在Linux安全模块框架授权钩放置的一致性分析

## 2.2 关系验证

图2示出的概念之间的关系。

(1)识别控制操作：查找定义中介接口的操作集，通过该接口访问所有安全性敏感操作。

(2)确定权限需求： 对于每个受控操作，确定必须由LSM钩子授权的授权需求（即策略）。

⑶ 验证完成授权：对于每个受控操作，验证LSM钩子是否对正确的权限需求完成授权。

⑷验证钩子放置清晰度：实施策略操作的受控操作应该可以容易从其授权的钩子中识别。否则，即使对源代码进行很小的改变也可能导致钩子无法操作。

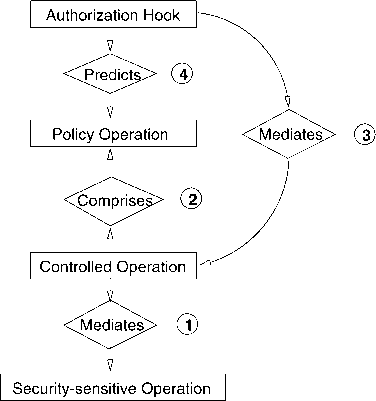


图2 授权概念之间的关系。验证问题是：（1）确定受控操作; （2）确定权限需求; （3）验证完成授权; 和（4）验证钩子放置清晰度。­

其基本思想是，我们确定受控操作及其授权要求，那么我们验证授权钩子是否适当的管理（作为中介）这些受控操作。首先，我们需要确定内核中典型的受控操作。其次，由于受控操作比策略操作（即授权要求）级别更低，我们需要一种方法，通过该方法可以确定每个受控操作的授权要求。第三，我们需要将LSM钩子授权与预期的权限需求进行比较。对内核内授权来说，这些任务很复杂，所以很明显，自动化的支持是必需的。­­­­

受控操作到权限需求的映射不一定是静态的。例如，可以在打开用于读取的文件上执行许多相同的操作，如同用于写入的文件一样。因此，上下文也是将受控操作映射到授权要求的决定性因素。我们的做法必须启用上下文的依赖，以进行有效的管理，从而可以测试受控操作和授权要求之间的预期关系。

## 2.3 相关工作

近日，静态分析来验证安全性已显示出希望。首先，现有的程序分析工具已被用于查找常见的安全错误，如缓冲区溢出和printf漏洞。其次，许多用于寻找安全漏洞的专业工具已经被开发出来，如xgcc，ITS4/RATS，MOPS，MAGIC等。­

静态分析工具基于编程语言的形式属性，因此它们可用于完整的分析（no false negatives，无错误否定）。但是，静态程序验证的计算成本高昂，所以常常对分析模型做一些简化。这些简化可导致更保守的分析（即，more false positives）或某些特性的抽象（即，false negatives）。此外，静态分析工具可能需要大量的代码注释工作，这是构建所需分析模型所必需的。

专业分析工具专注于特定类型的错误。 恩格勒等人。 允许扩展GCC（称为xgcc）来进行源分析，它们称之为元编译。 称为metal的规则语言用于在更高级语言中表达必要的分析注释。 metal规范不是直接注释代码，而是定义引导分析引擎的有限状态自动机。 由于规则匹配多个语句，因此减少了注释工作量。 此工具已发现各种软件错误，包括安全漏洞。

大多数专业分析工具缺乏完整性（即可能导致漏报），但MOPS专门致力于简化分析的规范和完整性。 使用MOPS，安全属性表示为有限状态自动机，程序表示为下推自动机。 未表示数据流，因此忽略别名和值关系。 但是，对于许多分析，仍然可以找到有用的bugs，并且通常可以通过其他方式显示许多数据流关系不存在。­­­

在另一项努力，我们使用一个程序分析工具，CQual ，一种静态查找LSM钩子放置错误的方法。使用GCC分析自动化CQual注释，然后，我们可以执行CQual分析，验证所有受控操作都由至少一个LSM钩子作为中介。 通常，我们还希望验证只有在检查了所需的授权钩子时才运行受控操作。CQual提供了一个可用于定义预期授权的类型点阵，尽管从概念上讲它很复杂。此外，受控操作和授权要求之间关系的上下文依赖性超出了CQual可以处理的范围。

有一个Java静态分析工具，叫 JaBA，已被用来收集对Java受控操作的实际授权。对于我们而言，这种方法有两个缺点：（1）它不分析Linux内核的C代码和（2）它没有提供有关所做的授权是否正确指导。我们实际上定义了从C到JaBA分析概念的转换，并构建了一个原型实现。关于第二点，JaBA确实提供了上下文敏感的控制流图和上下文敏感的数据流图，可用于任何分析。 因此，我们将在第5节中研究这些图形在生成分析日志中的用法。

由于使用这些方法的复杂性，我们发现运行时数据收集帮助我们快速获得准确的数据，这样我们就可以探索可能的分析选项。通过检查收集的数据，我们开发了一种一致性分析方法，我们在本文中描述了该方法，使我们能够确定对受控操作是否进行了适当授权钩子的检查。最终，该方法与我们是否对运行时收集的数据进行一致性分析或通过代码的静态分析无关。 在本文中，我们研究了两种数据收集方法。­­

另一个相关问题是系统认证。从历史上看，橙皮书被用于构建安全操作系统的指导，但现在这已被普通标准所取代。但是，认证任务是临时和费力的，并且通常不能成功地改进常用操作系统的安全性。 Gutmann在他的论文中指出，认证方法，包括形式验证工具，注定要失败，除非它们表示源代码层面的概念。 Gutmann还提倡静态和运行时分析的组合。我们使用的方法与认证的不同之处在于它检查特定错误而不是提供自上而下的保证，即整个系统满足其要求。一个有趣的研究问题是，这种检查的广度和深度是否能提供与认证相当的可信度。与认证不同，随着源代码的发展，可以保持这种信心。

## 3. 解决方案说明

我们在LSM框架运行时分析中依赖的关键是LSM授权挂钩放置在很大程度上是正确的，因此与规范不一致的情况可能表示错误。 例如，如果特定受控操作对同一系统调用的不同（不同时间或者不同上下文？）运行具有不同的授权要求，则将被视为异常。 虽然这种见解并不能保证我们找到所有LSM钩子放置错误（参见第6节），但它使我们能够识别一些错误，并且已经成为工具开发的有价值的指南。

在下面的所有讨论中，我们使用以下假设。 首先，我们相信一个假设，即受控操作中的对象在内核中以类型安全的方式处理。 这不会使我们发现的任何错误无效，但也可能存在其他错误。 其次，我们假设对授权数据类型对象的访问定义了受控操作集（即，中介接口）。 这些数据类型对应于系统调用概念（例如，文件，inode，套接字，skbuffs，ipc消息队列等）。 访问内核数据需要通过这些数据结构。 虽然我们尚未明确验证这一点，但我们已在其他地方进行了更详细的分析。­­

### 3.1 授权一致性

我们首先定义一个受控操作和一组授权要求之间的一致性。­

定义 1 （授权一致性）。 如果在执行受控操作时，受控操作和一组授权要求（即，被授权的策略操作）之间的关系是一致的，则调用与这些授权要求相关联的授权钩子。

我们发现这种形式的一致性并非绝对必要。 受控操作的执行可以在具有不同授权的不同系统调用的上下文中发生。 显然，在这种情况下，授权要求的满足情况将是不同的。­

因此，有必要定义期望一致性的上下文。 一般来说，上下文可以是任意的，但我们的经验是三种类型的上下文很重要：⑴系统调用;⑵具有特定输入的系统调用（例如，标志）; （3）一组特定的受控操作。 在第一种情况下，授权用于整个系统调用（例如，fcntl）。 此类授权适用于系统调用中的所有受控操作。 在第二种情况下，授权依赖于系统调用的一些参数，通常是标志（例如，打开以供读取）。 因此，一些系统调用属于一种情况，一些系统调用属于另一种。 在第三种情况下，受控操作集的出现与它们出现的系统调用无关，需要特定的授权（例如，与访问集所有者字段相关的操作）。 在这些情况下，一致性会忽略系统调用信息。

定义2（执行上下文）。 执行上下文定义了一组执行路径。 执行上下文可以通过（1）系统调用（所有执行）; （2）具有特定参数值（或值的范围）的系统调用; （3）一组受控操作（包括它们的所有路径）来定义。 其他上下文定义是可能的。

我们的解决方案必须支持我们期望持久授权的上下文的描述。 通常，静态分析中的上下文相关数据流指的是区分相同功能的不同输入。 在这种情况下，上下文敏感度要窄得多（仅取决于系统调用），或者可能忽略剩余上下文的大部分（对于受控操作集）。 这种分析需要对静态分析器进行大量注释，并且可能依赖于静态分析器之外的变量（例如，用户提供的标志）。 例如，JaBA完全忽略了原始类型的值，但显然那些可以影响分析。

### 3.2 授权一致性水平

执行上下文通常由许多受控操作组成，因此以相同方式聚合受控操作是有帮助的。

表一，授权一致性水平：名称和授权影响

|  |  |
| --- | --- |
| 级别 | 授权 |
| 系统调用 | 在系统调用的所有控制操作 |
| 系统调用输入 | 在同一个系统调用相同的输入所有控制操作 |
| 数据类型 | 相同的数据类型的对象，所有控制操作 |
| 对象 | 同一对象上的所有控制操作 |
| 成员 | 上相同的数据类型的所有控制操作，访问同一部件，用相同的操作 |
| 函数 | 在相同功能的所有成员一样控制操作 |
| Intrafunction | 同样的受控操作实例 |
| 路径 | 相同的执行路径相同的控制操作实例 |

例如，如果上下文中的所有受控操作具有相同的授权，那么我们可以查看相对于上下文的一致性而不是单个操作。

我们发现，我们可以描述每个受控操作与在特定上下文中通过一组离散值（称为一致性级别）执行时调用的授权钩子之间的一致性。 此外，一致性级别形成如下的总顺序。

定义 3 （一致性级别总顺序）。如果两个不同的受控操作对于级别i的相同值是授权一致的，则它们对于级别j的任何值是授权一致的，其中在一致性级别总顺序中i> j（参见表I）。

如果对同一对象执行了两个不同的受控操作，但它们具有一致的授权，那么这些操作的成员和访问的值不会影响一致性。 例如，如果对特定对象的所有受控操作具有相同的授权要求，则成员访问权限无关紧要。 表I列出了离散一致性级别。 我们将这组级别统称为授权一致性级别。 这些级别包括受控操作执行的各个方面，包括执行它的上下文（系统调用，系统调用输入，函数，函数中的位置，受控操作的路径），执行它的对象（数据类型） 和对象），以及执行的操作（成员和访问）。

一致性级别将受控操作聚合到一致性类中，其中所有受控操作都具有在给定当前放置时调用的相同授权钩子。

定义 4 （一致性类）。两个不同的受控操作属于执行上下文的相同一致性类，如果它们在每次在该上下文中执行时都具有相同的授权钩子。

### 3.3 授权一致性影响

通过授权一致性对受控操作进行分类将受控操作分为两类：⑴已知异常和（2）授权要求需要验证的一致性类。在第一种情况下，我们认为某些授权一致性级别是非法的。我们在下面为这些情况定义不变量。在第二种情况下，我们必须确定执行上下文中每个受控操作的最大一致性级别是否表示可接受的授权要求。例如，如果受控操作属于对象一致性级别的一组受控操作，则表示对象上的所有受控操作都检查了相同的授权。然后，确定这是否正确是手动任务。但是，一致性聚合的数量表示将受控操作划分为具有相同授权的最大大小的类。这些类可以验证授权要求和识别异常分类。

#### 3.1.1 反常现象

表I中最后两行，内部函数和路径，级别的授权一致性始终被认为是异常的。 这些类型的灵敏度意味着函数（函数内）中的执行路径（路径）或位置决定了同一成员的特定受控操作的授权要求。

以下不变量正式表达了我们的路径不一致性。

定义5（路径不一致不变）。

Vc1，c2 e C, e1,e2 e E, (c1 = c2)八（e1 = e2) — ^(c1, e1) = E(c2, e2) (1)

该不变量表明相同的受控操作（c1 = c2）在同一事件中运行（e1 =由系统调用及其输入定义的e2）必须具有相同的授权要求（由函数R定义）。 也就是说，事件内的执行路径不会影响受控操作的授权要求。­

*同样，我们为函数内不一致定义了一个不变量。*

定义6（Intrafunction Inconsistency Invariant，功能不一致不变量）。

Vc1,c2 e C, e1,e2 e E, (F(d) = F(c2))八

(M(c1) = M(c2))八（e1 = e2) — K(c1,e1) = E(c2,e2) (2)

在这种情况下，同一函数中的两个受控操作（由函数F计算）并且进行相同的成员访问（由函数M计算）必须具有相同的授权要求R.

#### 3.3.2 授权一致性类。

授权一致性类。 对于其他情况，我们不能轻易将它们识别为错误。 相反，我们将受控操作划分为其授权一致性类，并确定其授权要求是否正确。

授权一致性类计算如下。 对于从最高（系统调用）开始的每个一致性级别，我们将受控操作划分为一致性类，其中所有受控操作对于一致性级别具有相同的值，然后我们测试该类是否也具有相同的授权。 如果没有，那么我们尝试下一个较低级别并根据两个级别进行分区并再次测试。 这种方法重复进行，直到我们将每个受控操作分配给一致性类。

分类由一致性级别定义。 对于系统调用级别，系统调用的所有受控操作都在一个类中。 对于系统调用输入，将汇总同一系统调用和相同类型输入的所有受控操作（请参阅第3.4节）。 对于数据类型级别，受控操作按系统调用，输入和操作对象的数据类型进行分类。 因此，在分析的每个步骤中创建连续更精细的分区。

如果分类是该类中所有受控操作具有相同授权的第一级，则分类成功（即，x是一致的，其中x是级别）。 请注意，具有相同授权一致性的其他类将被聚合以形成最大大小的类。 创建类后，验证每个类的授权是否正确是一个手动过程。 对于文件系统，类的数量足够小，手动验证是实用的。

例如，考虑读取系统调用。 文件操作是数据类型一致的，因为文件对象上的所有受控操作都有权读取。 手动验证涉及检查文件的读取权限是否足够。 由于读取授权也适用于文件的inode，因此我们将文件的inode标记为已授权读取。 但是，在分类之后，未授权一个inode的受控操作。 它位于不同的对象上，因此inode操作可能是对象一致的。 这是对文件的目录inode的操作，以确定是否应该发送一个读取目录的信号。

除了完全缺失授权之外，识别错误的最常见方式是找到执行重要共同操作的两个分类（即，具有不同授权的两个聚合）。 这种情况发生在fcntl中，其中两个不同的分类（基于不同的系统调用输入）在同一f\_owner字段上运行（参见第4.2.4节）。

### 3.4 必要的数据采集

通过记录系统调用入口/出口/参数，函数入口/出口，受控操作（即对象，数据类型，成员和操作）和授权，我们收集一致性级别的所有必要值。 可以轻松记录所有信息，但需要进一步分析有意义的对象标识符和系统调用输入更改。

在执行期间，通过函数指针引用对象，但这不一定是对象的充分标识。 例如，inode具有在授权中使用的持久标识符（即，设备，inode号）。 因此，对于每种数据类型，我们定义了用于计算其对象标识符的特定方法。 这些标识符用于确定对象的所有操作和授权。

表二。日志记录类型

|  |  |
| --- | --- |
| 记录类型 | 数据 |
| 受控操作  授权  函数入口  函数退出 | 上下文ID控制操作ID OID  上下文ID授权ID OID  上下文ID指令地址  上下文ID |

在系统调用实例中，我们假设在同一个变量中使用的对象具有相同的授权要求。 为了模拟这一点，我们使用第一个受控操作，其中一个对象作为标识符出现。 如果在同一受控操作中首先访问两个对象，则必须将它们分配给同一变量（因为变量在两个受控操作中相同）。 然而，不同的执行路径可能导致首先在不同的受控操作中使用相同的变量。 但是，具有相同授权要求的类的聚合将合并这些情况，因此该假设已被证明是有效的。

系统调用参数几乎在每次调用时都会发生变化，但只有少数参数真正影响授权（例如，打开时的访问标志）。 因此，我们收集参数，但只使用我们发现影响授权要求的参数来进行分区。 我们检查的只有少数系统调用根据其输入参数具有不同的授权，例如open，ioctl和fcntl。 因为基于不同的输入使用不同的授权，所以这些系统调用更复杂，因此更容易出错。