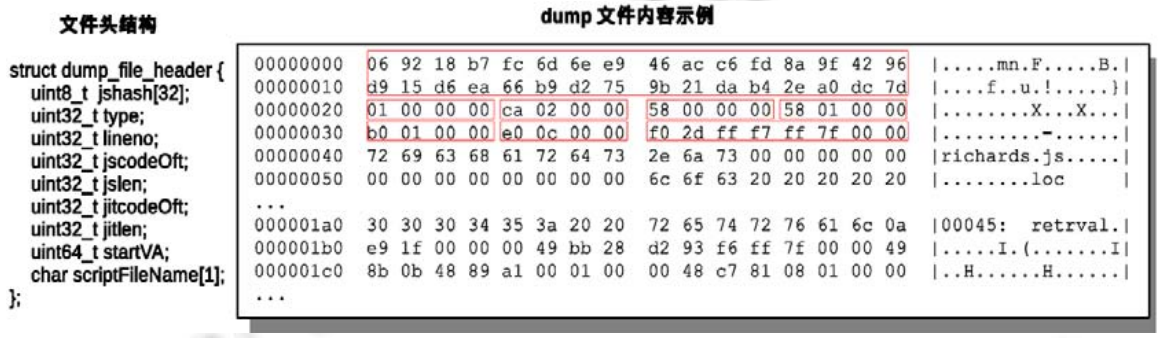


Fig.2 Format of dump file, and an instance for illustration

图 2 Dump 文件格式及示例文件

所有 dump 文件都统一命名,它们的文件名由 3 部分构成:前缀、字符化 hash 值,编译器编号.例如,上述的 dump 文件会被命名为“dump069218b7fc6d6ee946acc6fd8a9f4296d915d6ea66b9d2759b21dab42ea0dc7d01”.

需要指出的是,dump 文件名在整个信息系统中扮演关键字的角色.在离线分析时,我们会查找每一个 Javascript 脚本文件中的代码所编译得到的动态代码块,以及每一块中存在的 gadget.这些信息会依据即时编译器的不同分割开来,存放到若干分析结果文件中.以 SpiderMonkey 引擎下为例,由于该引擎有两个即时编译器,每个 Javascript 脚本文件会对应两个分析结果文件;每个结果文件被划分成多个节,节名用 dump 文件名来标识,每一个节存放了相应 dump 文件中所找到的 gadget.在实施面向脚本代码块的 gadget 注入时,我们会反过来:

1) 在结果文件中找到需要的 gadget,此时可以得到脚本文件名及编译器代码;
2) 为了得到更多信息,可以根据当前 gadget 所在的节名找到 dump 文件;
3) 进而在 dump 文件中得到脚本文件名及起始行号.

依据这些信息,我们可以由需要注入的 gadget 反推得到所要注入的脚本代码片段以及需要触发的即时编译器.

3.4 Gadget的查找

我们以 dump 文件为单位来查找 gadget.我们用 python 语言开发了一个经典 gadget 查找系统.该系统的架构与 pyrop[24]类似,但有自己的核心代码,包括 gadget 分类器、约束检测、if-gadget 搜索.和 pyrop 一样,我们使用最弱前置条件来对经典 gadget 进行分类:gadget 除了满足语义定义外,还要满足内存访问约束.我们为每一个

以间接控制流指令结束的代码片段生成一个可能的 gadget 集合;然后具体执行该代码片段若干次,每次具体执行都通过随机数来初始化每个使用到的寄存器和内存读操作;输出寄存器和内存写用来评估前置条件的真假.在多次具体执行中,一直存在的 gadget 则被视为可复用的 gadget.

为了查找 *lahfG*,*pushfG* 和 *negG* 这 3 类 gadget,我会对 dump 文件中的机器码进反汇编,并在其中搜寻可能的 gadget.为了保证每个找到的 gadget 有可控制的副作用,它们需要满足如下约束条件:(1) 以 *lahf*(*popf* 或 *neg*)指令开始;(2) 以 *ret* 指令结束;(3) 中间无任何控制流指令,且无内存读写操作.

为了确保找到的 gadget 不是因为显式常数和隐式常数所引入的,同时不受常数致盲和地址空间布局随机化影响,我们会在字节码和机器代码中找出所有可能的常数,然后按照第 2.3.2 节所述规则对 gadget 进行过滤.具体地,会采用如下过程来查找常数.

1. 脚本中的显式常数:我们会在 dump 文件的脚本字节码中找出可能的常数.在 SpiderMonkey 生成的字节码中,常数紧跟在关键字“int”或“uint”之后;在 GoogleV8 生成的字节码中,常数紧跟在关键字“LdaConstant”或“LdaSmi.Wide”之后.我们可以根据这些关键字信息找到可能的常数.
2. 动态代码中地址相关常数的出现:在 x86-64 架构下,动态代码通常用指令 *movabs* 来操作内存地址值(e.g. *movabs r*11,0x7ffff7e7d440).因此,如果这种指令有一个立即数操作数,则很可能是将一个地址值加载到寄存器中或存储到内存中.在我们的分析系统中,*movabs* 指令的立即数操作数会被认为是受地址空间布局随机化影响的常数;但在 SpiderMonkey 中有一个例外情况,如果立即数的前缀是 0xfff88000,则被认为是显式常数的出现.
3. 动态代码中显式常数的出现:除 *movabs* 指令外的其他指令,如果有两个显式操作数且其中一个为立即数,则会认为该立即数是显式常数的可能出现(GoogleV8 中的脚本常数就是以这种形式出现在动态代码中).
4. 动态代码中的隐式常数:在动态代码中有很多直接控制流转移指令(包括条件转移指令).Maisuradze 等人[15]证实,这些指令的操作数可用于注入 gadget.在这些常数上找到的 gadget 会被滤除.

4 实验评估

我们在 x86-64 架构上观察了 SpiderMonkey 和 GoogleV8 这两个即时编译引擎生成的动态代码.以 Octane 等基准测试集中的 JavaScript 脚本作为输入,可以得到较丰富的动态代码块.在这些动态代码中,可以找到大量的 gadget.我们评估利用这些 gadget 帮助 ROP 代码实现图灵完备计算的可行性.

4.1 实验用例与参数设置

我们在 x86-64 硬件架构上剖析了 SpiderMonkey 和 GoogleV8 这两个开源即时编译引擎的架构.它们的概要信息罗列在表 7 中.这两个引擎都很复杂,都拥有超过 50 万行的 C/C++代码(包括头文件).这两个引擎都在动态生成的代码中施加了一些安全机制,包括栈对齐检测和栈溢出检测.GoogleV8 给动态代码施加的安全机制多一些,包括常数致盲和独立于系统的地址空间布局随机化.

Table 7 A summary of information of evaluated JIT engines

表 7 即时编译引擎的概要信息

引擎名	版本号	C/C++代码量(万行)	包含的编译器开发代号	引擎给动态代码施加的安全机制
SpiderMonkey	v45	50.7	*Baseline, Ion*	栈对齐检测,栈溢出检测
GoogleV8	v6.0.99	77.9	*Full-codegen, Cranshaft, Turbofan*	栈对齐检测,栈溢出检测, 常数致盲,地址空间布局随机化

我们以 Octane、Sunspider 等常见的基准测试集中的 JavaScript 脚本作为引擎的输入.通常,这些基准测试集是用于测试浏览器(主要是即时编译引擎)的性能.本工作使用它们的目的是为了得到较为丰富的动态代码块.这些基准测试集提供了可观的脚本代码量.如表 8 所示,其中 Octane 中就有 36.3 万行 JavaScript 代码.当我们调用 js 和 d8 这两个宿主程序运行这些脚本时,采用默认选项.这意味着在 SpiderMonkey 中,当一个脚本函数(或

函数中的一个循环)被调用 10 次后,该函数会被 *Baseline* 编译器编译;当被调用超过 1 100 次后,会被 *Ion* 编译器再编译一次.GoogleV8 中的情形类似.

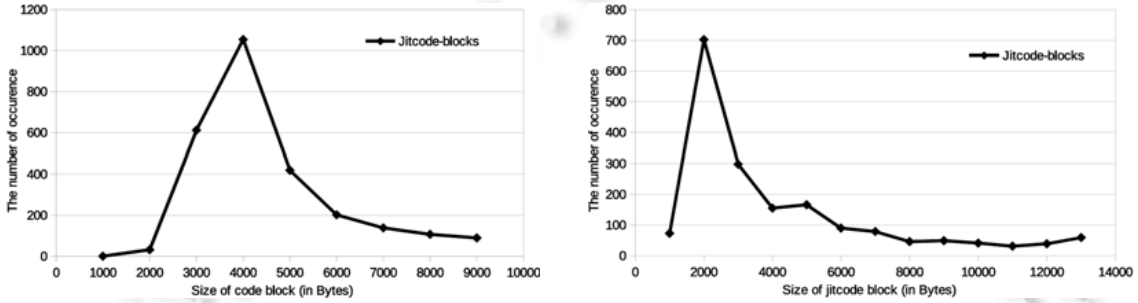
Table 8 A summary of information of JavaScripts fed to JIT engines

表 8 输入的 Javasript 的概要信息

JavaScript 基准测试集	脚本文件数量	代码量(万行)
Octane	15	36.3
Sunspider	26	0.5
jetstream	10	19.7
Kranken	14	7.0

4.2 动态代码块统计信息

在两个即时编译引擎上,可以分别得到 8 422 个和 10 846 个动态代码块.我们统计了这些代码块大小的分布情况.SpiderMonkey 所生成的动态代码的分布如图 3(a)所示,这些代码块集中分布在 1kb~10kb 区间,中心点约在 4kb 附近.GoogleV8 生成的动态代码的分布如图 3(b)所示,这些代码中集中分布在 1kb~12kb 区间,峰值约在 2kb 附近.参考到在 Ubuntu 下用 gcc 的默认编译选项生成的“helloworld”应用程序的代码段只有 1.8kb,可知这些动态代码块的大小都不是很大.这些统计结果显示,将大量动态代码块存储下来并不会消耗太多的磁盘空间.如后文第 5.2 节所讨论的,这有利于构造一个面向脚本代码块的 gadget 挖掘系统.



(a) SpiderMonkey 产生的动态代码块 (b) GoogleV8 产生的动态代码块

Fig.3 Distribution of JIT code blocks in terms of block size

图 3 动态代码块的块尺寸分布

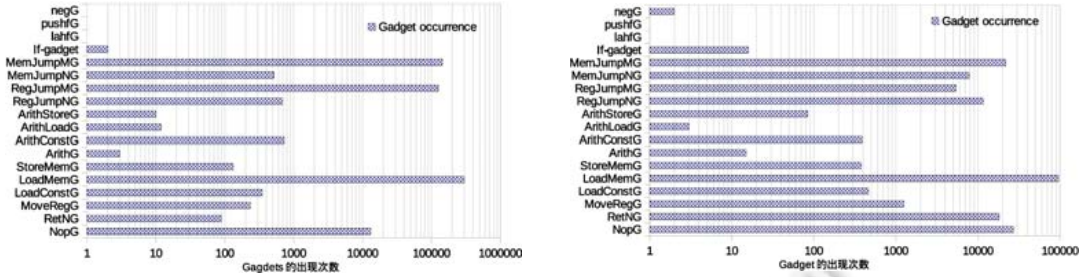
4.3 Gadget的统计信息

在得到的动态代码块中,我们查找第 2.3.1 节定义的各类 gadget 的出现.图 4(a)和图 4(b)分别给出了在 SpiderMonkey 和 GoogleV8 的生成的动态代码中找到的 gadget.由图可见,有些类型的 gadget 出现非常频繁,有些则出现很少,甚至不出现.在两个子图中,*MemJump*、*RegJump* 这两类 gadget 都出现得非常多.这是因为动态代码中经常使用“call*%rax”和“call*48(%rbx)”这样的指令来调用辅助函数、C++类的方法或其他动态代码块.*LoadMemG* 的也出现得非常多,这是因为动态代码采用保守的方案来保存寄存器的状态.在进入函数体前,会存储通用寄存器和多媒体寄存器的内容;相应地,在退出函数体前,会执行大量“*pop reg*”指令来还寄存器的原状态.从 ROP 攻击的角度来看,“*pop* reg_1;*pop* reg_2;*ret*”这样的指令序列可用于加载内存数据到寄存器,因此会被认为是可复用的 *LoadMem* 类型的 gadget.

从图 4 中也可以看到,*pushfG* 和 *lahfG* 在动态代码中没有出现,而 *negG* 只在 GoogleV8 生成的动态代码中出现 1 次(为了让它能够在对数坐标轴上表示出来,将其出现改为 2 次).

由此可见,由于缺少副作用较小的 gadget 来转移标识寄存器 EFLAGS 的值,用传统的 Shacham 方法实现条件转移会比较困难.需要强调的是,该统计结果只是表明,由于输入脚本的局限性,在得到的动态代码中没有找到某些类型的 gadget;但这并不意味着在其他脚本编译后得到的动态代码中必定不存在这些 gadget.我们会在

第 5.2 节讨论相关问题.



(a) SpiderMonkey 产生的动态代码块 (b) GoogleV8 产生的动态代码块

Fig.4 Occurence of gadgets in JIT code blocks

图 4 出现在动态代码块中的 gadget

尽管如此,if-gadget 在两个引擎生成的动态代码中都有出现.因此,用 if-gadget 实现 ROP 代码的条件转移是可行的.我们观察了在 SpiderMonkey 生成的动态代码中得到的那个 if-gadget(为了让它能够在对数坐标轴上表示出来,将其出现改为 2 次),它是 jetstream 测试集中的 container.cpp.js 脚本程序编译后得到的.图 5 的左侧展示了该 if-gadget 的细节信息,该 if-gadget 是由条件转移指令 *jno* 开始的,它的假分支(pass-through 分支)是“*pop rbx*; *ret*”,这是一个 *LoadMem* 类型的经典 gadget;它的真分支跳转到一条 *jne* 指令(偏移+*a* 处).*jne* 指令的假分支是“*ret*”,这是 NOP 类型的经典 gadget.这两个 gadget 会从不同的内存单元选择下一个 gadget,因此可以帮助 ROP 代码实现条件转移.当攻击者在利用该 if-gadget 时,需要让 *jne* 的条件恒为假(即标志 *ZF*==1).这样攻击者可以控制 *jno* 所检测的标志位 *OF* 来有条件地选择下一个 gadget.

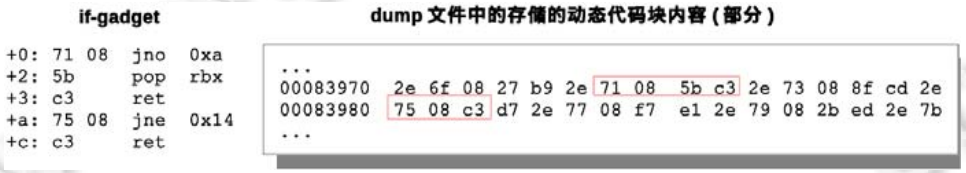


Fig.5 An occurence of if-gadget in JIT code blocks

图 5 出现在动态代码块中的 if-gadget

依据第 2.4 节制定的图灵完备 ROP 代码的构造方案,我们发现,在 SpideMonkey 和 GoogleV8 生成的动态代码中都能够找到图灵完备的 gadget 集合.其中,

- *ArithG*、*ArithLoadG*、*ArithStoreG*、*ArithConstG* 这几类 gadget 在两份动态代码中都有出现,因此可以帮助 ROP 代码实现“减法运算”和“比较运算”这两种功能;
- *StoreMemG*,*ArithStoreG* 都有出现,可以帮助 ROP 代码实现“写内存操作”的功能;
- *LoadMemG*,*ArithLoadG* 也都有出现,可以帮助 ROP 代码实现“读内存操作”的功能;
- if-gadget 的出现,可以帮 ROP 代码实现条件转移功能.

4.4 面向脚本代码块的Gadget注入实例

我们以构造图 1 所示的 ROP 代码为例,描述面向脚本代码块的 gadget 注入过程.在这段 ROP 代码中,字符串常量“/bin/bash”可以由脚本提供,让浏览器将它写入到数据段.攻击者的一个主要工作是注入“*pop %rdi*; *ret*”这个 gadget.我们以 SpiderMonkey 下的注入为例,描述注入过程.

在 SpiderMonkey 生成的动态代码中,这个 gadget 有多次出现,其中一次出现,是在 Octane 测试集的 box2d.js 脚本编译后得到的,具体是在 dumpf27c78fd3626cb1226d0dfcbc2e3e6e1e92b902f70a991d7c0e2833f409e43f400

文件中出现的.根据该 dump 文件,可以得到脚本文件名和被编译脚本的起始行号分别为“box2d.js”和“2528”.在脚本文件中,我们找到了该脚本函数,见表 9.在进行注入攻击时,攻击者可以将该函数填充到攻击脚本中,然后按照 Athanasakis 等人[14]所述的方法找到该 gadget 来构造 ROP 代码.

Table 9 A JavaScript function, we can find a “*pop %rdi*; *ret*” gadget in its JIT’ed code

表 9 编译后在动态代码中可以找到“*pop %rdi*; *ret*”的脚本函数

行号	JavaScript 代码
2528	*o.TimeOfImpact=function(b)* {
2529	*++o.b2_toiCalls*;
…	…
2607	**return** *v*;
2608	};

调用库函数 system(或 exit)的 gadget 可以在静态代码中查找.浏览器应用程序 Firefox 和其动态链接库 libxul.so 的过程链接表中可能存在该条目.如果不存在,则可以调用 Firefox 中封装该功能的内部函数.另一种方法是使用包含指令 *syscall* 的 gadget 或调用“*syscall*”库函数.在这种方法中,则需要设置 *rax* 寄存器的值来指定系统功能号,这可以通过“*pop rax*; *ret*”这样的 gadget 来实现.在动态代码中这个 gadget 有大量出现,其中一个是出现在脚本文件 container.cpp.js 的 *main* 函数所对应的动态代码中.

5 讨 论

通过实验评估可知,面向脚本代码块的注入方法可以向浏览器的进程空间注入 gadget,并帮助 ROP 代码实现图灵完备的功能.本节讨论构造攻击脚本的相关问题和可能的防御方法.

5.1 获取更丰富的Gadget类型

我们在第 4.3 节的评估实验证实了面向脚本代码块的 gadget 注入可以得到图灵完备的 ROP 代码.该实验同时也表明,有些类型的 gadget 的出现很少.这个现象在一定层度上与输入的脚本代码有关.为了得到更丰富的 gadget 类型,一种简单可行的方法是输入更丰富的脚本代码.为此,可以收集更多的 JavaScript 脚本作为输入,或者构造一个编译工具来自动化地生成大量 JavaScript 脚本.

5.2 脚本代码块的粒度

即时编译引擎是以脚本函数为单位来编译生成动态代码的,因此,攻击者需要以脚本函数为粒度来汇集代码块来构造攻击脚本.如果一个脚本函数被转化后可以注入 gadget,且它不会调用其他函数,那么我们称这种函数为叶子函数.攻击者在攻击脚本的主函数中设置好叶子函数被调用次数之后,即可触发相应的即时编译器来生成动态代码即注入期望的 gadget.因此,使用叶子函数可以简化攻击脚本的构造.另外,使用代码行数少的短小脚本函数有利于减小攻击脚本的规模.在后续的工作中,我们会探索可行的方法来寻找短小的叶子函数.

寻找短小叶子函数的一种可行方法是采用模糊测试的思想.

1) 首先创建一个变异工具来自动化地产生 JavaScript 脚本函数,并利用该工具生成大量的叶子函数;
2) 然后将汇集这些函数的脚本文件作为即时编译引擎的输入,来得到动态代码块;
3) 与本文的分析系统类似,在动态代码中查找 gadget,并将其与对应的脚本函数关联;
4) 在实际攻击中,选择找到的短小脚本函数来得到期望注入的 gadget.

由第 4.2 节的统计结果可知,存储动态代码所需要的磁盘空间并不大.现在的普通个人电脑也有上 Tb 的磁盘空间,在其上可以存储上百万个动态代码块的信息.攻击者可以用这个方法来挖掘可能的短小叶子函数.

5.3 可能的防御方案

面向脚本代码块的注入方法所能得到的 gadget 与即时编译器生成动态代码的算法非常相关,因此,那些面向脚本数据和脚本结构的防御机制并不能有效地遏制这种注入攻击.有两种可能的防御方案,它们在指令级别对动态代码进行保护.

- 一种方法是采用 gadget-free[11,27]的思想,通过二进制重写,保护动态代码中的合法控制流指令免于滥用,同时消除截断指令形成的非法间接控制流指令.这种方法是有效的,但是动态修改二进制代码会招致可观的性能开销.如果这种方法招致的性能开销会抵消引入即时编译机制带来的性能提升,那么该方法是不能接受的.因此,防御者需要找到一种针对动态代码的轻量级的 gadget-free 实现方案.
- 另一种可行的方法是在指令级别做细粒度的动态随机化.具体来说,在每一次生成动态代码时,随机地改变每一个指令的编码和长度,让攻击者难以得到需要的 gadget.在 x86-64 架构上的一种可能实现是实施动态寄存器分配随机化.x86-64 架构上,改变一条指令所使用的寄存器不仅能够改变指令的编码,而且可能改变指令的长度.

6 相关工作

本节先概述 ROP 相关的攻击和防御,再概述与即时编译机制相关的攻击和防御,最后概述与图灵完备 gadget 相关的工作.

6.1 ROP攻击与防御

近 10 年来,ROP 相关的攻击和防御技术一直成为研究的热点.ROP 通过复用目标进程空间中的已有代码进行攻击,而不需要注入外部代码,因此能够有效地绕过 DEP[2,3]等安全机制.ROP 攻击可以在众多的硬件架构上进行,包括 x86[4]、ARM[28]和 SPARC[29].因此,ROP 对已有计算机系统造成严重威胁.

地址空间布局随机化(ASLR)是缓解 ROP 攻击的有效方法[5,30−32].通过将代码模块加载到一个随机选择的地址空间,让代码所在的地址拥有不确定性,ASLR 能够显著增加 ROP 攻击的难度.但是,如果受害进程中存在信息泄露漏洞时,攻击者可以很容易让 ASLR 失效.以 JIT-ROP 为例,它通过反复利用一个信息泄露漏洞来暴露浏览器程序的已加载代码,并通过动态地收集 gadget 并构造 ROP 攻击代码[6].JIT-ROP 能有效地绕过 ASLR.

防范信息泄露是缓解 JIT-ROP 攻击的有效方法.近年来提出的方法通过去除代码页面的可读属性来防范信息泄露[7−10].另外,gadget-free 安全机制是根除 ROP 攻击的有效方法.G-free[11]通过消除所有非法的间接控制流指令(从合法指令的中间开始的指令),同时保护合法的控制流指令免于滥用来实施该机制.ret-less[27]通过消除内核中所有可能的 *ret* 指令来缓解 ROP 攻击.这两个工作都能很好地从根源上缓解 ROP 攻击,但它们都需要重新编译源代码.

6.2 即时编译机制滥用与防范

浏览器中引入的即时编译机制(just-in-time compilation)为攻击者注入恶意代码提供了便利.2010 年,Blazakis[12]提出并构建了 JIT 喷洒攻击.该攻击利用即时编译机制将同一份 shellcode 的大量拷贝喷洒到代码缓存区中.由于代码缓存区所在页面具有可写和可执行属性,因此该攻击可以绕过 DEP;同时,由于喷洒攻击可以让 shellcode 填充到可预见的地址区间,因此它也可以绕过 ASLR.随后,Song 等人[33]利用代码缓存区的可写可执行属性在多线程环境下实现了传统的 shellcode 注入攻击.现代浏览器采取两个措施来缓解这种攻击:(1) 将动态生成的代码和数据分割在不同的内存页上,并在数据页上实施 DEP[34];(2) 限制每一片代码缓存区的大小,并限制总量.同时,Zhang 等人[13]提出了一个在动态代码页上严格实施 DEP 的方案,能够强力地抵抗这种攻击.该方案把代码缓存区的可写权限分配给一个独立的外部进程,让浏览器对该区域只有可执行权限,这样攻击者不能以输入数据的方式向代码缓存区注入 shellcode.

尽管攻击者不再能够注入 shellcode,但是利用即时编译机制注入 ROP gadget 却依然可行.由于即时编译器会把脚本中的整型常数放置到代码缓存区,因此攻击者可以通过控制脚本常数来注入自定义的 gadget.这种注入攻击可以通过常数去毒化处理来防范.Wu 等人[34]在 RIM 中通过对包含有常量的指令进行重写,使攻击者不能在代码缓存区得到期望的 gadget.Wei 等人[35]提出了 INSeRT,不仅移除了指令中原来的常量,同时对寄存器以及指令的顺序也进行了随机化来防范这种攻击.Homescu 等人[16]提出了 *librando*,通过常数致盲来对脚本常数进行去毒化处理,同时在适当的位置随机地插入 *NOP* 指令来改变指令的地址.

但是,完全消除基于即时编译机制的 ROP gadget 注入并不是那么容易.Athanasakis 等人[14]的工作证实了利用 2 字节的脚本常量也能注入图灵完备的 gadget,但对 2 字节以下(包含)的脚本常数做去毒化处理所招致的性能开销会很高.Maisuradze 等人[15]的工作证实,动态代码中的隐式常量(即直接控制流指令的操作数)也可以被控制并用来注入 gadget.Maisuradze 等人[17]发现,GoogleV8 和 Chakra 即时编译引擎中的的内联优化机制会将致盲的常数恢复成明文,为注入 ROP gadget 提供了可能.为了防范这种逃逸案例,他们为即时编译器增加了一个预处理系统,通过对原脚本代码进行多态变形处理来干扰内联优化产生的结果,直至得到的脚本变种不会导致逃逸.

6.3 图灵完备的Gadget

作为构造功能代码的技术,ROP 在理论上是图灵完备的[4],在实际运用中也存在灵活性.Shacham 等人在 libc 上构造了图灵完备的 gadget 集合[4].Checkoway 等人[18]发现,x86 和 ARM 上存在一些与 *ret* 行为等价的指令;在实现 ROP 编程时,可用以这些指令结束的代码片段构造功能代码.JOP[19]证实了可用其他间接控制流指令(特别是间接 *jump* 指令)结束的代码片段来构造攻击代码,且 JOP 与 ROP 有同样的表达能力.Microgadgets[25]则用短小的代码片段(只有 2 字节~3 字节)在 x86 架构上构造图灵完备的 gadget 集合.Hu 等人[36]证实了通过修改非控制流数据也可以串接 gadget,并实现图灵完备的计算.另外,我们已有的工作证实了图灵完备的 gadget 是普遍可构造的,在代码量较小的日常可执行文件中,也存在图灵完备 gadget 集合[37].

7 总结与结论

尽管已有的防御技术使得基于漏洞利用的攻击变得困难,但是网页浏览器提供的脚本执行环境为攻击者发动 JIT-ROP 攻击提供了可能.同时,当代浏览器引入的即时编译机制为攻击者注入 ROP gadget 提供了便利.

本文提出了一种新的基于即时编译机制的 gadget 注入方法,称为面向脚本代码块的 gadget 注入.我们发现,在即时编译器为给定脚本生成的动态代码中,存在固定不变的机器指令序列,这些序列的存在不受常数致盲和地址空间布局随机化等安全机制的影响.在发动攻击时,攻击者可以将特定脚本代码片段填充到攻击脚本中,以便注入期望的 gadget 来构造 ROP 代码.我们在 x86-64 架构上评估了这种注入方法在 SpiderMoneky 和 GoogleV8 这两个主流开源引擎上的可行性.

面向脚本代码块的注入方法所能得到的 gadget 与即时编译器的动态代码生成算法密切相关.这一特性使得对这种注入方法的防范变得困难,当前实施的那些针对脚本数据和脚本结构的防御方案不能消除这种注入攻击.一种可行的防御方案是在动态代码上实施轻量级的 gadget-free 保护机制;另一种方案是实施指令级别的细粒度随机化.我们将在后续工作中探索这两种保护方案的实用性.

References:

[1] Dang TH, Maniatis P, Wagner D. The performance cost of shadow stacks and stack canaries. In: Proc. of the ASIACCS 2015. 2015.

[2] Team P. PaX non-executable pages design & implementation. 2015. http://pax.grsecurity.net/docs/pageexec.txt

[3] Andersen S, Abella V. Data execution prevention: Changes to functionality in Microsoft windows XP service pack 2, part 3: Memory protection technologies. 2005. http://technet.microsoft.com/en-us/library/bb457151.aspx

[4] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. of the CCS 2007. ACM, 2007. 552−561. http://doi.acm.org/10.1145/1315245.1315313

[5] Team P. PaX address space layout randomization (ASLR). 2013. http://pax.grsecurity.net/docs/aslr.txt

[6] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi AR. Justin-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the SP 2013. 2013. 110−121.

[7] Backes M, Nürnberger S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proc. of the USENIX Sec 2014. 2014. 433−447.

[8] Gionta J, Enck W, Ning P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In: Proc. of the CODASPY 2015. ACM, 2015. 325−336. http://doi.acm.org/10.1145/2699026.269910719

[9] Crane S, Liebchen C, Homescu A, Davi L, Larsen P, Sadeghi AR, Brunthaler S, Franz M. Readactor: Practical code randomization resilient to memory disclosure. In: Proc. of the SP 2015. 2015. 763−780.

[10] Tang A, Sethumadhavan S, Stolfo S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In: Proc. of the CCS 2015. ACM, 2015. 256−267. http://doi.acm.org/10.1145/2810103.2813685

[11] Onarlioglu K, Bilge L, Lanzi A, Balzarotti D, Kirda E. G-free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of the ACSAC 2010. 2010. 49−58. http://dl.acm.org/citation.cfm?id=1920269

[12] Blazakis D. Interpreter exploitation. In: Proc. of the WOOT 2010. 2010. 130−144.

[13] Zhang C, Niknami M, Chen KZ, Song C, Chen Z, Song D. JITScope: Protecting Web users from control-flow hijacking attacks. In: Proc. of the INFOCOM 2015. IEEE, 2015. 567−575.

[14] Athanasakis M, Athanasopoulos E, Polychronakis M, Portokalidis G, Ioannidis S. The devil is in the constants: Bypassing defenses in browser JIT engines. In: Proc. of the NDSS 2015. 2015. 25−37.

[15] Maisuradze G, Backes M, Rossow C. What cannot be read, cannot be leveraged? Revisiting assumptions of JIT-ROP defenses. In: Proc. of the USENIX Sec 2016. 2016. 139−156.

[16] Homescu A, Brunthaler S, Larsen P, Franz M. Librando: Transparent code randomization for just-in-time compilers. In: Proc. of the CCS 2013. ACM, 2013. 993−1004. http://doi.acm.org/10.1145/2508859.2516675

[17] Maisuradze G, Backes M, Rossow C. Dachshund: Digging for and securing against (non-) blinded constants in JIT code. In: Proc. of the NDSS 2017. 2017. 133−147.

[18] Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-oriented programming without returns. In: Proc. of the CCS 2010. ACM, 2010. 89−104.

[19] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-oriented programming: A new class of code-reuse attack. In: Proc. of the ASIACCS 2011. ACM, 2011. 30−40. http://doi.acm.org/10.1145/1966913.1966919

[20] Nurnberg PJ, Wiil UK, Hicks DL. A grand unified theory for structural computing. In: Hicks DL, ed. Proc. of the Metainformatics. Berlin, Heidelberg: Springer-Verlag, 2000. 1−16.

[21] Cowan C, Pu C, Maier D, Hintony H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the USENIX Sec'98. USENIX Association, 1998. 22−35.

[22] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow integrity. In: Proc. of the CCS 2005. ACM, 2005. 340−353. http://doi.acm.org/10.1145/1102120.1102165

[23] Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. In: Proc. of the USENIX Sec 2011. 2011. 201−216.

[24] Stewart J Dedhia V. ROP compiler. 2015. http://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf

[25] Homescu A, Stewart M, Larsen P, Brunthaler S, Franz M. Microgadgets: Size does matter in turing-complete return-oriented programming. In: Proc. of the WOOT 2012. 2012. 64−76.

[26] Yuan PH, Zeng QK. Universal availability of ROP-based turing-complete computation. Ruan Jian Xue Bao/Journal of Software, 2017,28(10):2583−2598 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5317.htm [doi: 10.13328/j.cnki.jos.005317]

[27] Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented rootkits with return-less kernels. In: Proc. of the 5th European Conf. on Computer Systems. 2000. 195−208.

[28] Kornau T. Return oriented programming for the ARM architecture. 2014. http://www.zynamics.com/downloads/kornau-tim-diplomarbeit-rop.pdf

[29] Buchanan E, Roemer R, Shacham H, Savage S. When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proc. of the CCS 2008. ACM, 2008. 27−38. http://doi.acm.org/10.1145/1455770.1455776

[30] Kil C, Jun J, Bookholt C, Xu J, Ning P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In: Proc. of the Computer Security Applications Conf. IEEE Computer Society, 2000. 339−348.

[31] Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson J. ILR: Where'd my gadgets go? In: Proc. of the SP 2012. 2012. 571−585.

[32] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of the CCS 2012. ACM, 2012. 157−168. http://doi.acm.org/10.1145/2382196.2382216

[33] Song C, Zhang C, Wang T, Lee W, Melski D. Exploiting and protecting dynamic code generation. In: Proc. of the NDSS 2015. 2015. 90−108.

[34] Chen P, Fang Y, Mao B, Xie L. JITDefender: A defense against JIT spraying attacks. In: Proc. of the Future Challenges in Security and Privacy for Academia and Industry. Springer-Verlag, 2000. 142−153.

[35] Wu R, Chen P, Mao B, Xie L. RIM: A method to defend from JIT spraying attack. IEEE, 2000. 143−148. http://ieeexplore.ieee.org/document/6329174/

[36] Wei T, Wang T, Duan L, Luo J. INSeRT: Protect dynamic code generation against spraying. In: Proc. of the ICIST 2011. IEEE, 2011. 323−328.

[37] Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z. Data-oriented programming: On the expressiveness of non-control data attacks. In: Proc. of the SP 2016. 2016. 110−125.

附中文参考文献:

[26] 袁平海,曾庆凯.ROP 图灵完备的普遍可实现性.软件学报,2017,28(10):2583−2598. http://www.jos.org.cn/1000-9825/5317.htm [doi: 10.13328/j.cnki.jos.005317]

袁平海(1982－),男,江西分宜人,博士,研究员,主要研究领域为软件安全,系统安全,区块链.

张云剑(1991－),男,工程师,主要研究领域为信息安全.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.

刘尧(1985－),男,博士,主要研究领域为信息安全,密码学.