УДК 004.451.004.056

РАЗГРАНИЧЕНИЕ ДОСТУПА И МИНИМИЗАЦИЯ УЩЕРБА ОТ АТАК С ПОМОЩЬЮ СИЛЬНОГО ПРИНЦИПА НАИМЕНЬШИХ ПРИВИЛЕГИЙ

П.С. Бушмакин – студент, лаборатория вычислительных комплексов, кафедра «Автоматизация систем вычислительных комплексов», факультет ВМК, МГУ им. М.В. Ломоносова
E-mail: imtheno@lvk.cs.msu.su
А.В. Сапожников – сотрудник, лаборатория вычислительных комплексов, кафедра «Автоматизация систем вычислительных комплексов», факультет ВМК, МГУ им. М.В. Ломоносова
E-mail: askold@lvk.cs.msu.su
Д.Ю. Гамаюнов – к.ф.-м.н., ст. науч. сотр., лаборатория вычислительных комплексов, кафедра «Автоматизация систем вычислительных комплексов», факультет ВМК, МГУ им. М.В. Ломоносова
E-mail: gamajun@lvk.cs.msu.su

Исследована возможность реализации сильного принципа наименьших привилегий, при котором набор доступных привилегий меняется со временем – по ходу выполнения приложения. Рассмотрен один из вариантов реализации данного принципа в операционной системе семейства Linux для различных классов приложений, в том числе многопоточных.

Ключевые слова: *минимизация ущерба, безопасность операционных систем, формальные модели безопасности.*

In this paper we research the possibility of implementing hard principle of least privilege, when the set of provoleges available for the given application changes over time according to the real observed path of application's execution. We propose implementation of the hard principle of least privileges for the Linux operating system for different types of applications, including multithreaded applications.

Ключевые слова: *attack mitigation, operating systems security, formal security models*

Введение

Повышение гранулярности контроля привилегий приложений во время выполнения и предотвращение эскалации привилегий позволяют блокировать успешное распространение вредоносного программного обеспечения (далее – ВПО) на уровне узла сети. В настоящее время существует большое число механизмов безопасности в распространённых операционных системах, которые пытаются решить эту проблему. Сможет ли вредоносный код исполниться на процессоре атакуемой системы, сможет ли он получить доступ к операционной системе и ценным пользовательским данным – это целиком зависит от того, насколько развиты механизмы защиты операционной системы узла. За последние несколько десятков лет получили развитие традиционные дискреционные модели доступа, впоследствии дополненные механизмами мандатного и ролевого контроля доступа, а также более специфичными механизмами защиты от порчи памяти – защита от исполнения стека, рандомизация адресного пространства и т.д. Кроме того, в современных UNIX-подобных операционных системах существуют механизмы контроля поведения приложений, например, SELinux и AppArmor в Linux.

В случае SELinux контроль доступа основан на принудительной типизации объектов и субъектов и явном описании правил доступа между типами, а также правил присвоения типов новым субъектам или объектам. В случае успешной эксплуатации уязвимости в приложении, находящемся под контролем SELinux, вредоносный код получит доступ только к тем объектам, которые доступны для текущего домена атакованного приложения, а не ко всем ресурсам, доступным пользователю, от имени которого запущен процесс.

Одним из недостатков SELinux и AppArmor является то, что они реализуют лишь базовый принцип наименьших привилегий. Однажды запущенный процесс получает один и тот же набор привилегий, кроме специальных случаев, когда процесс может сам переключать свой текущий контекст безопасности. Но принцип наименьших привилегий можно усилить, если ввести в модель время – предоставлять только те привилегии, которые необходимы процессу для нормального функционирования *в данный момент времени* (в текущем состоянии). Например, мы могли бы разрешить чтение и запись конфигурационных файлов только в фазе инициализации процесса, если в графе потока управления процесса нет других мест, где ему требуется доступ к этим файлам. Пример приложения, в котором можно эффективно применить сильный принцип наименьших привилегий – это приложение, в котором есть фаза аутентификации и авторизации пользователя, после которой все действия выполняются в рамках сессии аутентифицированного пользователя. В этом случае мы можем переключать контекст процесса, в зависимости от того, под каким пользователем аутентифицирована сессия. Подобная политика переключения контекстов реализована в сервере OpenSSH. Идея настоящей работы заключается в том, чтобы обобщить и распространить этот подход на как можно более широкий класс приложений.

Механизмы контроля поведения приложений становятся особенно актуальны, когда мы переходим от стационарного сервера или многопользовательского компьютера к мобильному устройству. Как правило, современные мобильные устройства – это компьютеры общего назначения с полноценной операционной системой и доступом к сети Интернет. При этом зачастую пользователи мобильных устройств не используют механизмы аутентификации и авторизации на доступ к устройству – это сводит на «нет» традиционные механизмы разделения доступа. Кроме того, приложения, устанавливаемые через стандартные источники, такие как Android Market или Apple AppStore, при первой установке запрашивают у пользователя привилегии на доступ к устройствам, ресурсам устройства и сетевым ресурсам, и зачастую запрашиваемый набор привилегий существенно превышает минимально необходимый для работы. Это также снижает эффективность традиционных механизмов контроля доступа. В работах [1, 2] предложены механизмы расширения стандартной модели безопасности для мобильных устройств. В частности, предложено разделять всё множество приложений на классы «пользовательских» (недоверенных) и «корпоративных» (доверенных и критичных). Доступ между различными классами приложений запрещён вне зависимости от того, какие привилегии выдаёт пользователь своим приложениям после их установки через Интернет. Предложенный подход также можно усилить контролем поведения приложений с выявлением внутреннего состояния приложения.

В настоящей работе используются автоматически построенные «модели» приложений в форме альтернирующих автоматов безопасности, которые используются для сопоставления наблюдаемого поведения с идеальным «модельным», и переключения контекста безопасности приложения в зависимости от того, в каком состоянии находится приложение. Подход к мониторингу поведения приложений во время выполнения достаточно новый, и впервые был предложен в контексте безопасности в начале 2000-х годов. Например, в работе [3] Мартинелли и Маттеуччи предложили теоретическую базу для автоматического построения контролирующих операторов – специальных автоматов, которые проверяют и контролируют выполнение условий заданной политики безопасности для приложения. Политика безопасности задана в виде μ-алгебры (расширение пропозициональной модальной логики) надо последовательностью важных с точки зрения безопасности действий. Позже Бауэр и другие авторы в работе [4] предложили язык описания политик Polymer и средство их проверки для программ на языке Java. В настоящее время также ведётся интересное исследование в рамках системы Wedge исследователями Биттау и другими [5], и системы Capsicum исследователями Ватсоном и другими [6]. В этих работах гранулярность контроля достигается за счёт ручного разделения приложения на компоненты с различными требованиями к набору привилегий. В данной работе похожий эффект достигается за счёт использования механизма контрольных точек, который не требует модификации приложения.

Предлагаемый подход

В основе данной работы лежит контроль поведения приложений во время выполнения со стороны операционной системы таким образом, что ядро операционной системы распознаёт внутренние состояния приложения и переключает контексты безопасности в зависимости от наблюдаемой последовательности состояний приложения. В дополнение к контролю системных вызовов и вызовов библиотечных функций помечаются все точки в графе потока управления приложения, в которых происходит передача управления. При каждом проходе контрольной точки, ядро операционной системы принимает решение об изменении или не изменении контекста безопасности.

Контрольные точки позволяют использовать информацию о графе потока управления программы для разметки системных и библиотечных вызовов, выполняемых приложением. Для минимизации активного набора привилегий необходимо разбить весь доступный набор привилегий на несколько подмножеств, так чтобы в каждый момент времени было доступно (активно) только одно из них. Разделение графа потока управления на соответствующие подобласти выполняется автоматизировано.

Задача контроля поведения приложения на основе сильного принципа наименьших привилегий может быть разбито на следующие подзадачи:

П о д з а д а ч а 1 . *Разбиение программы.* Первая подзадача заключается в том, чтобы разбить программу на набор линейных блоков, связанных по управлению и разметить их контрольными точками. Пусть задана программа и исходный набор привилегий. Необходимо восстановить граф потока управления программы и разбить его на множество непересекающихся блоков таким образом, чтобы набор различных привилегий в каждом блоке был строго меньше, чем исходный набор привилегий. Для каждого блока мы помечаем вход и все выходы контрольными точками.

П о д з а д а ч а 2 . *Построение модели нормального поведения.* Пусть задана программа, размеченная контрольными точками. Необходимо построить контролирующий и распознающий автомат, который принимает все нормальные последовательности системных вызовов и контрольных точек, и не принимает последовательности, которые не принадлежат множеству нормального поведения.

П о д з а д а ч а 3 . *Контроль поведения во время выполнения.* Пусть задана программа, размеченная контрольными точками, набор привилегий для каждого блока программы, и модель нормального поведения в виде распознающего альтернирующего автомата. Необходимо реализовать подсистему уровня ядра ОС, которая собирает трассу реального поведения программы, передаёт её контролирующему автомату и использует выход автомата для переключения действующего набора привилегий программы (рис. 1).

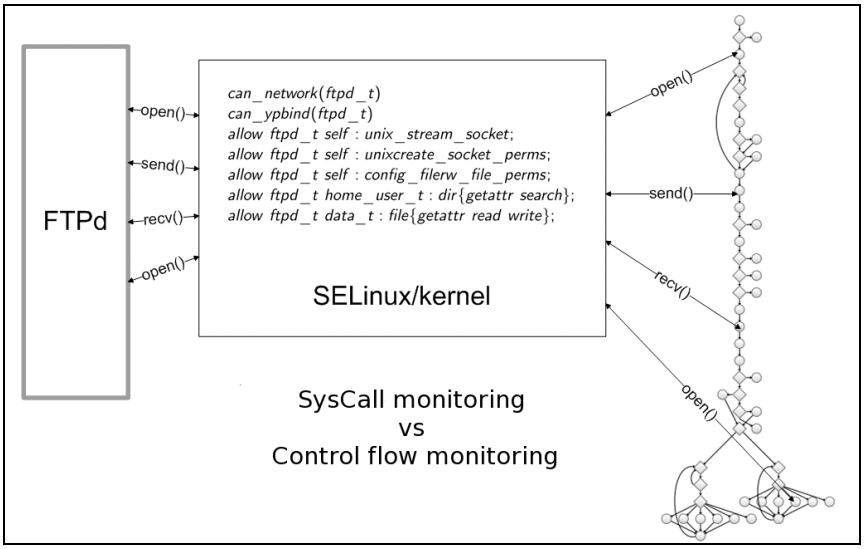

Рис. 1. Контроль поведения на примере сервера FTP

Разбиение программы

Первый шаг при решении задачи разбиения программы – построение графа потока управления (CFG) по исходному коду программы, частично решение данной задачи представлено в работе [9]. В данной работе в качестве средства для построения и работы с графом потока управления использована библиотека CIL [7]. С помощью данной библиотеки строится граф потока управления для всей программы, с выделением базовых блоков в этих графах потока управления, а также граф вызовов функций.

После построения CFG выполняется его разметка маркерами привилегий – системными вызовами и их параметрами, которые встречаются в реальных трассах выполнения данной программы. С точки зрения рассматриваемой в данной работе модели системный вызов с конкретным набором параметров является отдельной привилегией, которую можно разрешить или запретить во время выполнения приложения в контролируемой SELinux среде.

Введем следующие обозначения:

S = (s1 , s2 , ..., sn) – трасса;

si = (syscall, stack) – символ трассы;

syscall – системный вызов;

stack – стек, соответствующий этому системному вызову;

arg – аргумент системного вызова syscall, т.е. перечень ресурсы с которыми работает системный вызов.

З а д а ч а – необходимо поставить в соответствие базовому блоку программы системный вызов и его аргументы.

С помощью библиотеки libunwind [8] на основании stack можно получить строку кода программы (вершину CFG), которой соответствует системный вызов, а далее по этой строке найти базовый блок в CFG, которому принадлежит эта вершина. Найденный базовый блок помечается системным вызовом и его аргументами.

Алгоритм разметки:

Require: *trace* – трасса выполнения программы

1: **for all** *s* in *trace* **do**

2: {Проверяем нужно ли этот системный вызов переносить на граф}

3: **if** (test(*s.syscall*)) **then**

4: policy = getPolycyInfo(*s.syscall, arg*) {Запоминаем информацию, которой будем размечать граф}

5: markGraph(*s.stack, policy*) {Размечаем граф}

6: **end if**

7: rememberSpecInfo(s) {Запоминаем специфическую информацию, содержащуюся в этом системном вызове. Например, системный вызов open возвращает дескриптор открытого файла, дальнейшие операции над этим файлом, например, чтение, используют дескриптор, а не имя. Но для занесения информации в политику нужно имя файла, а не дескриптор, поэтому на этом этапе из системного вызова open извлекается информация о соответствии имени файла и дескриптора.}

8: **end for**

После разметки CFG программы маркерами привилегий выполняется разбиение CFG на блоки, и свёртка графа до некоторого заранее заданного числа блоков. Строим начальное разбиение: помеченные узлы преобразуются в блоки. Если получившееся число блоков меньше или равно требуемому, то считаем это разбиение финальным. Иначе применяется следующий алгоритм.

Рассмотрим множество U – множество самых внутренних помеченных узлов:

Require: U – множество самых внутренних помеченных узлов; U_p –множество помеченных узлов; n – требуемое число блоков; m – максимальное число символов в алфавите блока

1: blockCount = k {Число блоков в начальном разбиении.}

2: B=Ø

3: {Пока число узлов больше требуемого}
4: **while** (*blockCount* > *n*) **do**
5: **for all** *vertex* in *U* **do**
6: **if** (symbolCounts(*vertex*) < *m*) **then** {Смотрим не достигнуто ли в этом узле максимальное число символов}
7: newVertex = makeCompression(vertex) {К узлу из множества *U* применяется алгоритм свертки помеченного узла}
8: **if** (newVertex) **then** {Если свертка была произведена}
9: replace(vertex, newVertex, *U*, U_p) {Узел был свернут, в множествах *U* и U_p этот узел заменяется на новый}
10: **else**
11: U.remove(vertex, *U*, U_p) {Если узел не был свернут, то он удаляется из *U*, U_p}
12: bl = makeBlock(vertex)
13: B.add(vertex)
14: **end if**
15: **end if**
16: blockCount = count(|*B*|∩|*makeBlocks*(U_p)|) {Подсчитываем число блоков в программе}
17: **if** (blockCount > *n*) **then**
18: **break**
19: **end if**
20: **end for**
21: **end while**
22: B = *B* ∩ *makeBlocks*(U_p)

На рис. 2 представлены некоторые шаги алгоритма и конечный результат для графа потока управления простого FTP-демона.

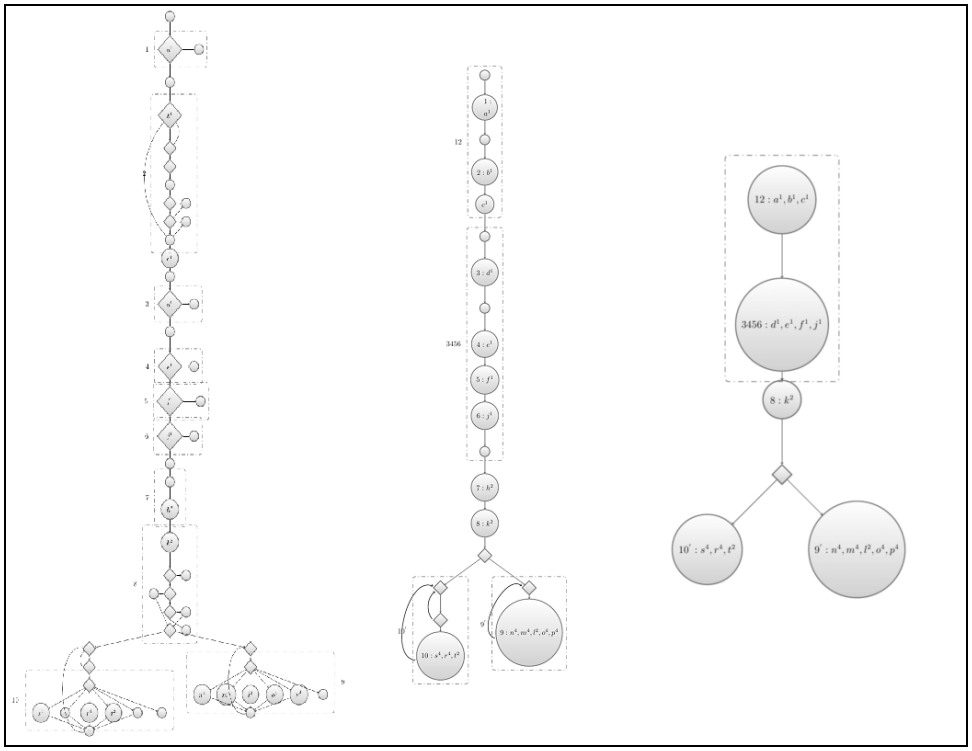

Рис. 2. Разбиение на блоки, разметка и свёртка графа потока управления простого FTP-демона

В итоговом разбиении программы выполняется разметка контрольными точками, которые устанавливаются на входе в блок и на каждом выходе из блока.

Построение модели нормального поведения

Д а н о : программа П с внедренным множеством контрольных точек *Chp*. Множество примеров нормального поведения {*smpl*}, примеры нормального поведения содержат символы из *Chp*.

Т р е б у е т с я : построить описание нормального поведения программы такое, что любой пример из {*smpl*} удовлетворяет этому описанию, множество примеров, удовлетворяющих описанию, строго шире, чем {*smpl*}, и при этом в расширенном множестве путей программы нет аномальных (которые противоречат известным последовательностям контрольных точек в программе).

Программа П представлена своим графом потока управления, где вершины размечены операторами программы, а ребра отражают передачу управления в программе. Нужно построить множество V вершин, размеченное шагами программы (обычными и шагами-сообщениями о прохождении контрольных точек), и множество дуг, соединяющее эти вершины, и показать, что построенное дерево включается в нормальное поведение программы. Алгоритм построения такого дерева будет описывать операцию расширения набора нормальных примеров до некоторого множества, всегда удовлетворяющего известным последовательностям контрольных точек.

Построение такой древовидной структуры будем проводить в три этапа. На первом этапе работы алгоритма построим ациклический ориентированный граф общего вида. Этот граф будет задавать все множество историй {*trace*}, которые соответствуют возможным путям передачи управления в CFG программы. Отметим, что среди этих историй могут быть такие, которые не входят во множество нормального поведения ($trace' \notin ВН^{norm}$). На втором этапе из этого графа выберем только те истории, в которых контрольные точки следуют в «допустимом порядке». На третьем этапе из набора отдельных историй получим дерево, которое и будет искомым описанием нормального поведения.

Алгоритм построения описания нормального поведения:

1: (V, E) = (∅, ∅); {Создаем пустой граф}.

2: add(v^{root}, (V, E)); {Добавляем корневую вершину к графу}.

3: {Построим L «слоев» вершин, соответствующих шагам-сообщениям о прохождении контрольных точек. Здесь L = max (len($order_{checkp}$(smpl))) – максимальная длина последовательности контрольных точек. На каждом l-м «слое» вершины строящегося графа будут размечены шагами-сообщениями о прохождении контрольных точек chp_i, встречающихся на l-й позиции в некоторой последовательности $order_{checkp}$ (smpl).

4: **for** $l = 1 \ldots L$ **do**

5: add({$v^{chp,l}$ }, (V, E)); {добавим к графу l-й слой размеченных вершин}.

6: **end for**

7: add($v^{root} \rightarrow v^{chp,l}$, (V, E)); {соединим корневую вершину дугами с вершинами первого «слоя»}

8: {Все пары вершин (chp_i , chp_j) на соседних «слоях» соединим всеми возможными путями}

9: **for all** упорядоченных пар (chp_i , chp_j) **do**

10: **for** $l = 1..L-1$ **do**

11: **for** all $s_1, s_2, \ldots, s_k \in Usubt_{smpl}$ (chp_i , chp_j) **do**

12: add(v^{s1}, (V, E)); add(v^{s2}, (V, E)); ... add(v^{sk} , (V, E)); {построим цепочку вершин, разметим их шагами}

13: add($v^{chp,l} \rightarrow v^{s1} \rightarrow v^{s2} \rightarrow \ldots \rightarrow v^{sk} \rightarrow v^{chp,l+1}$, (V, E)) {начало и конец каждой такой цепочки соединим соответственно с вершинами $v^{chp,l}$ и $v^{chp,j}$, промежуточные вершины соединим надлежащими дугами}

14: **end for**

15: **if** $\varepsilon \in Usubt_{smpl}$ (chp_i , chp_j) **then**

16: add($v^{chpi} \rightarrow v^{chpj}$, (V, E)); {соединим эти вершины дугой напрямую}

17: **end if**

18: **end for**

19: **end for**

20: {Получили ациклический граф (V, E) общего вида, в котором между каждой парой вершин, размеченных контрольными точками и находящимися на соседних слоях, существуют все возможные пути, соответствующие последовательностям шагов из U $subt_{smpl}$ (chp_i , chp_j)}.
21: {Рекурсивным поиском в глубину обойдем все возможные пути в этом ориентированном графе, и выберем те из них, в которых «контрольные точки» следуют в правильном порядке}.
22: (V', E') = (∅, ∅) {создаем новый граф}
23: **for all** *trace* = *rcrLookup.next*((V, E)); **do**
24: {Находим историю с допустимым порядком контрольных точек}
25: **if** $order_{checkp}$(*trace*) = chp_1 , chp_2 , ..., chp_k и ∃smpl : $order_{checkp}$ (smpl) = chp_1,chp_2, ..., chp_k, ... **then**
26: add(*trace*, (V', E')); {Добавим эту трассу в новый граф, присоединив ее в вершине v^{root}}
27: **end if**
28: **end for**
29: (V'', E'') = minimize((V', E')); {Приведем полученный граф к древовидному виду, последовательно объединяя вершины с наименьшими общими префиксами}

Построенное дерево (V'', E'') с размеченными вершинами является искомым множеством нормального поведения программы, которое используется в дальнейшем для контроля поведения приложений во время выполнения.

Контроль поведения во время выполнения

Каждое приложение (а именно: бинарный образ) разбивается на участки кода, каждый из которых требует определённого набора привилегий для исполнения. На все входы и выходы из этих участков устанавливаются контрольные точки, при попадании потока исполнения на которые контекст меняется на ассоциированный с участком контекст SELinux. Саму систему контроля поведения можно разбить на следующие компоненты:

1. Подсистема уровня ядра, отслеживающая попадание приложения на контрольные точки и осуществляющая смену контекстов SELinux;
2. Интерфейс конфигурации ядерной подсистемы.

Установка контрольных точек в процессе производится с помощью системы Uprobes. Эта система предоставляет возможность автоматизированной отладки пользовательских приложений; в качестве отладчика может выступать как отдельный процесс, так и управляющие потоки ядра Linux.

Uprobes позволяет устанавливать точки останова на любой адрес в виртуальном пространстве процесса и регистрировать функции, обрабатывающие события, связанные с данными точками (событие – это попадание исполнения на точку останова). Регистрация производится с помощью функции

uprobes_register(*struct uprobe *u*),

здесь struct uprobe содержит информацию об устанавливаемой точке останова:

u->pid – идентификатор процесса;
u->vaddr – адрес установки;
u->handler – функция обработчик.

При помощи данного механизма со стороны ядра Linux выполняется разметка контрольными точками тех процессов, которые специально помечены как отслеживаемые системой на основе сильного принципа наименьших привилегий. При прохождении потока исполнения приложения через контрольную точку исполнение приостанавливается, и управление передаётся в специальный обработчик на стороне ядра. В реализованной системе обработчик проверяет соответствие наблюдаемой трассы ожидаемому поведению потока программы, и переключает действующий профиль SELinux в соответствии с ранее построенной моделью нормального поведения и разбиением программы на блоки.

Эксперименты

Для тестирования работоспособности созданной реализации был выбран многопоточный FTP-сервер, использующий дочерние потоки для работы с пользовательскими запросами. В приложение специально была внедрена уязвимость, позволяющая пользователю прочитать любой файл в виртуальной файловой системе, с которой работает FTP-сервер.

Стандартная версия SELinux не позволяет предотвратить эксплуатации этой уязвимости, и неспособна как-либо ограничить возможности нарушителя в рамках политики ftpd_restr_t (действующей для указанного сервера). В результате злоумышленник сможет прочитать некоторые файлы конфигурации, которыми пользуется приложение, а также файлы, используемые подсистемой журналирования FTP-сервера (запись основных событий происходящих при работе приложения). Исходная политика была разбита на две: ftp_restr_no_etc_t для блока кода, считывающего файлы по запросу пользователей, и ftpd_restr_t для остальных блоков. Контрольные точки были установлены на все входы и выходы из функции обрабатывающей запросы пользователя, т.е. на cmd_retr. Вызов этой функции происходит в одном из потоков типа worker (граф жизненного цикла потоков исполнения в FTP-сервере на рис. 3).

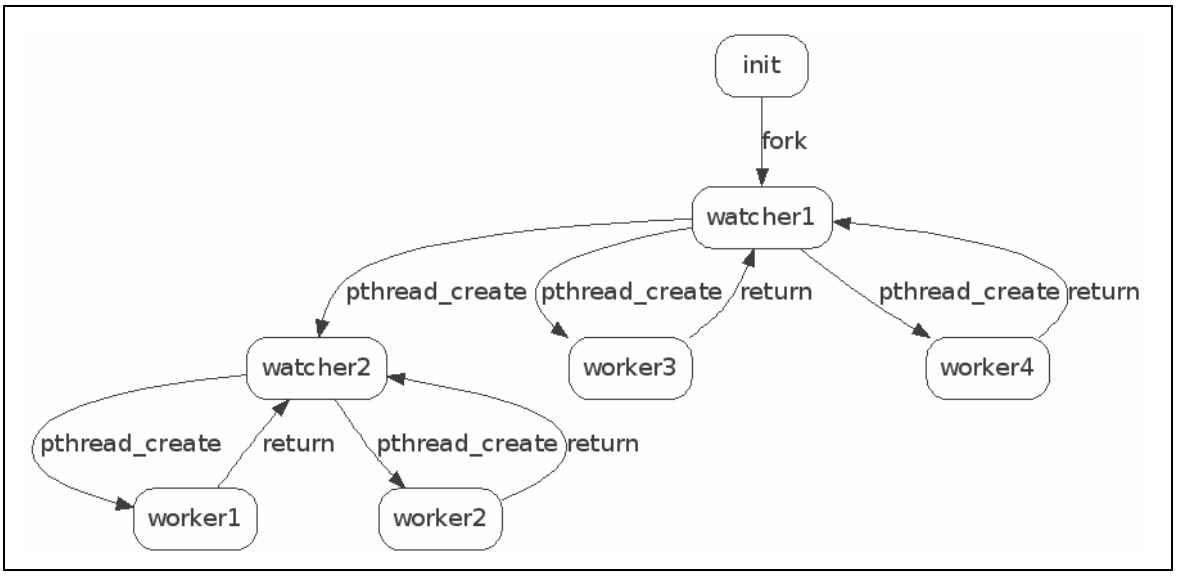

Рис. 3. Граф потоков исполнения FTP-сервера

В случае успешной эксплуатации уязвимости файл /etc/pftpd.conf успешно загружается с сервера. При этом SELinux активен, приложение pftpd работает в контексте unconfined_u:system_r:ftpd_restr_t:s0-s0:c0.c1023.

Листинг взаимодействия с сервером при успешной атаке:

```
Connected to localhost.
220 Peter's Anonymous FTP server (pftpd 1.0 at Dec 22 2011 10:46:03) ready.
Name (localhost:root): anonymous
331 Guest login ok; use your e-mail address as password.
Password:
230 Login OK.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get /../../etc/pftpd.conf /root/stolen
local: /root/stolen remote: /../../etc/pftpd.conf
200 PORT command successful.
150 Opening BINARY mode data connection for /home/ftp/../../etc/pftpd.conf.
```

В рамках базовой политики SELinux тип ftpd_restr_t позволяет выполнить операцию скачки файла, имеющего тип ftpd_restr_etc_t, этим типом обладает файл /etc/pftpd.conf:

```
root@debian:~# ls -Z /etc/pftpd.conf
system_u:object_r:ftpd_restr_etc_t:s0 /etc/pftpd.conf
```

Теперь включим систему, реализующую сильный принцип наименьших привилегий, и повторим атаку на FTP-сервер:

```
ftp> get /../../etc/pftpd.conf /root/stolen
local: /root/stolen remote: /../../etc/pftpd.conf
200 PORT command successful.
```

Сессия «зависает» после вызова передачи файла, и через какое-то время завершается. При этом в журнале SELinux отображается факт блокирования чтения файла /etc/proftpd.conf:

```
Apr 8 02:19:49 debian kernel: [6398.836454] type = 1400 audit(1336861189.736:9): avc: denied { read } for pid= 1481 comm= "pftpd" name = "pftpd.conf" dev = sda2 ino = 130618 scontext = unconfined_u : system_r : ftp_restr_no_etc_t : s0 tcontext = system_u : object_r : ftpd_restr_etc_t : s0 tclass = file
```

Таким образом, динамическое переключение контекстов позволило заблокировать чтение конфигурационного файла после успешного запуска FTP-сервера, когда ему больше не требуется данная привилегия.

Заключение

Предложеный подход к реализации сильного принципа наименьших привилегий на основе динамического изменения множества доступных приложению привилегий реализован для частного случая – операционной системы Linux и встроенной в неё системы безопасности SELinux. Для успешной реализации сильного принципа наименьших привилегий ядру операционной системы требуется информация о внутренней структуре и логике исполнения программ. В рассмотренной системе контроля поведение это достигается при помощи статико-динамического анализа кода программы, построения CFG и его разметки контрольными точками на основе примеров трасс нормального выполнения программы. Размеченный CFG и набор трасс используется для автоматического построения описания нормального поведения программы, которое в дальнейшем может быть использовано для динамического контроля доступных приложению привилегий.

Реализованный в настоящее время прототип системы для операционной системы Linux позволяет разделять внутренние состояния последовательных программ, а также разделять контексты безопасности отдельных потоков исполнения для параллельных (многопоточных программ), использующих библиотеку NPTL (Native Posix Threads Library). Прототип испытан на тестовом приложении – многопоточном FTP-сервере, в который искусственно внесена уязвимость, позволяющая читать файлы из каталога /etc, эксплуатация которой не блокируется исходным профилем SELinux для FTP сервера, но успешно блокируется прототипом системы контроля поведения.

Представляется перспективным дальнейшее развитие описанного подхода и реализованного прототипа в следующих направлениях:

1. Полная автоматизация анализа программы и генерации модели нормального поведения вместе с частными профилями SELinux для внутренних состояний программы;

2. Адаптация разработанного прототипа для SE Android – порта SELinux на ОС Android, с целью построения защищенных мобильных платформ, в которых критичные к безопасности кода и данных приложения могут функционировать без влияния сторонних недоверенных приложений.

🕮 Литература

1. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, Ahmad-Reza Sadeghi In: 19th Annual Network & Distributed System Security Symposium (NDSS), Februar 2012.

2. Towards Taming Privilege-Escalation Attacks on Android. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry. In: 19th Annual Network & Distributed System Security Symposium (NDSS), Februar 2012.
3. *Martinelli F. and Matteucci I.* Through modeling to synthesis of security automata // In Proc. of ENTCS STM06, 2006.
4. *Bauer L., Ligatti J., and Walker D.* Composing expressive run-time security policies // ACM Transactions on Software Engineering and Methodology. V. 18. №3. Article 9. 2009.
5. *Bittau A., Marchenko P., Handley M., and Karp B.* Wedge: Splitting Applications into Reduced-Privilege Compartments // In Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation (2008). P. 309–322.
6. *Watson R.N.M., Anderson J., Laurie B.* Capsicum: practical capabilities for UNIX // In Proc. of the USENIX Security. 2010.
7. CIL – Infrastructure for C Program Analysis and Transformation [HTML] (http://hal.cs.berkeley.edu/cil/)
8. The libunwind project [HTML] (http://www.nongnu.org/libunwind/)
9. *Гамаюнов Д.Ю., Горнак Т.А., Сапожников А.В., Сахаров Ф.В, Торощин Э.С.* Гранулярный контроль безопасности поведения приложений со стороны ядра Linux // Информационно-методический журнал «Защита информации. Инсайд». №4. СПб., 2010. С. 54–58.

Поступила 25 мая 2012 г.

HARD PRINCIPLE OF LEAST PRIVILEGE FOR FINE-GRAINED ACCESS CONTROL AND COMPUTER ATTACK MITIGATION

P.S. Bushmakin, A.V. Sapozhnikov, D.Yu. Gamayunov

The principle of least privilege is widely recognized in the field of operation systems security, and it states that application should be granted with only minimal set of privileges, neccessary for normal execution. In this paper we research the possibility of implementing hard principle of least privilege, when the set of provoleges available for the given application changes over time according to the real observed path of application's execution. We propose implementation of the hard principle of least privileges for the Linux operating system for different types of applications, including multithreaded applications, where privilege control should be done intependently for each of the application's threads. The proposed attack mitigation architecture looks promicing for mobile security especially.