软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(2):439−454 [doi: 10.13328/j.cnki.jos.005624]

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

一种基于元模型的访问控制策略描述语言*

罗 杨[1], 沈晴霓[1], 吴中海[1,2]

[1](北京大学 软件与微电子学院,北京 102600)

[2](软件工程国家工程研究中心(北京大学),北京 100871)

通讯作者: 吴中海, E-mail: wuzh@pku.edu.cn

摘 要: 为了保护云资源的安全,防止数据泄露和非授权访问,必须对云平台的资源访问实施访问控制.然而,目前主流云平台通常采用自己的安全策略语言和访问控制机制,从而造成两个问题:(1) 云用户若要使用多个云平台,则需要学习不同的策略语言,分别编写安全策略;(2) 云服务提供商需要自行设计符合自己平台的安全策略语言及访问控制机制,开发成本较高.对此,提出一种基于元模型的访问控制策略描述语言 PML 及其实施机制 PML-EM.PML 支持表达 BLP、RBAC、ABAC 等访问控制模型.PML-EM 实现了 3 个性质:策略语言无关性、访问控制模型无关性和程序设计语言无关性,从而降低了用户编写策略的成本与云服务提供商开发访问控制机制的成本.在 OpenStack 云平台上实现了 PML-EM 机制.实验结果表明,PML 策略支持从其他策略进行自动转换,在表达云中多租户场景时具有优势.性能方面,与 OpenStack 原有策略相比,PML 策略的评估开销为 4.8%.PML-EM 机制的侵入性较小,与云平台原有代码相比增加约 0.42%.

关键词: 访问控制模型;策略语言;策略转换;解释器;抽象语法树

中图法分类号: TP309

中文引用格式: 罗杨,沈晴霓,吴中海.一种基于元模型的访问控制策略描述语言.软件学报,2020,31(2):439−454. http://www.jos.org.cn/1000-9825/5624.htm

英文引用格式: Luo Y, Shen QN, Wu ZH. Access control policy specification language based on metamodel. Ruan Jian Xue Bao/ Journal of Software, 2020,31(2):439−454 (in Chinese). http://www.jos.org.cn/1000-9825/5624.htm

Access Control Policy Specification Language Based on Metamodel

LUO Yang[1], SHEN Qing-Ni[1], WU Zhong-Hai[1,2]

[1](School of Software and Microelectronics, Peking University, Beijing 102600, China)

[2](National Engineering Research Center for Software Engineering (Peking University), Beijing 100871, China)

Abstract: In order to protect the cloud resources, access control mechanisms have to be established in the cloud. However, cloud platforms have tendency to design their own security policy languages and authorization mechanisms. It leads to two issues: (i) a cloud user has to learn different policy languages to customize the permissions for each cloud, and (ii) a cloud service provider has to design and implement the authorization mechanism from the beginning, which is a high development cost. In this work, a new access control policy specification language called PML is proposed to support expressing multiple access control models like BLP, RBAC, ABAC and important features like multi-tenants. An authorization framework called PML-EM is implemented on OpenStack to centralize the authorization. PML-EM is irrelative to policy languages, access control models and programming languages that implement the authorization module. Other policies like XACML policy and OpenStack policy can be automatically translated into PML, which facilitates the migration between the clouds that both support PML-EM. The experimental results indicate PML-EM has improved the

* 基金项目: 国家自然科学基金(61232005, 61672062); 国家高技术研究发展计划(863)(2015AA016009)

Foundation item: National Natural Science Foundation of China (61232005, 61672062); National High Technology Research and Development Program of China (863) (2015AA016009)

收稿时间: 2017-08-19; 修改时间: 2018-04-19; 采用时间: 2018-07-17

flexibility of policy management from a tenant's perspective. And the performance overhead for policy evaluation is 4.8%, and the invasiveness is about 0.42%.

Key words: access control model; policy language; policy translation; interpreter; abstract syntax tree

近年来,云计算已经成为企业减少开销、按需分配计算资源的重要手段[1,2].目前,主流云平台通常利用 Web 接口提供服务.由于这些接口直接暴露在 Internet 上,因此,为了防止数据泄露和非授权访问,有必要利用访问控制策略对云平台资源进行安全防护.近年来,学术界提出了多种访问控制策略描述语言,如 XACML(extensible access control markup language)[3]、SPL(security policy language)[4]、Ponder[5]等.目前较为主流的 XACML 语言由标准组织 OASIS 提出,采用基于属性访问控制(attribute-based access control,简称 ABAC)模型[6].然而,除了 JBoss、Axiomaics、OpenAZ 等系统外,XACML 并没有被业界云平台广泛应用.大型公有云提供商如 AWS (Amazon Web services)、Microsoft Azure 等更倾向于自行设计一套策略描述语言用于自身平台.类似的还有 HP Openview PolicyXpert、CiscoAssure 等网络管理平台,也都仅支持自己的策略语言[7].OpenStack 作为目前主流的开源 IaaS(infrastructure as a service)云平台,利用 Keystone 组件为其他服务提供统一的身份认证,然而访问控制则依赖于云主机上本地存储的策略文件.该策略文件除了云服务提供商以外,其他人无法进行修改.OpenStack 策略支持 ABAC 及基于角色访问控制(role-based access control,简称 RBAC)[8,9].这些模型都是硬编码到云平台中,并且缺乏自定义功能.虽然目前多策略、混合策略成为访问控制策略领域的研究热点[10−12],然而这些解决方案仍然侧重于提高策略描述语言的表达能力和功能,大多数存在语法复杂、实现复杂度过高、难以直接适配实际环境等问题.

可见,当前云平台上存在多种不同的访问控制机制、策略语言.其差异具体表现为 3 个方面.

1) 各访问控制模型在概念上存在区别,难以互通.如 RBAC 定义了角色概念,ABAC 定义了属性概念等.
2) 即使访问控制模型相同或相似,不同的策略语言在语法层面仍存在巨大差异.例如,XACML 与 OpenStack 策略都支持 ABAC,然而 XACML 定义了策略集、策略、规则、集成算法等元素,并采用 XML 语法;而 OpenStack 策略则采用较为简单的"条件-动作"式的规则集合表达正向授权动作,基于 JSON 语法.
3) 策略决策、实施的逻辑需要依赖某种具体程序设计语言来实现.不同云平台的开发语言不同,导致策略决策、实施逻辑无法跨云平台使用.例如,OpenStack 采用 Python 实现,CloudStack 则采用 Java 实现,容器云平台 Kubernetes 则采用 Go 实现.即使这些云平台支持相同的访问控制模型及策略语言,其策略决策、实施逻辑仍需要在多个开发语言上分别进行实现.

本文假设云用户具有策略编写的能力,可以为本租户内的云资源定义访问控制策略.上述 3 个差异会造成如下问题:(1) 若同时使用多个云平台,云用户则需要学习多种安全策略语言,学习成本较高,并且云迁移时,安全策略无法迁移,需要重新编写;(2) 云服务提供商需要自行设计一套安全策略语言,并设计、实现相应的访问控制机制,设计、开发成本较高.

为了解决上述问题,本文提出一种基于请求(request)、策略(policy)、匹配器(matcher)、策略效果(effect)的访问控制元模型 PERM(policy-effect-request-matcher),并设计了相应的策略描述语言 PML(PERM modeling language).PML 能够表达一系列现有的访问控制模型,如 ACL(access control list),BLP(Bell LaPadula)[13]、RBAC、ABAC 等.基于 PML 提出了策略实施机制 PML-EM(PML enforcement mechanism).PML-EM 包含策略规则层、策略规则适配器层、模型层、解释器层等.现有的访问控制机制在实施安全策略时,通常直接依据某种具体策略语言设计相应的决策逻辑.这种方式虽然简单,但与策略语言、策略所基于的访问控制模型、程序实现语言都存在紧耦合,从而不具备通用型.本文提出了访问控制机制的 3 个性质:策略语言无关性、访问控制模型无关性、程序设计语言无关性.在 PML-EM 中,策略规则适配器层实现了策略语言无关性,模型层实现了访问控制模型无关性,解释器层实现了程序设计语言无关性.这 3 个性质对云用户、云服务提供商的意义在于:

1) 策略语言无关性:策略编写者可以利用一种自身所熟悉或业务所依赖的安全策略语言,对所有支持 PML-EM 的云平台实施统一访问控制.不需要因为业务迁移到不同云平台再学习额外的策略语言,极

大地节省了学习成本和迁移成本.

2) 访问控制模型无关性:策略语言本身通常基于某种访问控制模型,因此实现访问控制模型无关性是实现策略语言无关性的基础.与策略语言无关性类似,该性质方便了云用户使用同一套访问控制模型对不同云平台的资源实施访问控制.对云服务提供商而言,则可以在不显著增加设计成本的情况下,实现对多种访问控制模型(如 BLP、RBAC、ABAC)的支持,从而满足不同的应用场景.

3) 程序设计语言无关性:云服务提供商不需要使用自身云平台所使用的程序设计语言再将 PML 解释器再进行实现,从而极大地降低了云服务提供商的开发成本.解释器通常逻辑比较复杂.由于现代软件迭代速度加快,即使云服务提供商有意愿直接开发 PML 解释器,也很难保证各程序设计语言的 PML 解释器的版本一致,后续维护成本高昂.而在采用中间脚本语言如 Lua 实现 PML 解释器后,云服务提供商只需要提供 Lua 解释器即可,而 Lua 解释器在各知名程序设计语言中都有成熟的开源实现,可以直接采用.

本文在 OpenStack 云平台上实现了 PML-EM 框架.实验结果表明,在性能方面,与 OpenStack 原有策略相比,PML 的策略评估开销平均为 4.8%;在策略转换方面,实验验证了 PML 策略转换为 XACML、OpenStack、IAM 等策略的正确性;在表达能力方面,PML 支持 BLP、RBAC、ABAC 等访问控制模型以及多租户、RESTful (representational state transfer)[14]、策略集成、策略转换等特性,更适于云环境的访问控制需求.

1 相关工作

近年来,学术界和工业界都致力于研究如何利用访问控制实现更安全的云计算.业界主流公有云 AWS 通过 IAM(identity and access management)组件来实施基于身份的访问控制.IAM 为租户提供了用户、组、角色、权限、属性等概念来表达安全策略,包含了对 RBAC 和 ABAC 的支持.其他云平台如 Microsoft Azure、Google App Engine 也提供了类似的机制.这些安全策略通常都需要采用云服务提供商自己的策略语言来编写,不同平台的安全策略之间无法通用.

OASIS 组织基于 ABAC 提出了可扩展访问控制标记语言 XACML[3].XACML 定义了规则、策略、策略集等概念,并支持拒绝优先(deny-override)、允许优先(permit-override)等将决策集成算法.XACML 同时对访问控制框架进行了规范,定义了包括策略决策点(policy decision point,简称 PDP)、策略执行点(policy enforcement point,简称 PEP)、策略信息点(policy information point,简称 PIP)以及它们之间的交互流程.然而,XACML 仍存在一些不足,如不支持租户,以及必须依赖策略规则等限制.有些场景下(如 BLP、静态属性的 ABAC 等),只存在访问控制模型,而缺乏动态可变的策略规则.因此,若可以将访问控制模型和与其相应的策略规则分离,则可以实现更灵活的定义,如只定义访问控制模型而不定义策略规则.实现这种分离也有助于减少攻击面,在不存在策略规则的情况下,攻击者无法通过篡改策略进行攻击,而静态的访问控制模型在系统运行中通常保持不变,攻击者也难以进行篡改.

Wu 等人[15]提出了访问控制即服务(access control as a service,简称 ACaaS)的思想来实施细粒度的访问控制,然而该方法仅支持 RBAC 模型,并且依赖 AWS 云平台的 IAM,难以在其他云平台如 OpenStack 上应用.Tang 等人[16]提出了 OSAC(OpenStack access control)模型,对 OpenStack 的身份认证组件 keystone 中的域、租户等概念进行了形式化描述,并提出了跨域信任以支持两个域之间的权限授权.该工作与本文方法是正交的:OSAC 做出的域间信任决策可以看作 PML 策略的一个属性,从而进一步提高访问控制的粒度.Jin 等人[17]提出了适用于 IaaS 的形式化 ABAC 规范,并在 OpenStack 上进行了实现.该规范包括操作模型 $IaaS_{op}$ 和管理模型 $IaaS_{ad}$ 两部分,支持对访问控制策略本身的管理.然而该方法不支持桩函数、决策集成等特性,从而限制了模型的表达能力.

Damianou 等人[5]提出了面向分布式系统的策略规范语言 Ponder.Ponder 支持授权策略、代理策略、信息过滤策略、抑制策略以及基于事件触发的义务策略.相比之下,PML 只支持授权策略,并采用表达式求值来实现其策略决策的灵活性.在授权策略方面,Ponder 缺乏租户、算数运算符、逻辑运算符等表达特性,支持静态策略冲突检测,但缺乏决策集成机制(即冲突消解).Ribeiro 等人[4]提出了安全策略语言(security policy language,简称

SPL).SPL 支持实体、实体间关系的表达,支持利用策略属性、量词的比较进行策略决策.SPL 缺乏桩函数支持,因此在自定义策略方面受限.Kagal 等人[18]提出了基于道义逻辑的策略描述语言 Rei,以满足动态、开放环境中的安全需求.在 Rei 中,策略以约束的形式规定可在资源上进行的授权或义务性操作.Rei 通过元策略解决策略一致性和冲突问题.该语言支持组、角色等表达特性.Ashley 等人[19]提出了企业级隐私授权语言(enterprise privacy authorization language,简称 EPAL).EPAL 更侧重于隐私保护而不是访问控制.EPAL 采用基于目的的访问控制模型(purpose-based access control).例如,当医生具有治疗诊断目的时,就可以访问对应病人的信息.该模型简化了传统的 RBAC 模型,但引入了额外的目的元素.该语言提供了一定的策略精化和冲突检测机制.

2 PML 语言

实施访问控制策略通常包括策略规则以及实施这些策略规则的逻辑两个部分.一个支持多用户、多安全策略的系统通常支持用户自定义策略规则.然而实施这些策略规则的逻辑通常是静态不可改变的,导致策略配置方面不够灵活.为了让云用户能够灵活配置自身策略,需要明确策略评估中策略实施逻辑的概念.由于策略实施逻辑本质上反映了策略规则所属的访问控制模型,因此有必要将访问控制模型与策略规则进行分离.在本文中,将反映策略实施逻辑的抽象访问控制模型称为“模型”,将被实施的策略规则称为“策略”或“策略规则”.模型代表了逻辑,策略代表数据,它们之间类似于一个程序中代码与数据之间的关系.

本节提出 PERM 建模语言(PERM modeling language,简称 PML)来表达访问控制模型及其策略规则.PML 语言是一个元模型语言,其基于本文提出的 PERM(policy-effect-request-matcher)元模型.PERM 元模型如图 1 所示,包含 4 个基本原语:Request、Policy、Effect、Matcher,1 个可选原语:Role 以及若干特性,如域、属性、桩函数等.

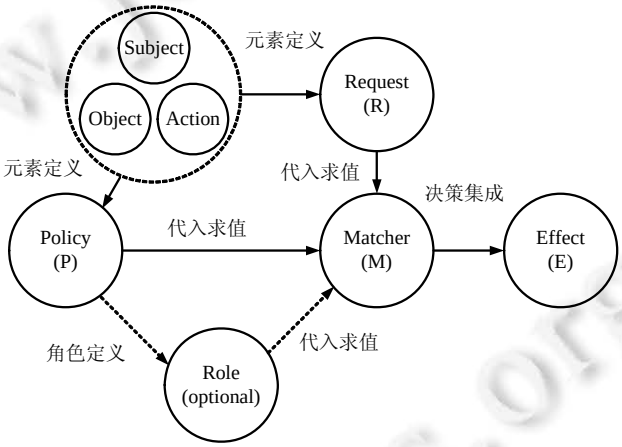

Fig.1 PERM metamodel

图 1 PERM 元模型

2.1 Request原语

Request 原语是对访问请求的抽象,其定义了访问请求的语义.一个访问请求通常由经典的三元组构成:进行访问的主体(subject)、被访问的客体(object)、访问动作(action).PML 中描述上述三元组的 Request 原语可表示为

$$r=sub,obj,act.$$

PML 提供了自定制 Request 原语的能力,例如,如果不需要控制到具体的资源,可采用 *sub,act* 两元组的格式.如果主体需要采用所属域等额外信息来标识,则可采用 *domain,sub,obj,act* 四元组作为 Request 原语声明.

2.2 Policy原语

Policy 原语定义了策略规则的语义,该字段制定了 PML 策略的解释器应如何解释策略规则.为简单起见,本文采用逗号分隔值(comma-separated values,简称 CSV)来作为策略规则的语法.与 Request 原语类似,经典的 Policy 原语也由主体、客体、动作三元组构成:

p=sub,obj,act.

与其相应的策略规则可以为

*p,alice,data*1,*read.*

策略中的每一行都是一条策略规则.策略规则的第 1 个元素是策略类型,该字段与 Policy 原语相对应,通过指定不同的策略类型可以支持多个 Policy 原语,如 *p*,*p*2 等.通常情况下,不需要指定多个策略类型.上面的策略规则与之前的经典 Policy 原语的绑定关系为

(*alice*,*data*1,*read*)→(*p.sub*,*p.obj*,*p.act*).

上述绑定类似于变量赋值,可以在 Matcher 原语中使用.

2.3 Matcher原语

Matcher 原语定义了策略规则如何与访问请求进行匹配的匹配器,其本质上是布尔表达式.可以理解为 Request、Policy 等原语定义了关于策略和请求的变量,然后将这些变量代入 Matcher 原语中求值,从而进行策略决策.Matcher 原语支持+,–,*,/等算数运算符、==,!=,>,<等关系运算符以及&&(与),||(或),!(非)等逻辑运算符.Matcher 原语最终的运算结果应为布尔值 0 或 1:1 表示策略规则满足该匹配器,0 则表示不满足.与上节中列举的经典 Policy 原语相对应的 Matcher 原语为

m=r.sub==p.sub && *r.obj==p.obj* && *r.act==p.act.*

该 Matcher 原语表示访问请求中的主体、客体、动作三元组应与策略规则中的主体、客体、动作三元组分别对应相同.

2.4 Effect原语

Effect 原语定义了当多个策略规则同时匹配访问请求时,该如何对多个决策结果进行集成以实现统一决策,决策结果包括 *allow* 和 *deny* 两种.Effect 原语支持 *some*,*any* 等量词以及条件关键字 *where*,&&(与),||(或),!(非)等逻辑运算符.*some* 量词判断是否存在一条策略规则满足匹配器,*any* 量词则判断是否所有的策略规则都满足匹配器.*Where* 关键字表示后面跟策略规则的筛选条件.Effect 原语可以表达允许优先(allow-override)、拒绝优先(deny-override)等多种决策集成方法.允许优先可以表达为

e=some(*where*(*p.eft==allow*)).

该 Effect 原语表示如果存在任意一个决策结果为 *allow* 的匹配规则,则最终决策结果为 *allow*,即 *allow-override*.其中,*p.eft* 表示策略规则的决策结果,可以为 *allow* 或者 *deny*.当不指定规则的决策结果时,取默认值 *allow*.利用 Effect 原语还可以表达拒绝优先:

e=!*some*(*where*(*p.eft==deny*)).

该 Effect 原语表示如果不存在任何决策结果为 *deny* 的匹配规则,则最终决策结果为 *allow*,否则为 *deny*.Effect 原语还可以利用逻辑运算符进行连接:

e=some(*where*(*p.eft==allow*)) && !*some*(*where*(*p.eft==deny*)).

该Effect原语表示当至少存在1个决策结果为*allow*的匹配规则,且不存在决策结果为*deny*的匹配规则时,则最终决策结果为 *allow*.这时正向授权和负向授权同时存在,并且拒绝优先.

2.5 Role原语

Role 原语定义了 RBAC 中的角色继承关系,该原语的特点是支持多重 RBAC 体系的表达.比如,当主体和客体同时具有角色(或组)的概念时,主体角色和客体角色两套 RBAC 体系可以互不干扰.同时支持主体角色和客

体角色的 Role 原语可以定义如下:

g=_,_

*g*2=_,_

上述 Role 原语表示 *g* 是一个 RBAC 体系,*g*2 是另一个 RBAC 体系._,_表示角色继承关系的前项和后项,即前项继承后项角色的权限.PML 把角色继承关系也表达为策略规则,如

g,*alice*,*data*2_*admin*.

上述策略规则表示 *alice* 具有角色 *data*2_*admin*.这里的 *alice* 可以为具体的某个主体或者另一个角色.*g* 则表示策略类型,与 Role 原语相对应.在 Matcher 原语中,请求主体与策略规则主体之间的是否具有角色继承关系可以用布尔函数 *g*(*r.sub*,*p.sub*)来表达:值为 1 表示两者具有角色继承关系,0 则表示不具备该关系.角色继承关系支持间接继承.可以看出,Matcher 原语中的一个布尔函数 *g*,与 Role 原语中的 *g* 和策略规则中相应的策略类型 *g* 互相对应,共同完成 RBAC 角色的指派.其他 RBAC 体系,如 *g*2 用法类似,这里不再赘述.下面的例子给出了完整的 Matcher 原语.

m=*g*(*r.sub*,*p.sub*) && *r.obj*==*p.obj* && *r.act*==*p.act*.

该 Matcher 原语表达了经典的 RBAC 场景:策略规则定义哪些角色可以具备哪些权限(即采用哪种动作访问哪个资源),只有当请求主体继承策略规则中指定的角色,并且请求与策略规则中的资源和动作对应相同时,Matcher 原语才会成功匹配该请求与该策略规则.

2.6 域

云平台中通常存在租户的概念.为了实现灵活的授权管理,PML 通过域内 Role 原语来支持租户内的授权.一个域代表一个租户,域内角色只在本域内生效,也只能分配本域内的资源的权限.域内角色机制实现了云平台租户间的安全隔离.域内 Role 原语可以定义为

g=_,_,_.

原语中第 3 个_表示域的概念.相对应的策略规则的实例如下:

- *p*,*admin*,*tenant*1,*data*1,*read*;
- *p*,*admin*,*tenant*2,*data*2,*read*;
- *g*,*alice*,*admin*,*tenant*1;
- *g*,*alice*,*user*,*tenant*2.

相对应的 Matcher 原语可以为

m=*g*(*r.sub*,*p.sub*,*r.dom*) && *r.dom*==*p.dom* && *r.obj*==*p.obj* && *r.act*==*p.act*.

该实例表示 *tenant*1 的域内角色 *admin* 可以读取 *data*1,*tenant*2 的域内角色 *admin* 可以读取 *data*2.*alice* 是在 *tenant*1 域中是 *admin* 角色,但在 *tenant*2 域中是 *user* 角色,因此根据策略规则,*alice* 只能读取 *data*1 而不能读取 *data*2.

2.7 属性

为了支持 ABAC 的属性概念,PML 采用.(点)语法来表达元素属性(元素即主体、客体、动作等).属性本身也是元素,因此可以继续获取属性的属性.例如,*r.sub.domain* 表示访问请求的主体所在的域,而 *r.sub.domain.owner* 则表示该域的所有者.

m=*r.sub.domain*==*r.obj.domain*.

上述 Matcher 原语表示只有当请求的主体所在的域与请求的客体所在的域相同时,才允许该请求.PML 本身也依赖与属性类似的语法,如 *r.sub* 可理解为请求 *r* 这个元素的 *sub* 属性.通过提供一致的语法,有助于降低策略编写者学习语言的成本.

2.8 桩函数

虽然 PML 的 Matcher 原语通过各种运算符提供了强大的表达能力,但是仍有一些复杂的处理逻辑无法用

PML 表达,如查询外部数据库、进行复杂数学运算等.PML 中提供了桩函数机制以满足自定义处理逻辑的需求.策略编写者可以在 PML 以外以其他程序设计语言自定义若干桩函数(stub function),然后将这些函数注册到 PML 中,从而被 PML 的 Matcher 原语所识别和调用.桩函数与普通函数类似,可以带有多个参数,并具有返回值.下面给出桩函数 *my_func* 的例子.*my_func* 判断请求客体与策略规则客体之间是否匹配,并返回布尔值.首先需要将该函数进行注册:

register_function(“*my_func*”,*my_func*).

my_func 的实现可以如下:

```
boolean my_func(r.obj,p.obj) {
  return r.obj==p.obj
}
```

这样就可以在 Matcher 原语中指定该函数,参与到匹配中:

m=r.sub==p.sub && *my_func*(*r.obj*,*p.obj*) && *r.act==p.act*.

PML 提供内置的桩函数,以满足常用 Matcher 匹配需求.比如,*keyMatch*(*r.obj*,*p.obj*)函数支持匹配 RESTful 路径,当请求中的 RESTful 路径匹配策略规则时,返回布尔值 1,并允许以“*”作为通配符.*regexMatch*(*r.obj*,*p.obj*)函数则提供了更强大的正则表达式匹配的功能,当请求中的客体匹配策略规则中指定的正则表达式时,返回布尔值 1.

3 PML 策略实施机制

访问控制策略需要实施机制来保证策略的决策、实施,例如,XACML 策略由策略决策点 PDP 来进行实施.现有的策略实施机制都依赖于某种策略语言或访问控制模型,并且需要在某个具体的程序设计语言上进行实现,无法很好地满足前文提出的访问控制机制的 3 个性质:策略语言无关性、访问控制模型无关性、程序设计语言无关性.第 2 节提出的 PML 语言能够实现对不同访问控制模型的统一描述,从而实现访问控制模型无关性.为了实现另外两个性质,本节在 PML 的基础上提出了 PML 策略实施机制(PML enforcement mechanism,简称 PML-EM).

3.1 总体架构

PML-EM 的体系结构如图 2 所示,自顶向下包括以下 4 个层次.

1) 策略规则层:策略规则层包含各个访问控制策略描述语言的策略规则,所有策略编写者所制定的策略规则位于该层,如 XACML 规则、OpenStack 规则、IAM 规则等.PML 语言除了描述访问控制模型外,自身也支持描述策略规则,即 PML 规则,PML 规则也属于策略规则层.与其他策略不同的是,PML 规则可以直接与 PML 模型兼容,因此不需要额外的策略规则适配器.
2) 策略规则适配器层:策略规则适配器层包含对各个策略描述语言的适配器,适配器可以将某一种策略语言描述的规则转换为等价的 PML 规则.例如,XACML 策略规则适配器可以将 XACML 规则等价转换为 PML 规则.这里,PML 规则就类似于一种中间语言.该层通过适配器机制实现了策略语言无关性.需要注意的是,该转换主要是语法层面的转换,不包括语义的转换,语义层面的解析由模型层负责.
3) 模型层:该层包含采用 PML 语言描述的各种不同的访问控制模型.不同的策略描述语言(如 XACML、OpenStack 等)的访问控制模型也不同,因此需要采用 PML 语言进行统一描述.该层实现了访问控制模型无关性.
4) 解释器层:该层主要负责解析 PML 模型和规则,依据解析结果对访问请求进行策略决策,给出 *allow* 或者 *deny* 的结果.解释器采用通用的脚本语言 Lua 实现,从而实现程序设计语言无关性.

在 PML-EM 机制中,策略规则层即为不同策略语言所描述的具体策略规则,如 XACML 规则、OpenStack 规则以及本文的 PML 规则.而模型层则为 PML 语言所描述的各种不同的访问控制模型,在第 2 节已经阐述.因此,下文主要阐述另外两层:策略规则适配器层和解释器层.

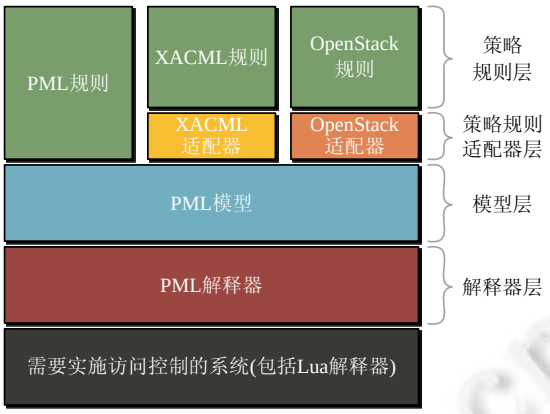

Fig.2 PML-EM: PML enforcement mechanism

图 2 PML 策略实施机制 PML-EM

3.2 策略规则适配器层

策略规则适配器负责将某种策略语言描述的策略规则(即目标策略)转换为 PML 模型及 PML 规则.本节提出一种基于抽象语法树(abstract syntax tree,简称 AST)的策略转换方法.AST 常用于编程语言源代码分析领域.将源代码通过词法、语法分析解析为 AST 后,该 AST 不再依赖于具体编程语言语法,相当于一种通用的“中间语言”,为下一步处理提供一个清晰的接口.策略语言与编程语言有类似之处,都具有特定的语法,因此首先将策略语言(包括目标策略和 PML 模型、规则)转换为 AST 进行处理,可以屏蔽策略语言的语法细节,有助于实现策略转换.AST 是一种无语义损失的表示形式,意味着策略不仅可以转换为 AST,AST 还可以转换回原有的策略.利用 AST 技术,原来的将目标策略转换为 PML 模型、规则的问题就可以简化为将目标策略 AST 转换为 PML 对应的 AST 的问题.由于策略语言大多基于某种通用数据格式,如 XML、JSON 等,因此可以直接借助相关的解析器等就能实现将目标策略的各个元素,如策略、规则、主体、客体、动作等表达成 AST.

为了完成策略转换,给出一种策略转换算法.算法的输入是目标策略 AST,输出是 PML 对应的 AST.目标策略 AST 也包含对请求格式的定义.策略转换算法的具体步骤如下.

1) 分析目标策略的请求格式 AST,将 AST 的每个叶子节点转换为 Request 原语 AST 的叶子节点.
2) 将目标请求 AST 中,每一个带有匹配逻辑的 AST 子树转换为 Matcher 原语的 AST 子树.
3) 将目标策略 AST 中,带有策略集成逻辑的 AST 子树转换为 Effect 原语的 AST 子树.
4) 将具有相同子树结构,并具有同一个父节点的 Matcher 原语 AST 子树定义为同类 Matcher 原语,将其合并为一个 Matcher 原语、一个 Policy 原语以及若干策略规则.
5) 相同格式的 Policy 原语进行合并,合并为一个 Policy 原语,同时合并对应的策略规则.

上述算法的伪代码如算法 1 所示.

算法 1. 策略转换算法.

Input: targetAST:目标策略 AST.

Output: pmlAST:PML AST.

```
1.  pmlAST.Request.leafNodes=targetAST.request.leafNodes   //生成 PML AST 的 Request 原语
2.  for each p in targetAST.subTrees do
3.    if hasMatchLogic(p) then
4.      pmlAST.Matcher.subTrees.add(p)   //转换带有匹配逻辑的 AST 子树
5.    else if hasIntegrationLogic(p) then
6.      pmlAST.Effect.subTrees.add(p)   //转换带有策略集成逻辑的 AST 子树
```

```
    end if
  end for
  for each m1, m2 in pmlAST.Matcher do
    if hasSameParent(m1,m2) && hasSameSubtree(m1,m2) then
      m, p, rules=combineMatcher(m1,m2)  //合并具有相同子树结构、父节点的 Matcher 原语
      pmlAST.Matcher.add(m)
      pmlAST.Policy.add(p)
      pmlAST.Rules.add(rules)
    end if
  end for
  for each p1, p2 in pmlAST.Policy do
    if hasSameFormat(p1,p2) then
      p=combinePolicy(p1,p2)
      rules=combineRules(p1.rules,p2.rules)
      pmlAST.Policy.add(p)  //合并相同格式的 Policy 原语
      pmlAST.Rules.add(rules)  //合并对应的策略规则
    end if
  end for
  return pmlAST
```

3.3 解释器层

PML 解释器主要包含 3 个原语:加载模型、加载策略和策略决策.本文假设 PML 模型、策略都存储在策略数据库中.接下来详细介绍这 3 个原语.

1) 加载模型:负责加载 PML 模型.输入为策略数据库,输出为 PML 模型.加载 PML 模型主要包括解析各种原语:Request 原语、Policy 原语、Matcher 原语、Effect 原语、Role 原语等,以及加载用户注册的桩函数.

2) 加载策略:负责加载 PML 规则.输入为策略数据库,输出为 PML 规则.加载策略较为简单,主要是将 PML 规则一条一条地加载到解释器中.如果存在角色继承规则,则需要在加载完毕时建立角色继承关系树,从而实现对间接角色继承的判断.每一个 RBAC 角色体系都具有不同的角色继承关系树,从而互不干扰.

3) 策略决策:对请求进行策略决策.输入为 PML 模型、PML 规则以及访问请求,输出为决策结果.在 PML 中,访问请求的语义由 Request 原语定义,PML 策略规则的语义则由 Policy 原语定义.在决策过程中,PML 解释器会根据上述语义将访问请求的上下文(即具体的主体、客体、动作等)、每一条 PML 规则以及所有的桩函数代入到 Matcher 原语中,然后对 Matcher 原语求值.对 Matcher 的求值可以采用 Lua 的按字符串执行的内置函数 loadstring 进行实现.需要注意的是,Role 原语(包括域)和属性也是以桩函数的形式定义的,因此可与用户自定义的桩函数一同代入到 Matcher 原语中,不需要特殊处理.由于可能存在多条 PML 规则,因此 Matcher 的求值结果也可能不唯一.这时,PML 解释器再通过 Effect 原语对多个 Matcher 求值结果进行集成,从而得到统一的决策结果.算法 2 描述了策略决策的过程.

算法 2. 策略决策算法.

Input: model:PML 模型;policy:PML 策略;request:访问请求.

Output: decision:决策结果.

```
1. Set decisions={·}
2. for each matcher in model do
```

```
3.  matcher.inject(model.stubFunctions)          //将桩函数代入 Matcher 原语
4.  matcher.inject(request)                      //将 Request 原语代入 Matcher 原语
5.  for each rule in policy do
6.    matcher.inject(rule)                       //将 Policy 原语对应的规则代入 Matcher 原语
7.    decisions.add(evaluate(matcher))           //对 Matcher 求值并记录决策结果
8.  end for
9. end for
10. decision=combineDecision(decisions,model.Effect)   //利用 Effect 原语进行决策结果集成
11. return decision
```

由于 Matcher 需要执行外部的桩函数代码来进行策略决策,假如桩函数内部存在恶意代码,则会影响访问控制机制甚至整个云平台系统的安全性.这个问题的解决方案包括:

1) 为桩函数创建沙箱,桩函数仅在沙箱内执行代码,不会影响到整个 PML 解释器以及外部系统.Lua 语言环境本身也是一层沙箱,因此可以作为第 2 道防线.
2) 静态代码检查,即服务提供商在为用户注册桩函数之前,先通过代码静态分析等手段检查桩函数源代码,确定其没有恶意行为.
3) 定制桩函数编辑接口,即服务提供商不直接提供编写桩函数代码的功能,而是将常用的功能封装在编辑接口(可以为命令行或图形界面)中,用户只能采用预定义的编辑接口定制桩函数的功能,由于编辑接口的安全性是可控的,因此该方式也可以有效屏蔽恶意代码.

PML 解释器本身采用脚本语言 Lua 编写.Lua 是一种轻量级的脚本语言,该语言代码通常不直接独立运行,而是作为脚本语言嵌套在其他程序设计语言中执行特定逻辑.Lua 的解释器通常非常轻量级(不大于 200KB),目前,绝大多数主流程序设计语言如 C、Java、Python 等都有成熟、开源的 Lua 解释器实现.因此,用 Lua 编写 PML 解释器具有跨程序设计语言的优势,从而可实现程序设计语言无关性:需要实施访问控制的系统不需要实现 PML 解释器,而只需要嵌入现有的 Lua 解释器,即可实现对 PML 的支持.另外,从接口交互来讲,访问控制与系统其他模块的接口比较明确、耦合程度较低,因此,用脚本语言实现也不会造成系统复杂度的明显增加.

4 实验验证

本文实验环境为 OpenStack Pike 云平台,集群中包含 1 个控制节点(包含网络)、4 个计算节点和 2 个块存储节点.本文实现的原型系统如图 3 所示,包含一个新增加的 OpenStack 组件 Patron 以及请求过滤器模块 AEM (access endpoint middleware).AEM 是一个安装在云平台其他组件(如 Nova、Glance)上的 WSGI(Web server gateway interface)过滤器组件.所有的云平台请求都会经过 AEM 的过滤,将请求的安全上下文信息(如主体、客体、动作等)发送到 Patron 服务.Patron 服务实现了 PML-EM 策略实施机制,对 AEM 发送过来的请求进行授权决策,决定是否允许该请求:若允许,则 AEM 会放行该请求;否则,则返回 HTTP 403 拒绝访问错误.为了提高性能,AEM 中加入缓存机制,可将已进行的策略决策结果进行缓存,从而减少对 Patron 的决策请求.

Patron 组件包括以下 3 个模块.

1) API 模块(patron-api):对外以 RESTful 接口提供 Patron 服务,接受决策请求,返回决策结果.
2) 决策模块(patron-decide):负责进行策略决策.实现了 PML-EM 策略实施机制,包括 PML 解释器、各策略规则适配器等.
3) 策略更新模块(patron-update):负责策略存储和策略更新.

原型系统中,PML 解释器、策略规则适配器采用 Lua 语言实现,其他模块则与 OpenStack 一致,采用 Python 实现.这里的 PML 解释器、策略规则适配器可以在其他系统上重用,印证了 PML-EM 的程序设计语言无关性,即,对需要实施访问控制系统的开发语言不存在依赖.

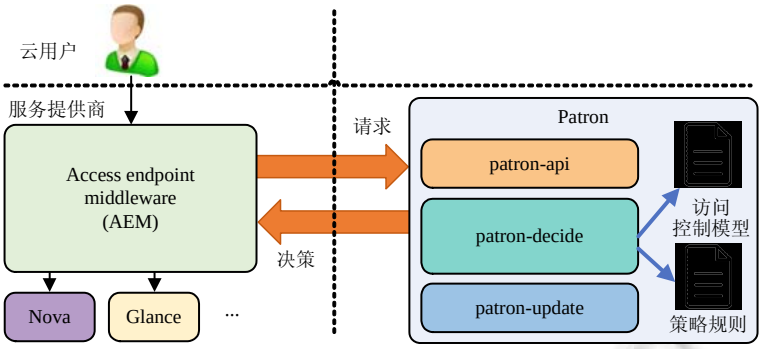

Fig.3 Architecture of the OpenStack cloud after using PML

图 3 采用 PML 后的 OpenStack 云平台架构

4.1 表达能力

本文将 PML 与其他主流策略语言进行了对比,结果见表 1.ACL 模型允许客体的所有者(即属主)自动拥有对该客体的权限.在 PML 中,可通过属性来描述 ACL 中属主的概念.BLP 模型需要依据主体、客体的安全级别来决定主体是否可以读、写客体,因此需要策略语言对安全级别的支持.PML 可以通过 Matcher 原语提供安全级别的比较.PML 的 Role 原语和属性机制则分别对 RBAC 和 ABAC 进行了支持.Superuser 表示特权用户,特权用户可以执行任意操作,从而方便云服务提供商对自身进行管理.PML 可以利用 Matcher 原语对特权用户进行定义,当匹配特权用户 ID 时,使决策结果为允许.云平台通常以租户(域)为单位来管理云用户,因此,是否支持多租户对云平台访问控制至关重要.PML 通过域特性以支持云环境下租户粒度的权限管理.云平台以及其他 Web 服务通常采用 RESTful[14]接口作为程序调用接口,PML 通过内置桩函数 keyMatch,regexMatch 等支持对 RESTful 路径进行访问控制.从策略转换和多语言可以看出,支持则是 PML 所独有的特性.PML-EM 的策略规则适配器机制可将其他语言的策略转化为 PML 策略,而多语言支持则允许 PML 解释器在不同开发语言所编写的系统中运行,从而满足不同平台的访问控制需求.另外,从表 1 中可以看出,所有策略语言均支持 RBAC,大部分策略语言支持 ABAC.由于大多数策略语言并不是为云环境设计的,因此不支持多租户.在策略语言中,除了 PML 以外,XACML 支持最多的语言特性.

Table 1 Expressiveness comparison of different policy languages

表 1 不同策略语言的表达能力对比

策略语言	支持特性										
	ACL	BLP	RBAC	ABAC	Superuser	多租户	RESTful	决策集成	桩函数	策略转换	多语言
OpenStack	×	×	√	√	×	√	√	×	×	×	×
AWS IAM	✓	×	√	√	×	√	×	√	×	×	×
ACaaS[15]	×	×	√	×	√	√	√	√	×	×	×
OSAC[16]	×	×	√	×	×	√	√	×	×	×	×
$IaaS_{op}$[17]	√	×	√	√	√	√	√	×	×	×	×
XACML[3]	√	√	√	√	×	×	√	√	√	×	×
Ponder[5]	×	×	√	√	×	×	×	×	√	×	×
SPL[4]	√	×	√	√	×	×	×	√	×	×	×
PML	√	√	√	√	√	√	√	√	√	√	√

4.2 策略转换

为了验证策略规则适配器,本文设计了 XACML 策略、OpenStack 策略和 AWS IAM 策略的策略规则适配器,并采用用例验证其效果.

4.2.1 XACML 策略转换

XACML 访问控制策略采用 XML 语法来描述,包括 Policy、Rule 等核心概念,并通过策略集成算法和规则集成算法实现统一决策.本节通过 XACML conformance 测试集来验证 XACML 策略规则适配器的可用性.由

OASIS 推出的 XACML conformance 测试集提供了对 XACML 策略、请求、响应的标准测试.所有 XACML PDP 实现都需要满足该测试集的要求.该测试集包括强制的 5 类测试用例:IIA、IIB、IIC、IID、IIE,分别为属性用例、目标用例、函数用例、集成算法用例和模式用例.下面以 IIA001 策略为例验证策略转换的效果,XACML 策略规则适配器首先需要将 IIA001 策略预处理为如图 4 所示的 AST.

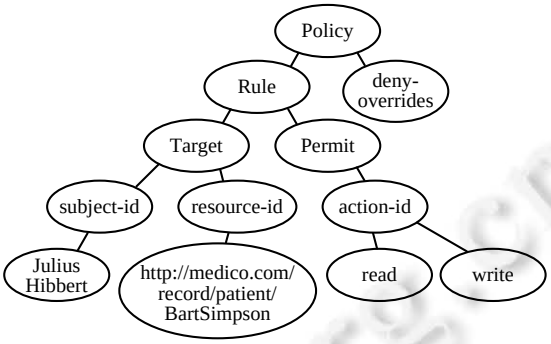

Fig.4 AST of XACML policy: IIA001

图 4 XACML 策略 IIA001 对应的抽象语法树

IIA001 策略主要针对主体 id、客体 id、动作 id 进行匹配,授权效果为 Permit,采用 deny-overrides 的规则集成算法.由于其动作 id 存在两个:*read* 和 *write*,为了在 PML 规则中同时指定两个动作,可以采用 regexMatch 桩函数进行正则匹配,同时匹配 *read* 和 *write* 两个动作.根据第 3.2 节提出的策略转化算法,可以设计出相应的 XACML 策略规则适配器.通过适配器转换后,IIA001 策略可转换为 PML 模型和规则.其中,PML 模型为:

r=sub,obj,act

p=sub,obj,act,eft

e=some(*where*(*p.eft==allow*)) && !*some*(*where*(*p.eft==deny*))

m=r.sub==p.sub && *r.obj==p.obj* && *regexMatch*(*r.act,p.act*)

PML 规则为

Julius Hibbert, http://medico.com/record/patient/BartSimpson, (*read*)|(*write*), *allow*.

XACML 官方策略示例集共有 374 条策略,本文 XACML 策略适配器已经成功适配其中的 350 条,其他未适配的策略由于需要实现特殊的匹配函数(如复杂字符串操作),在未实现相应 PML 桩函数时暂时无法匹配,但在 PML 中注册相应的桩函数后也可实现.

4.2.2 OpenStack 策略转换

本节验证 OpenStack 策略规则适配器的可用性.以下为 OpenStack Nova 计算组件内置策略的一个片段.

“context_is_admin”: “role: admin”

“admin_or_owner”: “is_admin: True or project_id:%(project_id)s”

“default”: “rule: admin_or_owner”

“compute: get”: “”

“compute: get_all”: “”

“compute: get_all_tenants”: “is_admin: True”

“compute: delete”: “rule: admin_or_owner”

OpenStack 策略主要采用的 ABAC 模型,有很多属性的表达,可以采用 PML 的属性机制来描述.例如,其用户的管理员身份通过 role 和 *is_admin* 两个属性来表达,因此在 Matcher 原语中,可以采用 *r.sub.role*==“*admin*”||*r.sub.is_admin*==true 来描述这两种属于管理员的情形.OpenStack 策略中还有针对属主(owner)的权限,即云资源的所有者对该资源具有访问权限,具体是通过比较主体、客体两者的 *project_id* 属性(即租户 id)是否相同来实

现,因此在 PML 中可以表述为 *r.sub.project_id==r.obj.project_id*.由于 OpenStack 策略将动作划分成了 Admin 动作和 Owner 动作两组,Admin 动作只有管理员能够执行,Owner 动作则可以被管理员和资源所有者执行.因此在 PML 中,可以将 Owner 动作用规则来枚举,即出现在规则中的动作为 Owner 动作,否则则为 Admin 动作.结合上述 *project_id* 属性的比较,则可以给出 Matcher 原语:*r.act==p.act* && *r.sub.project_id==r.obj.project_id*.

通过以上分析,可以设计出相应的 OpenStack 策略规则适配器.通过适配器转换后,上述 Nova 策略实例可转换为 PML 模型和规则.其中,PML 模型为:

r=sub,obj,act

p=act

e=some(*where*(*p.eft==allow*))

m=r.sub.role==“*admin*”||*r.sub.is_admin*==true||(*r.act==p.act* && *r.sub.project_id==r.obj.project_id*)

PML 规则为:

compute: *get*

compute: *get_all*

compute: *delete*

通过分析易得出,OpenStack Nova 内置策略和策略转换后的 PML 策略是等效的.本文还验证了 Glance、Neutron、Cinder 等其他组件的内置策略均可转换为等效的 PML 策略.

4.2.3 IAM 策略转换

本节验证 AWS IAM 策略规则适配器的可用性.IAM 策略综合采用了 RBAC 与 ABAC 两种模型,采用 JSON 格式进行描述.AWS 公有云目前共内置 261 条安全策略,分别应用于 AWS 的众多类型的云服务.以下为针对 EC2(elastic compute cloud)弹性计算服务的 AmazonEC2ReadOnlyAccess 策略,该策略允许附加该策略的用户只读访问 EC2、ELB 弹性负载均衡、AutoScaling 自动扩缩服务、CloudWatch 监控服务的所有资源.

```
“Version”: “2012-10-17”,
    “Statement”: [{
        “Effect”: “Allow”,
        “Action”: “ec2: Describe*”,
        “Resource”: “*”
    },{
        “Effect”: “Allow”,
        “Action”: “elasticloadbalancing: Describe*”,
        “Resource”: “*”
    },{
        “Effect”: “Allow”,
        “Action”: [
            “cloudwatch: ListMetrics”, “cloudwatch: GetMetricStatistics”, “cloudwatch: Describe*”
        ],
        “Resource”: “*”
    },{
        “Effect”: “Allow”,
        “Action”: “autoscaling: Describe*”,
        “Resource”: “*”
    }]
```

设计 AWS IAM 策略适配器,主要需要对客体(resource)、操作(action)、效果(effect)进行适配.

由于 Resource 和 Action 都需要通配符“*”的支持,因此在 PML 中需要采用 keyMatch 桩函数来表达通配符“*”,即 *keyMatch*(*r.obj*,*p.obj*)和 *keyMatch*(*r.act*,*p.act*).由于 IAM 策略采用绑定在主体上的方式进行策略生效操作,因此策略本身并不出现主体,PML 中也无需再描述主体信息.另外,由于 IAM 策略支持 Allow 和 Deny 两种授权效果,并且拒绝优先,因此需要在 PML 的 Effect 原语指定请求允许当且仅当 Matcher 原语的匹配结果存在 Allow 且不存在 Deny,即 *some*(*where*(*p.eft*==*allow*)) && !*some*(*where*(*p.eft*==*deny*)).

通过以上分析,可以设计出相应的 AWS IAM 策略规则适配器.

通过适配器转换后,上述 AmazonEC2ReadOnlyAccess 策略实例可以转换为 PML 模型和规则.其中,PML 模型为:

r=*obj*,*act*

p=*obj*,*act*,*eft*

e=*some*(*where*(*p.eft*==*allow*)) && !*some*(*where*(*p.eft*==*deny*))

m=*keyMatch*(*r.obj*,*p.obj*) && *keyMatch*(*r.act*,*p.act*)

PML 规则为:

, ec2: Describe, allow

, elasticloadbalancing: Describe, allow

*, cloudwatch: ListMetrics, allow

*, cloudwatch: GetMetricStatistics, allow

, cloudwatch: Describe, allow

, autoscaling: Describe, allow

类似于上述 AmazonEC2ReadOnlyAccess 策略,通过分析 AWS 现有的 261 条安全策略的策略转换结果,验证了 AWS 原策略和策略转换后的 PML 策略是等效的.

4.3 策略评估性能

本文采用 OpenStack 云平台的 tempest 测试集对 PML 策略评估性能进行测试.OpenStack 原有安全策略与 PML-EM 的策略评估性能对比结果见表 2.在未引入 AEM 缓存时,PML-EM 运行测试集的开销增加 8.8%,引入 AEM 缓存后开销降低为 4.8%.这是由于时间开销主要是 Patron 与 AEM 之间的通信延迟.而在 AEM 引入缓存后,能够将之前的决策结果缓存起来,同类请求下次直接使用该决策结果,从而不再需要查询 Patron 服务,有效降低了开销.从表 2 中可看出,开销最坏情况发生在 Cinder 组件,不开启缓存的延迟为 15.4%,开启后降低为 7.9%.这是由于 Cinder 组件的每个请求的执行时间都比较短,而 PML-EM 对每个请求引入的延迟是相对较为固定的(约为 77ms),这就导致决策延迟所占比例在 Cinder 上表现得比较明显.PML-EM 的引入,对 Ceilometer 组件的影响不明显.这是由于 Ceilometer 的操作较少,策略也较为简单,因此在策略决策方面开销较小.PML-EM 对每个云平台访问请求引入的平均开销为 77ms,相对于云用户与云平台在 Internet 上的数据传输延迟来说是较小的,因此不会显著影响用户体验.

Table 2 PML evaluation performance (s)

表 2 PML 策略评估性能 (s)

组件	OpenStack 原有策略	PML 策略	开销(%)	PML 策略+缓存	开销增加(%)
Nova	676.24	726.58	7.4	715.43	5.8
Glance	273.10	297.86	9.1	285.76	4.6
Neutron	227.79	246.76	8.3	237.31	4.2
Cinder	146.92	169.55	15.4	158.47	7.9
Heat	394.95	430.68	9.0	415.23	5.1
Ceilometer	643.85	665.13	3.3	651.04	1.1

4.4 侵入性分析

PML-EM 对 OpenStack 原有访问控制机制的侵入性见表 3.由于 OpenStack 原有策略依赖云平台中数量众

多的钩子函数来进行权限检查,这种方式导致访问控制机制与云平台业务逻辑耦合过于紧密.PML-EM 原型系统中的策略决策组件 Patron 以独立组件的形式提供决策服务,与云平台其他组件实现松耦合.AEM 则以通用请求过滤器的形式进一步降低对云平台内部钩子函数的依赖.本文中,AEM 对 OpenStack 其他组件的修改仅限于 3 个 Python 源代码文件.除了代码修改以外,本文还分析了其他类型的修改,如配置修改、PML 模型修改、PML 规则修改等,产生的修改行数(lines of code,简称 LOC)增加约为 0.42%.由于 AEM 是一个请求过滤插件,因此需要对配置进行修改来启用该插件.PML 模型和 PML 规则共同构成 PML 策略,替代 OpenStack 原有的安全策略,并保持策略语义不变.采用 PML-EM 可以去除对原有权限检查钩子的依赖,从而不需要对云平台现有代码文件进行任何修改.在策略管理方面,Patron 为云服务提供商和云用户(策略编写者)提供了策略管理的接口.当 Patron 组件所存储的策略被修改时,为了保持一致性,AEM 的决策缓存会被清除.AEM 的缓存清除功能以 API 的形式提供,与被 AEM 所过滤的云平台组件共用同一个 RESTful 接口,这并不会增加被过滤组件的请求处理的复杂度.因为缓存清除的请求由 AEM 直接进行处理并进行响应,不会进入到被过滤组件的业务逻辑中.所有基于 OpenStack 云平台的原有应用都可以不加修改地在应用 PML-EM 的云平台上运行而不受影响.

Table 3 Modification of PML-EM to OpenStack's original access control mechanism (LOC)

表 3 PML-EM 对 OpenStack 原有访问控制机制的影响(LOC)

组件	OpenStack 原 LOC	PML-EM 影响					LOC 增加(%)
		−安全钩子	+AEM 代码	+AEM 配置	+PML 模型	+PML 规则	
Nova	278 614	484	423	4	16	463	0.33
Glance	78 645	51	423	5	16	49	0.63
Neutron	97 983	132	423	3	16	181	0.64
Cinder	207 442	77	423	6	16	82	0.25
Heat	114 692	86	423	4	16	73	0.45
Ceilometer	59 713	24	423	3	16	6	0.75
Patron	6 055	N/A	N/A	N/A	14	0	N/A

5 总结与展望

本文提出一种基于元模型的访问控制策略描述语言 PML 及其实施机制 PML-EM.PML 支持表达 BLP、RBAC、ABAC 等多种访问控制模型,并且具备多租户、RESTful、桩函数等特性,以支持云平台场景下的访问控制.PML-EM 则实现了策略语言无关性、访问控制模型无关性和程序设计语言无关性,不仅降低了用户编写策略的成本,而且降低了云服务提供商设计、开发访问控制机制的成本.在 OpenStack 上的实验结果表明,PML 策略表达能力较强,支持从 XACML、OpenStack、IAM 等其他策略进行转换.性能上,PML 策略评估的开销平均为 4.8%,从而能够在不影响租户体验的同时,有效提高云平台访问控制的安全性和灵活性.PML-EM 机制的侵入性较小,与云平台原有代码相比,增加约 0.42%.

下一步工作包括:研究 PML 策略之间的冲突检测问题,给出冲突消解方案;进一步探索提高策略转换、策略评估性能的优化方法.

References:

[1] Feng DG, Zhang M, Zhang Y, Xu Z. Study on cloud computing security. Ruan Jian Xue Bao/Journal of Software, 2011,22(1): 71−83 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3958.htm [doi: 10.3724/SP.J.1001.2011.03958]

[2] Lin C, Su WB, Meng K, Liu Q, Liu WD. Cloud computing security: Architecture, mechanism and modeling. Chinese Journal of Computers, 2013,36(9):1765−1784 (in Chinese with English abstract).

[3] Moses T. Extensible Access Control Markup Language (XACML) Version 2.0. Oasis Standard. 2005.

[4] Ribeiro C, Zuquete A, Ferreira P, Guedes P. SPL: An access control language for security policies with complex constraints. In: Proc. of the Network and Distributed System Security Symp. 2001. 89−107.

[5] Damianou N, Dulay N, Lupu E, Sloman M. The ponder policy specification language. In: Proc. of the Workshop on Policies for Distributed Systems and Networks (Policy 2001). 2001. 18−38.

[6] Li NH, Mitchell JC, Winsborough WH. Design of a role-based trust-management framework. In: Proc. of the IEEE Symp. on Security and Privacy. 2002. 114−130.

[7] Han WL, Lei C. A survey on policy languages in network and security management. Computer Networks, 2012,56(1):477−489.

[8] Ferraolo D, Kuhn R. Role-based access control. In: Proc. of the 15th National Computer Security Conf. 1992. 554−563.

[9] Sandhu RS, Coyne EJ, Feinstein HL, Youman CE. Role-based access control models. Computer, 1996,29(2):38−47.

[10] Bao YB, Yin LH, Fang BX, Guo L. Approach of security policy expression and verification based on well-founded semantic. Ruan Jian Xue Bao/Journal of Software, 2012,23(4):912−927 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4023.htm [doi: 10.3724/SP.J.1001.2012.04023]

[11] Bertino E, Jajodia S, Samarati P. Supporting multiple access control policies in database systems. In: Proc. of the IEEE Symp. on Security and Privacy. 1996. 94−107.

[12] Minsky NH, Ungureanu V. Unified support for heterogeneous security policies in distributed systems. In: Proc. of the 7th USENIX Security Symp. 1998. 131−142.

[13] Bell DE, Lapadula LJ. Secure computer systems: Mathematical foundations. MITRE Corporation Report, 1973.

[14] Fielding RT, Taylor RN. Architectural styles and the design of network-based software architectures [Ph.D. Thesis]. University of California, 2000.

[15] Wu R, Zhang X, Ahn G, Sharifi H, Xie H. ACaaS: Access control as a service for iaas cloud. In: Proc. of the Int'l Conf. on Social Computing. 2013. 423−428.

[16] Tang B, Sandhu R. Extending openstack access control with domain trust. In: Proc. of the Int'l Conf. on Network and System Security. 2014. 54−69.

[17] Jin X, Krishnan R, Sandhu R. Role and attribute based collaborative administration of intra-tenant cloud IaaS. In: Proc. of the 10th IEEE Int'l Conf. on Collaborative Computing: Networking, Applications and Worksharing. IEEE, 2014. 261−274.

[18] Kagal L, Finin T, Joshi A. A policy language for a pervasive computing environment. In: Proc. of the Int'l Conf. on Collaborative Computing: Networking, Applications and Worksharing. 2003. 63−74.

[19] Ashley P, Hada S, Karjoth GUN, Powers C, Schunter M. Enterprise privacy authorization language (EPAL). In: Proc. of the IBM Research. 2003. 1−69.

附中文参考文献:

[1] 冯登国,张敏,张妍,徐震.云计算安全研究.软件学报,2011,22(1):71−83. http://www.jos.org.cn/1000-9825/3958.htm [doi: 10.3724/SP.J.1001.2011.03958]

[2] 林闯,苏文博,孟坤,刘渠,刘卫东.云计算安全:架构、机制与模型评价.计算机学报,2013,36(9):1765−1784.

[10] 包义保,殷丽华,方滨兴,郭莉.基于良基语义的安全策略表达与验证方法.软件学报,2012,23(4):912−927. http://www.jos.org.cn/1000-9825/4023.htm [doi: 10.3724/SP.J.1001.2012.04023]

罗杨(1989－),男,河北衡水人,博士,主要研究领域为云计算安全.

吴中海(1968－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为大数据系统与分析,大数据与云安全,嵌入式系统.

沈晴霓(1970－),女,博士,教授,博士生导师,CCF 高级会员,主要研究领域为操作系统与虚拟化安全,云计算和大数据安全与隐私,可信计算.