计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.190700109

GDL:一种通用型代码重用攻击 gadget 描述语言

蒋 楚 王永杰
国防科技大学电子对抗学院 合肥 230037
(1532173962@qq.com)

摘 要 由于代码重用攻击的方式方法多样,相应的 gadget 在结构上也不尽相同,目前尚没有一种通用的方法能够用来描述多种代码重用攻击下的 gadget。结合几种常见代码重用攻击的攻击模型和图灵机模型,文中提出了一种代码重用攻击的通用模型,为了能够对代码重用攻击中的 gadget 进行结构化的描述,设计了一种用于代码重用攻击的 gadget 描述语言(Gadget Description Language,GDL)。首先,介绍了代码重用攻击的发展历程,总结了代码重用攻击的攻击模型和 gadget 特征;然后,以此为基础设计了 GDL,给出了 GDL 中的关键字和各种约束类型的语法规范;最后,在 ply 和 BARF 等开源项目的基础上,实现了基于 GDL 的 gadget 搜索原型系统 GDLgadget,并描述了 GDLgadget 的执行流程,通过实验验证了 GDLgadget 的可用性。

关键词:代码重用攻击;攻击模型;图灵机模型;gadget 描述;gadget 搜索
中图法分类号 TP309

GDL:A Gadget Description Language for General Code Reuse Attack

JIANG Chu and WANG Yong-jie
College of Electronic Engineering,National University of Defense Technology,Hefei 230037,China

Abstract Considering code reuse attacks have various types,and the corresponding gadgets are different in structure,there is no general method to describe gadgets under multiple code reuse attacks. Combining several common attack models of code reuse attack and Turing machine,this paper proposes a general model of code reuse attack. A gadget description language(GDL) for code reuse attack is designed to describe the gadget in code reuse attack structurally. Firstly,the development history of code reuse attack is introduced,and the attack model and gadget characteristics of code reuse attack are summarized. Secondly,GDL is designed and the key words and grammatical specifications of various constraint types in GDL are given. Finally,on the basis of open-source project such as ply and BARF,the gadget searching prototype system named GDLgadget is implemented,which is based on GDL. The execution process of GDLgadget is described,and the effectiveness of GDLgadget is verified in experiments.

Keywords Code reuse attack,Attack model,Turing machine,gadget description,gadget discovery

1 引言

内存漏洞是当前网络空间安全领域中面临的主要问题之一。近年来,利用内存漏洞对软件进行攻击的案例数不胜数,极大地损害了人们的利益,严重干扰了社会秩序,并给网络空间安全乃至国家安全带来了巨大的威胁。

自 1988 年第一例缓冲区溢出漏洞攻击——莫里斯蠕虫(Morris worm)之后,针对内存漏洞的攻击方法与相应的防护手段便层出不穷。渗透测试人员利用溢出漏洞向内存中注入恶意代码并篡改保存在内存中的跳转地址,使程序的控制流转向注入的恶意代码中,这就是早期的代码注入攻击(Code Inject Attack)。与此同时,计算机系统安全机制的设计者也在不断地完善安全机制,以缓解甚至避免内存漏洞被渗透测试人员利用。数据执行保护(Data Execution Prevention,DEP)将内存中的数据和代码严格区分开,使得渗透测试人员注入的恶意代码只能被当作数据解析而无法被执行,这使得常规代码注入攻击难以发挥作用。

然而,攻防技术相生相长,另一种攻击方法慢慢兴起。渗透测试人员利用内存漏洞改变程序的执行流程,通过把内存空间中已有的、分散的代码片段链接起来,构造出能实现特定功能的程序逻辑来实现攻击目的,这就是代码重用攻击(Code Reuse Attack,CRA),它能够达到与传统的代码注入攻击相同的攻击效果。针对代码重用攻击,学术界已经提出了许多防御策略和方法,但由于性能、安全性和兼容性等方面的制约,各种防御机制并不能达到令人满意的效果,特别是代码重用攻击的各种变异技术被不断提出,当前的防御机制很难有效地抵御新型攻击。

本文对代码重用攻击的实现机制和主要攻击方法进行了概述,分析了代码重用攻击中 gadget 的特征和组织结构,并在此基础上设计了一种用于描述代码重用攻击中 gadget 特征的语言——GDL,以便形式化地定义代码重用攻击的攻击方法,以指导针对代码重用攻击的防御机制的设计。

本文第 2 节介绍代码重用攻击的基本概念、主要攻击方法以及防御机制和绕过技术;第 3 节讨论代码重用攻击的特

到稿日期:2019-07-17 返修日期:2019-11-07 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
通信作者:王永杰(w_yont_j@189.cn)

征，结合代码重用攻击的攻击模型以及图灵完备性，提出一种代码重用攻击的通用模型；第 4 节在代码重用攻击通用模型的基础上，设计一种用于描述 gadget 结构的语言 GDL；第 5 节实现原型系统并进行实验；最后总结全文。

2　背景

2.1　代码重用攻击的概念

代码重用攻击是一种利用漏洞改变程序在内存中原有代码执行流程以实现攻击目的的漏洞利用技术。代码注入攻击和代码重用攻击都是对计算机内存漏洞进行利用的攻击方式，两者均会篡改一些影响程序执行的关键数据，如栈中的返回地址、函数指针等，以此实现超出程序设计者预期的功能逻辑。它们的区别在于渗透测试人员执行恶意代码的方法，代码重用攻击不需要向内存中注入外部代码，只需要重用程序中已经存在的指令片段，这些指令片段能够完成一定的计算和赋值等功能，因此也被称为 gadget，通过多个 gadget 的拼接，渗透测试人员可以实现一些恶意目的。由于现代操作系统中已经广泛采用数据不可执行(DEP)等防御机制，代码注入攻击已经难以实现攻击目的，因此代码重用攻击成为当前的主流攻击方式。

2.2　代码重用攻击的发展历程

2.2.1　代码重用攻击的发展历程

早期的漏洞利用过程中，渗透测试人员直接将劫持的控制流重定向到布局在栈中或堆中的外部代码(shellcode)来实现任意代码执行。但在现代操作系统中，内存不可执行(Non-Executable memory，NX)、数据执行保护(Data Execution Prevention，DEP)和动态代码签名(dynamic code-signing)等机制的广泛部署使得渗透测试人员注入到内存中的 shellcode 不具备执行权限，代码注入攻击变得不可行。

于是，攻击的思路转变为执行程序原有的具有可执行权限的代码，这种攻击方法早在 1997 年就被 Solar 提出[1]，若渗透测试人员将控制流的跳转目标指向某函数的入口，并已经在栈中布局好该函数的调用参数，就可以以自定义的参数完成对该函数的调用。

2007 年，Shacham 提出了一种名为 ROP(Return-Oriented Programming)[2]的攻击方法，能够完美地解决上述问题。在 ROP 攻击中，渗透测试人员利用 ret 指令，让控制流跳转到来自于渗透测试人员布局在栈上的目标地址，这样只需依据要执行的 gadget 的顺序，将 gadget 的地址和参数进行拼接，构建一条 gadget 链，就能实现内存中分散代码片段的连续调用。ROP 被证明是图灵完备的[2]。

在 ROP 的基础上出现了许多代码重用攻击方法，其中比较有代表性的有 JOP[3]，但真实环境中“pop＋jmp”类型的 gadget 数量很少，因此这种攻击难以在实际中应用，为此，Bletsch 等引入了一种用于调度程序控制流的 gadget，并使用以 jmp 或 call 间接跳转指令结尾的 gadget 来实现代码重用攻击[4]，但 call 指令在执行时会将它的下一条指令压入栈中，这样就使得 gadget 链的构造十分困难。Sadeghi 等通过 strong trampoline gadget 调整堆栈实现 PCOP(Pure-Call Oriented Programming)[5]，能够组合以 call 指令结尾的 gadget 实现攻击。

2.2.2　代码重用攻击技术的改进

随着防护技术的日益发展，现有的攻击手段已经很难突破新的保护机制，如微软的 CFG(Control Flow Guard)[6]和 RFG(Return Flow Guard)，渗透测试人员也在不断发掘新的攻击方法。学术界设计的防御机制通常基于随机化或者基于控制流完整性(Control Flow Integrity，CFI)[7]保护。

代码重用攻击的实现依赖于内存中已有的指令，渗透测试人员需要知道这些指令的确切位置，因此增加渗透测试人员对内存的不可知性很大程度上能够阻止攻击，使渗透测试人员难以构造出可执行的 gadget 链。比较典型的方案有 ILR[8]，Binary Stirring[9]，Smashing the gadget[10]，CodeArmor[11]以及 Oxymoron[12]等，它们通过指令重排、等效指令替换、寄存器重新分配、函数顺序重排、代码块重排、周期性内存分布随机化和去除指令中的地址引用等方法实现保护。

针对基于随机化的保护方案，Snow 提出了 JIT-ROP(Just In Time-ROP)攻击[13]，利用程序中的信息泄漏漏洞来读取内存空间中的数据，定位关键数据结构、模块的位置，动态地搜索 gadget 并构造 ROP 链，使得细粒度地址空间分布随机化(Address Space Layout Randomization，ASLR)[14]以及指令集随机化(Instruction Set Randomization，ISR)失效；PI-ROP (Position Independent ROP attack)[15]针对页内代码数据相对偏移不变的特点，利用内存漏洞修改代码指针，在不泄漏内存信息的情况下绕过基于随机化的防御机制。

通常，代码重用攻击会改变程序设定的执行顺序，因此限制程序的控制流转移均处于事先定义的预期控制流图(Control Flow Graph，CFG)中，能够使渗透测试人员构造的跳出原有程序控制流的 gadget 链不被成功执行。比较典型的方案有 Bin-CFI[16]，CCFIR[17]，TypeArmor[18]以及 FlowGuard[19]等，它们通过二进制改写插入保护段，结合动态改写与 ASLR 进行间接跳转指令的检查，依靠调用前传递的参数个数和类型精确匹配被调用函数，利用处理器中 LBR(Last Branch Record)记录的历史跳转指令进行判断等方法，实现 CFI 保护。

针对基于 CFI 的防御机制，渗透测试人员会利用系统设计上的缺陷实现绕过。最为典型的一种攻击方法是 SROP[20]：一般地，控制流完整性保护不能用于操作系统信号及中断，而在信号或中断发生时，会涉及进程状态的保存和恢复，并且这些操作都会通过栈来进行，因此渗透测试人员可以利用内存漏洞修改这些数据进而劫持控制流；Lan 等提出了 LOP(Loop-Oriented Programming)攻击[21]，这种攻击方法类似于 Return-to-Libc，但它在引入 loop gadget 后能够多次调用函数，由于其 gadget 是完整的函数而非代码片段，因此这种攻击方法可以绕过粗粒度的 CFI 保护；结合 C＋＋语言的特点，Schuster 提出了 COOP(Counterfeit Object-oriented Programming)攻击[22]，篡改 C＋＋对象并调用特定的虚函数来实现自己的攻击意图，这种攻击方法利用了 C＋＋语言的特性，能够绕过大多数通用的安全机制。

2.3　小结

代码重用攻击的方式方法多样，相应的 gadget 在结构上也不尽相同，但是目前尚没有一种通用的方法能够用来搜索多种代码重用攻击下的 gadget。一般地，研究人员在设计出

一种新的代码重用攻击方法后，都需要利用相应的 gadget 搜索工具来实现攻击，短期来说，攻击成本有所提高，很多代码重用攻击的工具并没有开源，渗透测试人员将不得不自己编写并使用不同的代码重用攻击工具；但从长远来说，这大大提高了研究人员评估攻击技术的危害性和防御机制安全性的难度，并不利于网络空间的长治久安。

本文试图解决以下 3 个问题：1）代码重用攻击的特征有哪些；2）gadget 需要满足哪些约束；3）怎样用形式化的方法描述 gadget 的约束以便于搜索。

3 代码重用攻击的特征

尽管代码重用攻击的方法层出不穷，但是它们通常具备两个特征：攻击模型的相似性和图灵完备性。

3.1 代码重用攻击的攻击模型

代码重用攻击的一般流程是：利用内存漏洞将 gagdet 的地址链注入到内存的数据区，然后劫持控制流使得程序依次执行渗透测试人员设定的 gadget，渗透测试人员需要使用某种方式将 gadget 链接起来，并确保它们按照指定的顺序执行。链接 gadget 的过程会使得不同的代码重用攻击方法具有一些相似的攻击模型，本节以 ROP，JOP，LOP 和 COOP 为例进行说明。

在 ROP 攻击模型（见图 1）中，gadget 链的顺序执行依赖于栈空间的布局和返回指令的使用。以 Intel 架构下的程序为例，用于调用函数的 call 指令会将当前指令的下一条指令地址压入栈中，而用于函数返回的 ret 指令会将栈顶处的值作为返回地址。若在函数返回前将下一个 gadget 的地址放到 ret 语句弹出的位置，则它会在执行完当前 gadget 后执行下一个 gadget，这样渗透测试人员就能够将各个 gadget 链接起来。

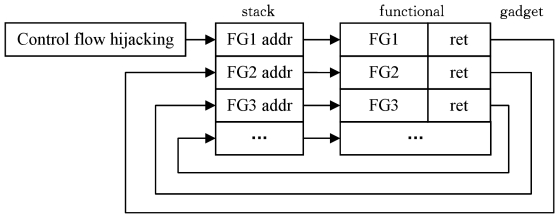

图 1 ROP 攻击模型

Fig. 1 ROP attack model

在 JOP 攻击模型（见图 2）中，gadget 链的顺序执行依赖于一种名为调度 gadget 的特殊 gadget。在 JOP 中，功能 gadget 不再是以 ret 指令结尾，因此不能通过栈布局和 ret 指令结合的方式链接 gadget，渗透测试人员通过引入调度 gadget 作为跳板，使得功能 gadget 能够按照指定的顺序执行。在此过程中，调度 gadget 需要从渗透测试人员可控的目标区域加载地址，功能 gadget 需要在结束时让程序执行流再次转移到调度 gadget 中。

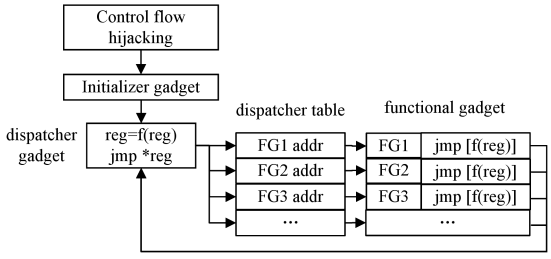

图 2 JOP 攻击模型

Fig. 2 JOP attack model

在 LOP 攻击模型（见图 3）中，gadget 链的顺序执行依赖于一种名为循环 gadget 的特殊 gadget。在 LOP 中，功能 gadget 是一个完整的函数，函数在返回后会跳转到函数调用指令的下一条指令，渗透测试人员通过引入循环 gadget，使得功能 gadget 能够按照指定的顺序执行。为了保证 LOP 的顺利执行，循环 gadget 需要包含一个循环结构，该循环结构能循环加载函数地址并调用函数。

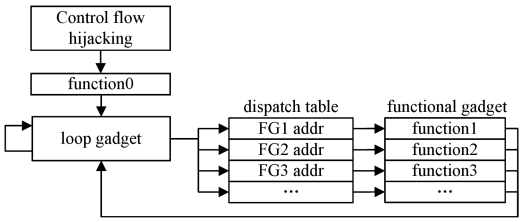

图 3 LOP 攻击模型

Fig. 3 LOP attack model

在 COOP 攻击模型（见图 4）中，修改 C++对象中的成员数据能够实现特定虚函数的调用，整个攻击则是通过修改多个 C++对象中的成员数据来实现的，其主要过程是利用 C++程序中的 Main-loop gadget。Main-loop gadget 在程序中原本的功能就是循环调用多个对象的某一虚函数，渗透测试人员通过篡改这些对象实现 gadget 链的顺序执行。这种攻击方法本质上与 JOP 和 LOP 并无不同，但使用了 C++的语言特性，因此更加隐蔽和难以利用。

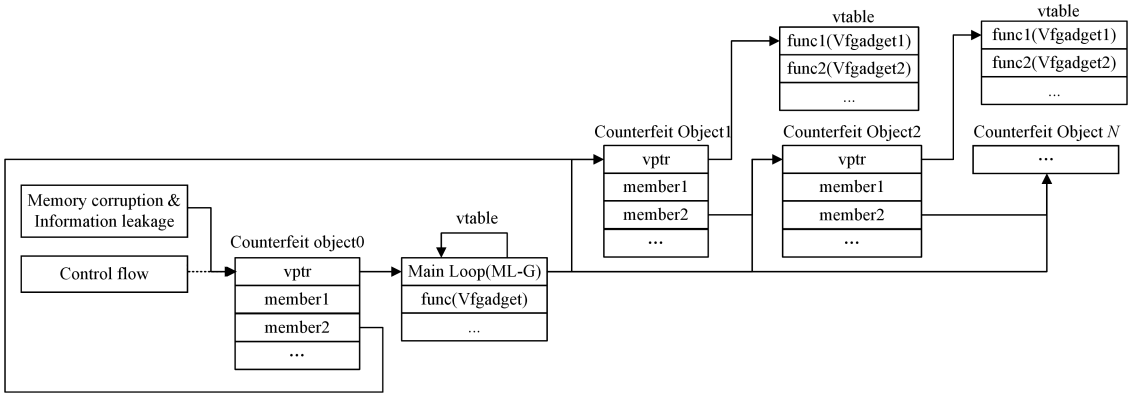

图 4 COOP 攻击模型

Fig. 4 COOP attack model

结合以上攻击模型，可以总结出代码重用攻击的以下特征。

(1)代码重用攻击通常需要一段内存空间用于存储各个 gadget 的地址，一般是渗透测试人员可控的区域。

(2)代码重用攻击通常需要两类 gadget：一类是执行各种操作的功能 gadget，实现具体功能；另一类是作为控制枢纽的调度 gadget，将各功能 gadget 链接到一起。两类 gadget 组合使用以实现代码重用攻击，其中，调度 gadget 需要将存储 gadget 地址的内存空间作为间接跳转的操作数。对于 ROP 而言，两类 gadget 合而为一，gadget 末尾的 ret 指令履行了调度 gadget 的职能。

(3)代码重用攻击是将分散在程序中的代码片段组合起来实现攻击的，因此，gadget 的末尾通常是控制流转移指令。

3.2　代码重用攻击的图灵完备性

图灵机 M 是一个七元组$\langle Q,\Gamma,b,\Sigma,\delta,q_0,F\rangle$，其中，$Q$ 是一个有限非空的状态集；Γ 是一个有限非空的纸带符号集；b 是空符号且满足 $b\in\Gamma$；$\Sigma\subseteq\Gamma\backslash\{b\}$ 是输入符号集，即纸带初始时的符号集；q_0 是起始状态并满足 $q_0\in Q$；F 是最终的接受状态并满足 $F\subseteq Q$；$\delta:(Q\backslash F)\times\Gamma\mapsto Q\times\Gamma\times\{L,R\}$ 是一个转换表，L 和 R 表示纸带移动的方向，对于当前的状态和纸带符号，如果 δ 没有定义如何转换，则机器停止。

开始执行时，M 将输入符号集 Σ 填入纸带，M 处于状态 q_0，并按照转移表 δ 定义的规则进行计算：若当前 M 的状态是 q，纸带上读取的符号为 s，且 $\delta(q,s)=(q',s',L)$，则 M 会进入新状态 q'，并将纸带左移一个格子，这样 M 就能从纸带上读取下一个符号，若 M 在某一时刻进入了状态 F，则会停止执行。

在可计算性理论中，若一系列操作数据的规则能够模拟单带图灵机，那么它就是图灵完备的(Turing complete)。代码重用攻击的最终目的是能够实现任意操作，也就是需要满足图灵完备性(Turing completeness)。证明某种代码重用攻击方法具有图灵完备性只需要证明该方法下的 gadget 能够实现如下功能：算术/逻辑运算、内存装载/存储、控制流转移和函数/系统调用[2]。其中，算术/逻辑运算结合内存装载可以模拟图灵机的查表及状态转移的操作；内存装载/存储可以模拟图灵机在纸带上读写符号的操作；控制流转移可以模拟图灵机移动纸带的操作；函数/系统调用则是一种更为复杂的操作，是以上 3 种操作的组合。

当前主要的代码重用攻击方法下的 gadget 都满足图灵完备性[4,5,13,22-24]，因此代码重用攻击的功能 gadget 可以按图灵完备功能的要求分成 4 类。

(1)算术/逻辑运算

算术运算主要包括加法、减法、乘法和除法，逻辑运算包括逻辑与、或、非和异或。通常，此类 gadget 并不能直接实现渗透测试人员的目的，渗透测试人员需要通过这类 gadget 实现一些操作数或标志位的修改，以便调用其他类型的 gadget。

(2)内存装载/存储

内存的装载/存储主要包括：装载常量到寄存器中；从内存中装载数据到寄存器中；存储数据到内存中。渗透测试人员需要通过这类 gadget 来读写内存，以便调用其他类型的 gadget。

(3)控制流转移

控制流转移可以分为直接跳转和条件跳转，直接跳转的目标地址可能存储在栈上，也可能存储在内存或寄存器中，渗透测试人员需要结合内存漏洞以及内存装载和存储类的 gadget 跳转到特定的位置；条件跳转通常依赖于标志位，因此需要内存漏洞以及前两类 gadget 来实现特定目标的跳转。

(4)函数/系统调用

调用函数的过程通常依赖于栈和寄存器，渗透测试人员需要结合内存漏洞及装载和存储 gadget 调用特定的函数。系统调用在本质上和函数调用并无不同，一般是通过在特定寄存器中设置系统调用号，然后执行系统调用指令来实现。此外，系统调用通常会存在对应的功能函数，调用功能函数即可实现相同的效果。

3.3　代码重用攻击的通用模型

假设某种代码重用攻击方法满足图灵机模型$\langle Q,\Gamma,b,\Sigma,\delta,q_0,F\rangle$，在实施攻击前，图灵机已经处于某个状态，设为 q_t，渗透测试人员需要利用内存漏洞将输入的符号集和转换表加载到内存空间中，并劫持程序控制流到初始化 gadget，通过初始化 gadget 让程序的状态由 q_t 变为 q_0。随后，调度 gadget 和跳板 gadget 模拟图灵机在纸带上读取符号并进行状态转移的过程，这个过程中需要读取转换表所在的内存空间并进行一定的计算，有时还要执行存储操作，程序的状态按照转换表转移后，程序的执行流执行功能 gadget，功能 gadget 在实现功能之后需要进行控制流转移，控制流转移就是模拟纸带移动的过程。如此循环往复，在所有的功能 gadget 执行完后，虚拟图灵机到达 F 状态，攻击完成。

初始化 gadget 实现了图灵机从状态 q_t 变为 q_0 的转换，其具体形式依赖于内存漏洞的类型和利用方式；调度 gadget 要实现读取纸带符号和状态转换的功能，涉及内存装载、算术运算和控制转移多项操作，有时还需要进行内存存储操作，因此多数情况下由多个 gadget 实现，此外，调度 gadget 要使程序控制流转移到功能 gadget，其结尾必定是一条控制流转移指令，通常是 jmp 指令或 call 指令；功能 gadget 要转移回调度 gadget，其结尾必定是一条控制流转移指令，通常是 jmp 指令或者 ret 指令。

综上，设计代码重用攻击的通用模型如图 5 所示，其中，调度 gadget 和跳板 gadget 共同实现了内存装载、算术运算、内存存储和控制转移(包括函数调用)的功能，调度 gadget 以 jmp 指令或 call 指令间接跳转结尾。在 ROP 攻击中，则由 ret 指令同时担任调度 gadget 和跳板 gadget 的角色。

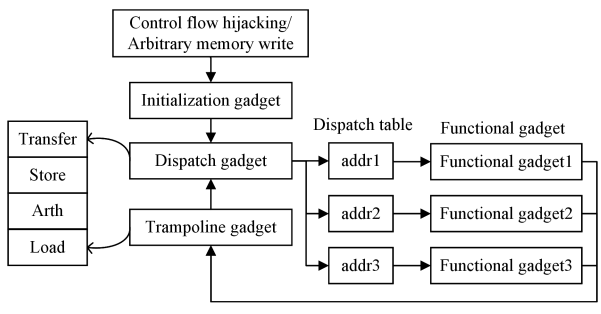

图 5　代码重用攻击的通用模型

Fig. 5　General model of code reuse attack

4 gadget 描述语言

4.1 设计思路

本文设计 GDL 的目的是为安全研究人员提供一种描述代码重用攻击中使用的 gadget 的约束条件的语言，使得研究人员能够以较为简单、较为通用的方式描述 gadget 的特征，并以此为基础搜索 gadget，根据图 4，可以将 gadget 需满足的约束归结为以下几种。

(1)指令约束，对 gadget 使用的指令进行限定，通过限制指令的字节码和助记符来实现。部分代码重用攻击，如 ROP，JOP，其 gadget 以特定的指令结束；而 PIROP 则使用了 x64 指令中 RIP 间接寻址的寻址方式，可以通过指令的字节码找到满足这些约束的指令。

(2)位置约束，对 gadget 的地址进行限定。部分代码重用攻击为了绕过粗粒度 CFI，如 BATE(Back to The Epilogue)[25]，其 gadget 的地址位于特定的内存区间中。

(3)结构约束，指定 gadget 包含特定的控制结构，通过基本块之间的前趋后继关系来判断。部分代码重用攻击，如 LOP，COOP，要求调度 gadget 中包含循环结构。

(4)逻辑约束，指定 gadget 对特定寄存器或内存区间的影响，通过对 gadget 进行语义分析来确定。通常，gadget 会修改或保留部分寄存器和内存空间的值，有时需要利用这些信息来搜索 gadget。

(5)调用约束，对 gadget 的交叉引用情况进行限定。部分代码重用攻击需要利用特定的库函数，如 CFB，其 gadget 的调用和被调用情况可以通过交叉引用信息来确定。

4.2 语言规范

4.2.1 程序结构

GDL 程序主要包含约束定义、搜索目标和注释 3 个部分。

约束定义部分用于对搜索 gadget 过程中涉及的几种主体，如指令、基本块或函数等进行条件上的限定。单个约束的定义以“类型名 对象名 {”表示开始，以“}”表示结束，该部分类似于 C 语言中的结构体声明。对象名可以包含字母、数字和下划线，需要区分大小写。

搜索目标部分用于确定输出的对象。在执行过程中，根据主体之间的依赖关系，所有满足约束定义的主体都会被找出，而用户通常只关心特定的一个或几个主体，这些主体可以通过搜索目标部分得到，搜索的对象是约束定义部分中定义的主体的子集。搜索目标部分通常位于程序的结尾，每一个要输出的对象以“find 类型名 对象名”形式定义。

注释部分是对内容的附加说明，设计时参考了 C 语言的注释规则，单行以“//”标识开始；多行以“/*”标识开始，以“*/”标识结束。

程序主体及搜索目标部分的 BNF 范式定义如下：

⟨program⟩::=⟨definition_list⟩⟨find_list⟩

⟨find_list⟩::={“find”⟨find_target⟩“;”}

⟨find_target⟩::=“pos”⟨pos_name⟩ | “ins”⟨ins_name⟩ | “block”⟨block_name⟩ | “basicblock”⟨basicblock_name⟩ | “bbs”⟨bbs_name⟩ | “function”⟨function_name⟩ | “pos”⟨pos_name⟩

4.2.2 关键字

GDL 需要通过关键字来实现具体约束的定义，关键字包括类型名和约束指令，其中类型名分为 3 种。第一种是变量类型，这种类型的对象不能被输出，但可以为其他类型使用，包括 var，val 和 mr，分别表示变量，变量值和内存/寄存器；第二种是主体类型，这种类型的对象可以被输出，并被其他主体使用，包括 ins，pos，basicblock，block，struct，bbs 和 function，分别表示指令、位置、基本块、代码段、结构、多个基本块和函数；第三种是约束类型，这种类型是为了进一步筛选主体类型的搜索结果，包括 var_exp，logic 和 pos，分别表示条件表达式、逻辑和地址。

不同的类型有不同的约束指令，表示指令约束的 opcode，mnemonic，arg，xref_from 和 xref_to 可以指定指令的字节码、伪助记符和指令的交叉引用情况；表示逻辑约束的 change 和 reserve 可以指定主体修改寄存器的情况；表示结构约束的 has_path 和 nblock 可以指定基本块的前后继关系和基本块数量；表示位置约束的 addr 和 pos 可以用于创建一个地址集合的白名单；表示主体需要满足约束的 startwith，endwith，include，notinclude 和 satisfy 能够用相对简单的主体，如指令来构建较为复杂的主体。

各类型和约束指令的对应关系与依赖关系如图 6 所示。

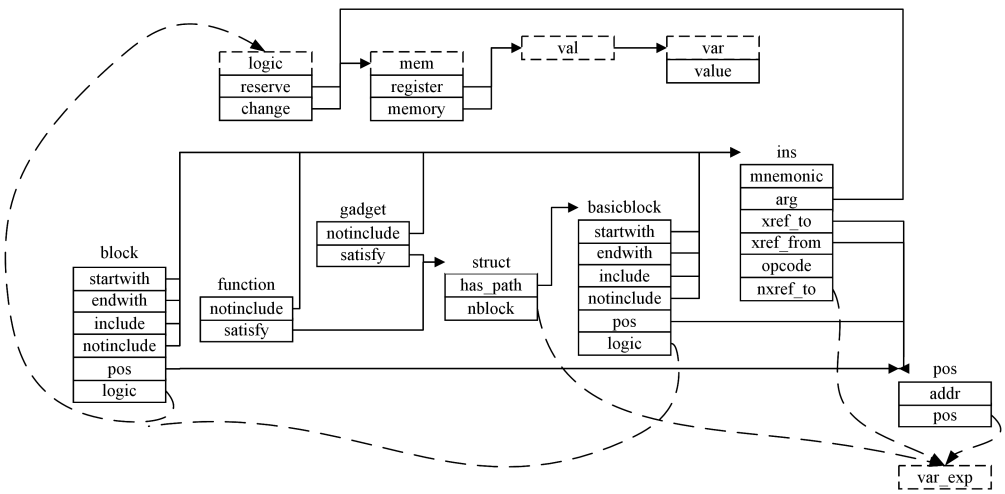

图 6 各类型和约束指令的对应关系与依赖关系

Fig. 6 Relationship between types and constraint

图 6 中,单箭头、实线和虚线都是约束指令的对象;实线箭头指向搜索当前约束对象前需要确定的对象,如搜索 block 对象,若该 block 对象包含约束指令 startwith,则在搜索该对象前需要先确定其依赖的 ins 对象;虚线箭头指向搜索当前约束对象后需要进一步进行约束判断的对象,如搜索 block 对象,若该 block 对象包含约束指令 logic,则在搜索该对象后需要判断搜索结果是否满足 logic 约束。

4.2.3　变量类型

在代码重用攻击中,很多时候需要搜索对特定寄存器或内存空间进行操作的 gadget,变量类型 var 能够对寄存器或地址值的范围进行限定,约束指令 value 后接正则表达式,其 BNF 范式定义如下:

〈var_definition〉::=“var”〈var_name〉“{”“value”〈stringconst〉“;”“}”

在搜索的过程中,可能需要多个主体对象对应同一寄存器。为了使各主体对象使用的寄存器保持一致,引入变量类型 val,由 var 类型的对象名定义,它能够选取 var 对象中定义的任意一个值,并能够保证在其他对象中出现的 val 值是一致的,其 BNF 范式定义如下:

〈val_definition〉::=“val”〈val_name〉“{”〈var_name〉{“or”〈var_name〉}“;”“}”

为了降低 GDL 的实现难度,寄存器和内存被封装到同一个变量类型 mr 中,mr 能够指定一个内存空间的范围,以“寄存器名/地址值,初始偏移值,最终偏移值,步长”的格式给出,其 BNF 范式定义如下:

〈mr_definition〉::=“mr”〈mr_name〉“{”〈mr_condition〉“}”

〈mr_condition〉::=“register”“val”〈val_name〉“;”|“register”〈stringconst〉“;”|“memory”“val”〈val_name〉“,”〈intconst〉“,”〈intconst〉“,”〈intconst〉“;”|“memory”〈stringconst〉“,”〈intconst〉“,”〈intconst〉“,”〈intconst〉“;”

4.2.4　主体类型

主体类型 ins 用于指定某条或某类指令,约束指令 opcode 后可接正则表达式,以指定指令字节码;约束指令 mnemonic 能够和 arg 结合起来指定指令的伪助记符;约束指令 xref_from,xref_to 和 pos 后可接一个 pos 对象名,用于指定代码的交叉引用情况和地址需满足的条件,其 BNF 范式定义如下:

〈ins_definition〉::=“ins”〈ins_name〉“{”{〈ins_condition〉}“}”

〈ins_condition〉::=“opcode”〈stringconst〉“;”|“mnemonic”〈stringconst〉“;”|“arg”〈stringconst〉[〈mem_name〉]“;”|“xref_from”〈pos_name〉“;”|“xref_to”〈pos_name〉“;”|“pos”〈pos_name〉“;”|“nxref_to”〈pos_name〉“;”

主体类型 basicblock 用于指定一个基本块,约束指令 startwith,endwith,include 和 notinclude 后可接指令名,这样就构成了指令和基本块的联系;约束指令 logic 后接一个 logic 对象名,用于指定基本块对寄存器的修改情况;约束指令 pos 后接一个 pos 对象名,用于指定基本块地址需满足的条件,其 BNF 范式定义如下:

〈basicblock_definition〉::=“basicblock”〈basicblock_name〉“{”{〈basicblock_condition〉}“}”

〈basicblock_condition〉::=“startwith”“ins”〈ins_name〉“;”|“endwith”“ins”〈ins_name〉“;”|“include”“ins”〈ins_name〉“;”|“notinclude”“ins”〈ins_name〉“;”|“logic”〈logic_name〉“;”|“pos”〈pos_name〉“;”|“;”

主体类型 block 用于指定一个连续代码片段,通常以一个控制流转移指令结束,它与主体类型 basicblock 相似,新增的约束指令 bdepth 和 idepth 用于指定 block 中字节长度和指令数量需满足的条件,其 BNF 范式定义如下:

〈block_definition〉::=“block”〈block_name〉“{”{〈block_condition〉}“}”

〈block_condition〉::=“startwith”“ins”〈ins_name〉“;”|“endwith”“ins”〈ins_name〉“;”|“include”“ins”〈ins_name〉“;”|“notinclude”“ins”〈ins_name〉“;”|“logic”〈logic_name〉“;”|“pos”〈pos_name〉“;”|〈var_exp_name〉“(”“bdepth”〈pos_name〉{“,”〈arg_name〉}“)”“;”|〈var_exp_name〉“(”“idepth”〈pos_name〉{“,”〈arg_name〉}“)”“;”

主体类型 struct 由一个或多个邻近的基本块构成,一般作为中间结构,用于指明已经搜索到的基本块之间的联系,使用约束指令 has_path 表明两个基本块之间有前驱后继关系,通过 nblock 进一步约束两个基本块之间的距离,若不指定则距离为 1,即前后相邻,其 BNF 范式定义如下:

〈struct_definition〉::=“struct”〈struct_name〉“{”{〈struct_condition〉}“}”

〈struct_condition〉::=“has_path”“(”〈basicblock_name〉“,”〈basicblock_name〉[“,”〈var_exp_name〉“(”“nblock”{“,”〈arg_name〉}“)”]“;”

在 struct 类型的基础上,主体类型 gadget 和 function 用来指定多基本块代码片段和函数,其 BNF 范式定义如下:

〈bbs_definition〉::=“bbs”〈bbs_name〉“{”{〈bbs_condition〉}“}”|“notinclude”“ins”〈ins_name〉“;”|“satisfy”〈logic_name〉“;”

〈function_definition〉::=“function”〈function_name〉“{”{〈function_condition〉}“}”|“startwith”“ins”〈ins_name〉“;”|“endwith”“ins”〈ins_name〉“;”|“notinclude”“ins”〈ins_name〉“;”|“satisfy”〈logic_name〉“;”

4.2.5　约束类型

约束类型 logic 用于判断 gadget 是否修改或保留了指定寄存器的值,其 BNF 范式如下:

〈logic_definition〉::=“log”〈log_name〉“{”{〈logic_condition〉}“}”

〈logic_condition〉::=“change”〈mem_name〉“;”|“reserve”〈mem_name〉“;”

约束类型 var_exp 用于判断操作数是否满足条件表达式,操作数由约束名后的参数列表指定,在判断时将实际值代入条件表达式并检查表达式是否为真,其 BNF 范式如下:

〈var_exp_definition〉::=“var_exp”〈var_exp_name〉“(”〈arg_name〉{“,”〈arg_name〉}“)”“{”〈var_exp〉“;”“}”

〈var_exp〉::=〈add_exp〉〈relop〉〈add_exp〉|〈add_exp〉

〈relop〉::=“<=”|“<”|“>”|“>=”|“==”|“!=”|“&&”|“||”

〈add_exp〉::=〈add_exp〉〈addop〉〈term〉|〈term〉

〈addop〉::=“+”|“-”

〈term〉::=〈term〉〈mulop〉〈factor〉|〈factor〉

〈mulop〉::=“*”|“/”|“%”

〈factor〉::=〈unary〉〈elem〉|〈elem〉

〈unary〉::=“-”|“!”|“~”

〈elem〉::=〈arg_name〉|〈addrconst〉|〈intconst〉

约束类型 pos 用于检查主体类型的地址是否在指定区间内，可以在定义时取得主体类型所在的基本块与函数的起始地址和结束地址，并通过约束指令 pos 来设定地址应满足的条件表达式，其 BNF 范式定义如下：

〈pos_definition〉::=“pos”〈pos_name〉“{”{〈pos_condition〉}“}”

〈pos_condition〉::=“addr”“start”〈pos_argument〉“;”|“addr”“end”〈pos_argument〉“;”|〈var_exp_name〉“(”“pos”{“,”〈arg_name〉}“)”“;”

〈pos_argument〉::=“block”〈block_name〉|“ins”〈ins_name〉|“bbs”〈bbs_name〉|“function”〈function_name〉|〈STRING-CONST〉

4.3 示例

如图 7 所示，GDL 代码会搜索 Intel x64 架构下，所有以 jmp 间接跳转指令结尾并且字节长度不超过 20 的连续代码片段。

```
1 var gen_reg
2 {
3    value "eax|ebx|ecx|edx|esi|edi";
4 }
5 val gen_reg
6 {
7    gen_reg;
8 }
9 mem gen_reg
10 {
11    register val gen_reg;
12 }
13 ins ins_indirect_call
14 {
15    mnemonic "call";
16    arg0 gen_reg;
17 }
18 log change_reg
19 {
20    change gen_reg;
21 }
22 log reserve_reg
23 {
24    reserve gen_reg;
25 }
26 basicblock b_change_reg
27 {
28    logic change_reg;
29 }
30 basicblock b_indirect_call
31 {
32    logic reserve_reg;
33    include ins ins_indirect_call;
34 }
35 struct st1
36 {
37    has_path(b_change_reg, b_indirect_call);
38 }
39 gadget g1
40 {
41    satisfy st1;
42 }
43 find gadget g1;
44
```

图 7 GDL 代码示例一

Fig. 7 Code for GDL example 1

如图 8 所示，GDL 代码会搜索两个相邻的基本块，第一个基本块修改了寄存器 reg 的值，第二个基本块保留了寄存器 reg 的值并且包含“call reg”指令。

```
1 var gen_reg
2 {
3    value "eax|ebx|ecx|edx|esi|edi";
4 }
5 val gen_reg
6 {
7    gen_reg;
8 }
9 mem gen_reg
10 {
11    register val gen_reg;
12 }
13 ins ins_indirect_call
14 {
15    mnemonic "call";
16    arg0 gen_reg;
17 }
18 log change_reg
19 {
20    change gen_reg;
21 }
22 log reserve_reg
23 {
24    reserve gen_reg;
25 }
26 basicblock b_change_reg
27 {
28    logic change_reg;
29 }
30 basicblock b_indirect_call
31 {
32    logic reserve_reg;
33    include ins ins_indirect_call;
34 }
35 struct st1
36 {
37    has_path(b_change_reg, b_indirect_call);
38 }
39 gadget g1
40 {
41    satisfy st1;
42 }
43 find gadget g1;
44
```

图 8 GDL 代码示例二

Fig. 8 Code for GDL example 2

5 原型系统

5.1 实现

文中构建原型系统主要用到两个开源项目：PLY(Python Lex-Yacc)和 BARF(Binary Analysis and Reverse engineering Framework)。其中，PLY 是 python 版本的 lex(Lexical Analyzar)和 yacc(Yet Another Compiler Compiler)工具，它提供了词法分析器和语法分析器的生成器，这里主要用于生成 GDL 的词法分析器和语法分析器，以解析 GDL 文件；BARF 是一个综合性的逆向工程框架，能够支持多种架构的二进制文件，并能将指令转化成中间表示 REIL(Reverse Engineering Intermediate Language)，为了能够得到 gadget 的更多信息以便分类和去重，需要利用 BARF 将所有生成的 gadget 转化成 REIL 指令。在此基础上，构建原型系统 GDLgadget，如图 9 所示。

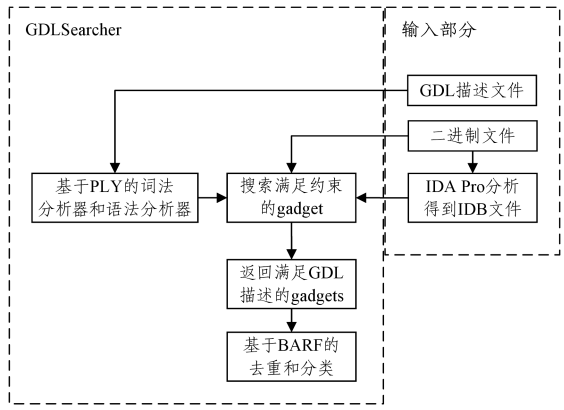

图 9 原型系统结构示意

Fig. 9 Architecture of prototype system

原型系统的执行流程如下：

(1)用户按照 GDL 的语法规范编写 GDL 文件以描述 gadget 的约束，并使用 IDA Pro 分析二进制文件，将 GDL 描述文件、二进制文件和 IDA Pro 生成的 IDB 文件作为输入；

(2)GDLgadget 读取 GDL 描述文件，在读取时根据约束分析各对象的依赖关系，并据此进行拓扑排序，确定各种约束的搜索顺序；

(3)GDLgadge 读取二进制文件以初始化 BARF 对象，读取 IDB 文件以得到所有的函数和基本块信息；

(4)预处理 var 和 val 对象，将 var 和 val 对象提取出来，如有多个 val 对象，按照深度优先的顺序遍历所有的 val 值，每生成一种 val 的组合就执行一次第 5 步，遍历完成之后进入第 6 步；

(5)按照拓扑顺序依次搜索满足约束条件的对象，跳过变量类型 var、val 和所有的约束类型。在这个过程中，若某个主体类型的对象与某种约束类型的对象有依赖关系，则在搜索后按照约束类型的条件进行进一步的筛选；

(6)返回需要输出的结果。

将 GDLgadget 与当前开源的 gadget 搜索工具的几种特性进行比较，结果如表 1 所列。

表 1　几种开源 gadget 搜索工具的特性

Table 1　Characteristics of several open source gadget tools

工具名	是否支持 gadget 结构的定义	是否支持 gadget 的分类	是否支持 gadget 的语义级搜索	是否支持 gadget 链的构建	能否用于 ROP	能否用于 JOP
ROPgadget	×	×	×	√	√	√
ropc	×	√	√	√	√	×
Ropper	×	×	×	×	√	√
angrop	×	√	√	√	√	×
rp++	×	×	×	×	√	√
BARFgadget	×	√	×	×	√	×
GDLgadget	√	√	×	×	√	√

搜索 gadget 的过程很大程度上依赖于对一些特定指令的搜索。不同的代码重用攻击技术中特定指令与所需 gadget 的关系通常有所不同，而现有搜索工具仅考虑了某一种或两种代码重用攻击技术中指令与 gadget 之间的关系，且考虑的类别较少，往往只包含指令和连续代码块两种类别，因此难以适用于多种代码重用攻击技术。而在 GDLgadget 中，用户能够利用 GDL 语言来指明 gadget 与指令之间的联系，GDLgadget 会根据各种类型的依赖关系搜索 gadget，因此能够用于多种代码重用攻击。

一般情况下，构建 gadget 链需要具备搜索特定语义 gadget 的能力，ROPgadget 虽然不具备这种能力，但能够通过几种特定的指令来构造在 Linux 下调用命令行的 ROP 链，然而这种方法所需指令较多，只适用于较大的文件，且不能构造其他的逻辑功能。angrop 和 ropc 是用于 ROP 的自动化 gadget 搜索和利用工具，它们能自动化地搜索 ROP 下的 gadget，并对其进行语义分析，然后根据设定的启发式的 gadget 链构建规则搜索具有特定语义的 gadget，这种方法能够适用于大多数文件且更加灵活，可以实现多种逻辑功能，但构建规则仅适用于 ROP。

截至目前，GDLgadget 没有实现 gadget 的语义级搜索。一方面，GDLgadget 需要结合 gadget 链的构建来确定搜索何种语义信息，而不同代码重用攻击技术构建 gadget 链的方法不同，因此需要关注的 gadget 语义信息也有所不同；另一方面，没有一种有效的方法能够构建多种代码重用攻击下的 gadget 链，这也是代码重用攻击技术自动化过程中的瓶颈所在，使得实现 gadget 工具的过程中缺乏先验知识来明确搜索的语义信息。GDL 中定义的逻辑约束已经涵盖了部分 gadget 的语义，可在此基础上分析现有的 gadget 链构建工具，进一步提取一些共性的特征作为语义搜索的依据。

5.2　实验

本文首先选择具有开源搜索工具的 ROP 和 JOP 作为目标，ROPgadget 作为对比工具，搜索对象选择 Windows 和 Linux 下常见的共享库和二进制文件，实验环境为 Ubuntu 16.04LTS 操作系统，内存 8GB，硬盘 30GB。

搜索以“ret”指令结尾的 gadget，设定总字节长度不超过 20，范围为程序的.text 节，实验结果如表 2 所列。从表 2 可以看出，对于多数二进制文件，两者搜索结果基本一致，但是存在部分 ROPgadget 能搜索到而 GDLgadget 搜索不到的 gadget，主要有以下两方面的原因。

(1)IDA 在分析二进制文件时易受到二进制文件本身编译优化、加壳加花等多种因素的影响，有时可能得不到完整的二进制信息，因此，存在部分代码片段不属于任何函数进而无法生成控制流图的情况。在 GDLgadget 的编码过程中为了简化后续的搜索，这一部分的代码片段会被直接筛选掉。

(2)在最后生成 gadget 的返回结果时，使用了 BARF 中将 gadget 转化成 REIL 指令的功能，而对于部分指令集如单指令多数据流扩展(Streaming SIMD Extensions，SSE)，BARF 尚不支持将其转化成 REIL 指令，因此在转化过程中也会被过滤掉。

表 2　ROP 下的 gadget 数量

Table 2　Number of gadgets in ROP

Name	Size	ROPgadget	GDLgadget
kernel32.dll	595KB	16425	17427
msvcrt.dll	637KB	22688	24841
AcroRd32.exe	350KB	1676	1708
libc6_2.19-0ubuntu 6.14_i386.so	1.8MB	80452	82657
libm-2.23.so	1.1MB	21560	23209
Proftpd	1.2MB	14176	14938
sudo	137KB	1695	1689

搜索以“jmp”和“call”间接跳转指令结尾的 gadget，设定总字节长度不超过 20，范围为程序的.text 节，实验结果如表 3 所列。

表 3　JOP 下的 gadget 数量

Table 3　Number of gadgets in JOP

Name	Size	ROPgadget			GDLgadget		
		jmp	pop+jmp	call	jmp	pop+jmp	call
kernel32.dll	595KB	86	1	419	744	16	1856
msvcrt.dll	637KB	50	0	372	226	7	459
AcroRd32.exe	350KB	0	0	543	23	0	744
libc6_2.19-0ubuntu 6.14_i386.so	1.8MB	4515	932	5874	9592	989	13597
libm-2.23.so	1.1MB	120	25	32	519	55	118
Proftpd	1.2MB	229	0	2646	2628	183	2794
sudo	137KB	141	74	81	179	74	456

搜索以“jmp”和“call”间接跳转指令结尾的 gadget 时，GDLgadget 的搜索数量远大于 ROPgadget，这是因为 ROPgadget 不能搜索到形如“call/jmp dword ptr [eax+4]”的指令，而文件中这类指令的数量并不少。

原型系统中使用 BARF 对 gadget 进行去重和分类，但 BARF 使用的中间语言 REIL 还不够完善，支持的指令集有限，使得 GDLgadget 的效果在某些方面有所欠缺，这可以通过使用其他二进制分析框架进行改善，如 angr、s2e 等。

多数代码重用攻击方法如 LOP，COOP 等，均未公开其 gadget 搜索工具，但是可以通过指令特征以及基本块之间的结构特征找到一些特定的 gadget。以 LOP 为例，其 loop gadget 主要涉及两个基本块，一个包含了比较指令 cmp，用于判断循环是否终止；另一个包含了 call 的间接跳转，用于函数调用，因此可用图 10 所示的 GDL 代码来搜索 loop gadget。

```
1 var_exp gt(t)
2 {
3    t<=5;
4 }
5 var_exp gt2(t)
6 {
7    t<=200;
8 }
9 mr gen_reg
10 {
11    register "eax|ebx|ecx|edx|esi|edi";
12 }
13 ins ins_indirect_call
14 {
15    mnemonic "call";
16    arg gen_reg;
17 }
18 ins ins_cmp
19 {
20    opcode "\x43?[\x38-\x3b]";
21 }
22 struct s0
23 {
24    include ins ins_cmp;
25 }
26 struct s1
27 {
28    include ins ins_indirect_call;
29 }
30 basicblock b_cmp
31 {
32    satisfy s0;
33 }
34 basicblock b_call
35 {
36    satisfy s1;
37 }
38 bbs bbs1
39 {
40    has_path(b_cmp, b_call, gt(ndepth), gt2(nbyte));
41    has_path(b_call, b_cmp, gt(ndepth), gt2(nbyte));
42 }
43 function f1
44 {
45    include bbs bbs1;
46 }
47 find bbs bbs0;
48 find function f0;
```

图 10 用于搜索 loop gadget 的 GDL 代码

Fig. 10 GDL code for searching loop gadget

以 Windows XP SP3 和 Ubuntu 12.04LTS 中的部分共享库为目标，实验环境为 Ubuntu 16.04LTS 8GB 30GB，对不可用的 loop gadget 进行筛选后实验结果如表 4 所列。

表 4 常用共享库下的 loop gadgets 数量

Table 4 Number of loop gadgets in widely used libraries

Platform	Name	Size/KB	loop gadgets
Linux	libc-2.15.so	1694	6
	libdbus-1.so.3.5.8	350	3
	libssl.so.1.0.0	347	1
Windows	ntdll.dll	576	4
	msvcrt.dll	335	3
	crtdll.dll	146	2

GDLgadget 是在 GDL 语言的基础上设计实现的，能够通过 GDL 语言定义约束来指定指令与 gadget 以及多个基本块之间的联系，但对于一些利用高级语言的语言特性的代码重用攻击，如 COOP，难以用约束表示语言特性，需要先总结出部分指令特征或基本块结构特征才能用 GDL 语言描述，再使用 GDLgadget 搜索。

实验证明，GDLgadget 适用于 ROP，JOP 和 LOP 下 gadget 的搜索，但仍存在以下问题需要在后续工作中解决：

1)结构上的约束很大程度上依赖于二进制文件的 CFG 图，因此，在定义结构约束时，不能处理 CFG 图中难以体现的间接引用，但可以加入源码级的分析作为辅助；

2)没有提供更加细致的语义级的约束，这是下一步工作的重点可以结合一些现有 gadget 语义分析技术来实现[26-27]；

3)没有特别考虑非控制数据攻击，因此通用模型和非控制数据攻击模型有所偏差，需要在以后的工作中加以修正。

结束语 代码重用攻击被广泛用于漏洞攻击，在当前的计算机体系结构和生态环境下暂时无法彻底消除，设计相关安全机制的研究人员需要综合考虑多种代码重用攻击方法，而各种攻击方法需要不同的 gadget，多数还缺少开源工具，这大大增加了设计安全机制的负担。针对这一问题，本文设计了一种代码重用攻击中的 gadget 描述语言，该语言能够以一种结构化的方式对代码重用攻击中的典型 gadget 进行描述，并且在描述过程中不需要关注 gadget 的搜索算法，降低了安全机制的设计难度。实验结果表明，本文所提出的 gadget 描述语言和原型系统能够较好地支持 ROP，JOP 和 LOP 攻击模式下的 gadget 搜索。

随着保护机制的不断增强，代码重用攻击正以一种更加复杂的形式发展，逐渐从控制流劫持转向数据流劫持，变得更加隐蔽且难以构造。本文提出的 GDL 能够适用于多种代码重用攻击，特别是指令特征明显、结构简单，以及多个基本块之间组织结构清晰的代码重用攻击。但 GDL 还不够成熟，它主要考虑了指令与 gadget，以及多个基本块之间的联系，而对于一些利用高级语言的语言特性实现的代码重用攻击，GDL 还不能够较好地适用，需要根据具体的语言特性对 GDL 作进一步的修改和完善。

参 考 文 献

[1] SOLAR DESIGNER. Getting around non-executable stack (and fix)[EB/OL]. https://seclists.org/bugtraq/1997/Aug/63.

[2] SHACHAM H. The geometry of innocent flesh on the bone: Return-into-libc without Function Calls (on the x86)[C]// ACM Conference on Computer and Communications Security. 2007:552-561.

[3] CHECKOWAY S, DAVI L, DMITRIENKO A, et al. Return-Oriented Programming without Returns[C]// Proceedings of the 17th ACM Conference on Computer and Communications Security. 2010:559-572.

[4] BLETSCH T, JIANG X, FREEH V W, et al. Jump-Oriented Programming: A New Class of Code-Reuse Attack[C]// Proceedings of the 6th ACM Symposium on Information. Computer and Communications Security, 2011:30-40.

[5] SADEGHI A, NIKSEFAT S, ROSTAMIPOUR M. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions[J]. Journal of Computer Virology and Hacking Techniques, Springer Paris, 2018, 14(2):139-156.

[6] MICROSOFT. Control Flow Guard[EB/OL]. https://docs.mi-

crosoft.com/en-us/windows/desktop/secbp/control-flow-guard.

[7] ABADI M,BUDIU M,ERLINGSSON U,et al. Control-Flow Integrity: Principles, Implementations, and Applications [J]. ACM Computing Surveys,2005,50(1):1-33.

[8] HISER J,NGUYEN-TUONG A,CO M,et al. ILR:Where'd my gadgets go? [C]// 2012 IEEE Symposium on Security and Privacy. 2012:571-585.

[9] WARTELL R,MOHAN V,HAMLEN K W,et al. Binary Stirring:Self-randomizing Instruction Addresses of Legacy x86 Binary Code[C]// Proceedings of the 2012 ACM Conference on Computer and Communications Security. 2012:157-168.

[10] PAPPAS V,POLYCHRONAKIS M,KEROMYTIS A D. Smashing the gadgets:Hindering return-oriented programming using in-place code randomization[C]// 2012 IEEE Symposium on Security and Privacy. 2012:601-615.

[11] CHEN X,BOS H,GIUFFRIDA C. CodeArmor:Virtualizing the Code Space to Counter Disclosure Attacks[C]// 2017 IEEE European Symposium on Security and Privacy (EuroS&P). 2017:514-529.

[12] BACKES M,NÜRNBERGER S,PLANCK M,et al. Oxymoron:Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing[C]// 23rd USENIX Security Symposium. 2014:433-447.

[13] SNOW K Z,MONROSE F,DAVI L,et al. Just-in-time code reuse:On the effectiveness of fine-grained address space layout randomization[C]// 2013 IEEE Symposium on Security and Privacy. 2013:574-588.

[14] PAX TEAM. PaX address space layout randomization [EB/OL]. https://pax.grsecurity.net/docs/aslr.txt.

[15] GOKTAS E,KOLLENDA B,KOPPE P,et al. Position-Independent Code Reuse:On the Effectiveness of ASLR in the Absence of Information Disclosure[C]// 2018 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE,2018:227-242.

[16] ZHANG M,SEKAR R. Control Flow Integrity for COTS Binaries[C]// 22nd USENIX Security Symposium. 2013:337-352.

[17] ZHANG C,WEI T,CHEN Z,et al. Practical Control Flow Integrity & Randomization for Binary Executables[C]// 2013 IEEE Symposium on Security and Privacy. 2013:559-573.

[18] VEEN V V D,GOKTAS E,CONTAG M,et al. A Tough Call:Mitigating Advanced Code-Reuse Attacks at the Binary Level [C]// 2016 IEEE Symposium on Security and Privacy (SP). 2016:934-953.

[19] LIU Y,SHI P,WANG X,et al. Transparent and Efficient CFI Enforcement with Intel Processor Trace[C]// 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). 2017:529-540.

[20] BOSMAN E,BOS H. Framing Signals—A Return to Portable Shellcode[C]// 2014 IEEE Symposium on Security and Privacy. 2014:243-258.

[21] LAN B,LI Y,SUN H,et al. Loop-oriented programming:A new code reuse attack to bypass modern defenses[C]// 2015 IEEE Trustcom/BigDataSE/ISPA. 2015:190-197.

[22] SCHUSTER F,TENDYCK T,LIEBCHEN C,et al. Counterfeit Object-oriented Programming on the Difficulty of Preventing Code Reuse Attacks in C++ Applications[C]// 2015 IEEE Symposium on Security and Privacy. 2015:745-762.

[23] CARLINI N,BARRESI A,PAYER M,et al. Control-Flow Bending:On the Effectiveness of Control-Flow Integrity[C]// 24th USENIX Security Symposium. 2015:161-176.

[24] ISPOGLOU K K,ALBASSAM B,JAEGER T,et al. Block Oriented Programming:Automating Data-Only Attacks[C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018:1868-1882.

[25] BIONDO A,CONTI M,LAIN D. Back To The Epilogue:Evading Control Flow Guard via Unaligned Targets[C]// Network and Distributed Systems Security (NDSS) Symposium. 2018.

[26] JIANG C,WANG Y J. A Technique of gadget Semantic Analysis Based on Expression Tree[J/OL]. Computer Engineering:1-10[2020-05-28]. https://doi.org/10.19678/j.issn.1000-3428.0056671.

[27] SCHWARTZ E J,AVGERINOS T,BRUMLEY D. Q?: Exploit Hardening Made Easy[C]// USENIX Security Symposium. 2011: 2541.

JIANG Chu, born in 1995, postgraduate, is a member of China Computer Federation. His main research interests include software security and so on.

WANG Yong-jie, born in 1974, Ph.D, associate professor. His main research interests include cyber security and so on.