УДК 004.413

Логический анализ корректности конфигурирования межсетевых экранов

© В.В. Девятков, Мьо Тан Тун

МГТУ им Н.Э. Баумана, Москва, 105005, Россия

Статья посвящена статическому анализу поиска ошибок конфигурирования межсетевых экранов (брандмауэров). В отличие от известных работ для моделирования поведения брандмауэров предлагается использовать не списки управления доступом ACL (ACL – Access Control Lists), а процессные модели, выразительные возможности которых гораздо шире, а теория более развита, что позволяет описывать гораздо более сложные модели программ конфигурирования. Статический анализ поиска ошибок предлагается осуществлять методами логического программирования как доказательство или вывод свойств процессов конфигурирования, что является гораздо более изящным, полным и не имеющим ограничений подходом проверки корректности конфигурирования брандмауэров. Требования (свойства) корректности предлагается формализовать на языке модальной логики. Переход от этого формального описания свойств предлагается алгоритмически формировать как цель в языке ПРОЛОГ. В статье приводятся примеры логических программ проверки корректности конфигурирования брандмауэров на языке ПРОЛОГ и результаты их испытаний.

Ключевые слова: *межсетевой экран, брандмауэр, процесс, статический анализ корректности, язык логического программирования ПРОЛОГ.*

Введение. Сетевой экран или брандмауэр – широко известное средство защиты сетей. Для того чтобы обеспечить требуемую защиту, брандмауэр должен быть соответствующим образом настроен, или, как говорят, конфигурирован. К сожалению, конфигурирование может быть сопряжено с ошибками, которые делают даже опытные администраторы, что приводит к снижению уровня защиты сети и проникновению в сеть нежелательных пакетов. Как отмечается в работе [1], изучение 37 брандмауэров показало, что в каждом из них имеются ошибки, делающие сети уязвимыми для атак. В работе [2] было проанализировано много реальных ошибок конфигурирования, результатом которых было проникновение в защищаемую систему вредоносных программ. Безошибочное конфигурирование брандмауэра является непростой задачей. Эта задача усложняется еще и тем, что в большинстве случаев конфигурирование связано с написанием его правил на языке низкого уровня, описание на котором трудно анализировать на предмет соответствия уровня защиты проводимой политике. Особенно это трудно делать при большом числе правил конфигурирования. Более того, когда крупная организация разворачивает большое количество брандмауэров на множестве сетевых

компонентов в условиях динамической маршрутизации, то пакет от одного и того же источника к одному и тому же приемнику может анализироваться различными брандмауэрами в разное время. Это еще больше усложняет процедуру поиска ошибок и анализа соответствия проводимой политике защиты.

В работе [3] для поиска ошибок конфигурирования предлагается использовать так называемый статический анализ, суть которого состоит в том, что описание на языке конфигурирования рассматривается как программа конфигурирования, которая анализируется без ее запуска непосредственно на языке ее описания. Это довольно распространенный и давно известный подход [4]. Статический анализ программ позволяет:

а) выявлять и устранять уязвимость брандмауэра до его развертывания в сети;

б) осуществлять нахождение всех типов ошибок на всех возможных путях выполнения операторов (правил) программы брандмауэра;

в) в случае использования многих брандмауэров одновременно выявлять уязвимости защиты путем анализа взаимодействия брандмауэров, не требуя при этом их реального развертывания.

Достоинством работы [3] является то, что авторы применили статический анализ для поиска реальных ошибок в программах конфигурирования и создали инструментарий FIREMAN (FIREwall Modeling and ANalysis), с помощью которого можно найти два класса ошибок конфигурирования:

1) ошибки нарушения политики безопасности пользователя (например, если брандмауэр позволяет входящему пакету достигать TCP порта 80 на внутреннем хосте, то это, как правило, является ошибкой нарушения политики безопасности в большинстве сетей);

2) ошибки противоречивости правил в программе конфигурирования (например, если некоторое правило запрещает доступ какому-либо пакету, а выполняемое ранее правило его разрешает, то это является ошибкой противоречивости указанных правил). Авторы статьи [3] предложили классификацию ошибок программ конфигурирования и алгоритм статического анализа этих ошибок для различных видов использования брандмауэров.

Настоящая статья также посвящена статическому анализу поиска ошибок конфигурирования, который будем называть проверкой корректности конфигурирования. Однако, в отличие от [3], в данной статье для моделирования поведения брандмауэров предлагается использовать не списки управления доступом ACL (ACL – Access Control Lists), а процессные модели, выразительные возможности ко-

торых гораздо шире, а теория более развита, что позволяет описывать гораздо более сложные модели программ конфигурирования. Типы ошибок в программах конфигурирования предлагается описывать, в отличие от работы [3], формально как свойства процессных моделей программ конфигурирования на языке временной модальной логики. Процессные модели в дальнейшем будем называть просто процессами. Процессы позволяют гораздо более четко и полно классифицировать и описывать типы ошибок. Статический анализ поиска ошибок предлагается осуществлять методами логического программирования как доказательство или вывод свойств процессов конфигурирования, что является гораздо более изящным, полным и не имеющим ограничений подходом проверки корректности конфигурирования брандмауэров.

Межсетевые экраны или брандмауэры различных производителей могут существенно различаться языками конфигурирования. Однако, если не учитывать эти различия, то обычно конфигурирование любого сетевого экрана состоит в составлении списков управления доступом (access control lists: ACL [3]), используемых для управления процессом доступа. Брандмауэр обычно имеет несколько интерфейсов. Каждый интерфейс конфигурируется различными списками управления доступом. Каждый список управления доступом состоит из списка правил. Отдельное правило представляется парой $<p$, $action>$, где p – пакет, над которым совершается действие, *action*. Каждый пакет, поступающий на сетевой экран обрабатывается последовательно каждым правилом в порядке записи их сверху вниз. Если пакет p удовлетворяет некоторому свойству $R(p)$, то над ним совершается действие *action* и он к следующему правилу не направляется. Если пакет p не удовлетворяет этому свойству, то действие *action* над ним не совершается и он направляется к следующему правилу, переходя, таким образом, от правила к правилу до тех пор, пока не достигнет последнего правила. Свойство $R(p)$ можно рассматривать как некоторый предикат, который истинен, если упоминаемое свойство имеет место и ложен в противном случае. Списки управления доступом как язык конфигурирования брандмауэров с точки зрения проверки корректности конфигурирования, на наш взгляд, обладают рядом следующих недостатков.

Списки управления доступом имеют недостаточно проработанный формальный синтаксис, что не позволяет однозначно понимать некоторые описания на этом языке.

Последовательная природа обработки и передачи пакетов от правила к правилу в порядке записи правил декларируется, но никак не описывается средствами языка, что делает затруднительным фор-

мальное метаописание свойств корректности собственно списков управления доступом.

Переходы между правилами различных списков требуют от вызываемого списка после его выполнения возврата в ту же точку вызывающего списка, откуда произошел вызов, что ограничивает возможности проверок корректности и повышение уровня эффективности вычислений.

Описание на языке списков управления доступом использует только свойства пакетов, но не содержит формализмов описания отдельных элементов этих пакетов и их совокупностей, что в случае моделирования динамической проверки на корректность конфигурирования требует генерации этих элементов внешними по отношению к языку средствами.

Использование процессных моделей как языков описания поведения брандмауэров позволяет устранить указанные недостатки.

Процессные модели конфигурирования брандмауэров. Каждый процесс имеет алфавит восприятий и реакций $A = \{a_1, a_2,..., a_m\}$. Каждый символ a этого алфавита именует некоторый объект, получаемый (воспринимаемый) процессом из внешней среды (восприятие процесса), выдаваемый процессом во внешнюю среду (внешняя реакция процесса) или объект, используемый процессом для внутренних нужд (внутренняя реакция процесса). Процессы действуют, воспринимая, порождая для внутреннего употребления или выдавая наружу объекты с соответствующими именами. Для того, чтобы различать типы действий, будем использовать следующие обозначения: $?a$ – для восприятий, $!a$ – для внешних реакций, b – для внутренних реакций.

Нитью a^* будем называть кортеж (конечный или бесконечный) действий $a^* = a_0a_1a_2\ldots\ a_{m-2}a_{m-1}$. Выполнением нити процессом P называется последовательность выполнения ее действий в определенном порядке слева направо. Символом e обозначается пустое действие. Нить, состоящая из единственного пустого действия, называется пустой нитью. Процессом P называется множество нитей S, которые он может выполнять. Поведением процесса P называется порядок выполнения множества этих нитей.

Обозначим $?A$ множество всех восприятий некоторого процесса P, включая пустое восприятие $?e$, $!A$ – конечное множество всех внешних реакций процесса P, включая пустую внешнюю реакцию $!e$, S – множество всех нитей, выполняемых процессом P, таких что $S \subseteq ?A^* \times !A$, где $?a^*$ – нить, состоящая только из восприятий (нить восприятий) процесса P, $?A^*$ – множество всех нитей $?a^*$ в алфавите $?A$, φ^* – функция на множестве $?A^*$, которая ставит в соответ-

ствие каждой нити $?a^* \in ?A^*$ внешнюю реакцию из множества $!A$. Процесс P, выполняющий множество нитей $?a^*\varphi^*(?a^*) \in S$, будем обозначать $P(S)$. Значение функции φ^* на тех нитях множества $?A^*$, на которых эта функция не определена и на тех нитях, которые этому множеству не принадлежат, считается равным $!e$. Множество нитей $?a^*$ таких, что $\{?a^* \in ?A^* \mid ?a^*\varphi^*(?a^*) \in S\}$ будем обозначать $?S$.

Популярным языком представления процессов является язык графов переходов. Для построения графа переходов процесса считается, что после каждого его восприятия, в том числе пустого, процесс осуществляет внутреннюю реакцию или, как говорят, переходит во внутреннее состояние ожидания следующего восприятия. Находясь в этом внутреннем состоянии, процесс может порождать внешнюю реакцию, в том числе пустую. В графе переходов каждое состояние процесса изображается кружком, внутрь которого помещается символ этого состояния; каждому восприятию соответствует стрелка, соединяющая состояния этого перехода. Начальное состояние выделяется двойным кружком. На рис. 1 показан граф переходов некоторого процесса *P*.

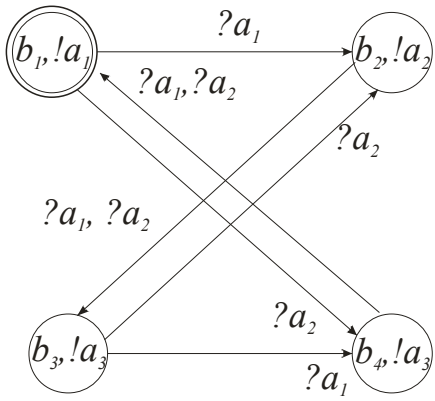

Рис. 1. Граф переходов процесса *P*

Граф переходов процесса позволяет компактно описывать небольшое множество нитей, в том числе бесконечных. В этих условиях естественным кажется описывать процесс непосредственно его графом переходов. Но при большой размерности графа такое описание становится громоздким и ненаглядным. Альтернативой этому является использование адекватных графу канонических

процессных выражений, позволяющих легко переходить от графа к этим выражениям и наоборот.

Так для графа переходов на рис. 1 каноническими процессными выражениями, адекватными этому графу, будут следующие.

$$P \triangleq \begin{pmatrix} b_1 = ?e, \\ b_1 = b_4 ? a_1, \\ b_1 = b_4 ? a_2, \\ b_2 = b_1 ? a_1, \\ b_2 = b_3 ? a_2, \\ b_3 = b_2 ? a_1, \\ b_3 = b_2 ? a_2, \\ b_4 = b_1 ? a_2, \\ b_4 = b_3 ? a_1, \\ !a_1 = b_1, \\ !a_2 = b_2, \\ !a_3 = b_3 \,|\, b_4 . \end{pmatrix}$$

В этих выражениях каждое внутреннее процессное выражение вида $b_i = b_j ? a_k$ задает один переход из состояния b_j в состояние b_i в результате восприятия $?a_k$. Выражение $!a_i = b_j$ задает реакцию $!a_i$ после перехода процесса в состояние b_j.

Очевидно, что если исходным описанием процесса являются канонические процессные выражения, то переход от этих выражений к графу переходов процесса и наоборот очевиден. Канонические процессные выражения могут быть рекурсивными.

Простые процессные модели для отдельного брандмауэра. В случае использования вместо списков управления доступом процессных моделей восприятиями являются выражения ?*R*(*p*) или ?*notR*(*p*), которые будем записывать соответственно просто как ?(*p*) или ?*not*(*p*), где *p* – пакеты адресов. Внешними реакциями являются выражения !*action*, где *action* – действие.

Пример простой списковой модели, взятый из статьи [3], показан на рис. 2.

Простая процессная модель поведения брандмауэра, представленная каноническими процессными выражениями и соответствующая простой списковой модели, показанной на рис. 2, представлена на рис. 3.

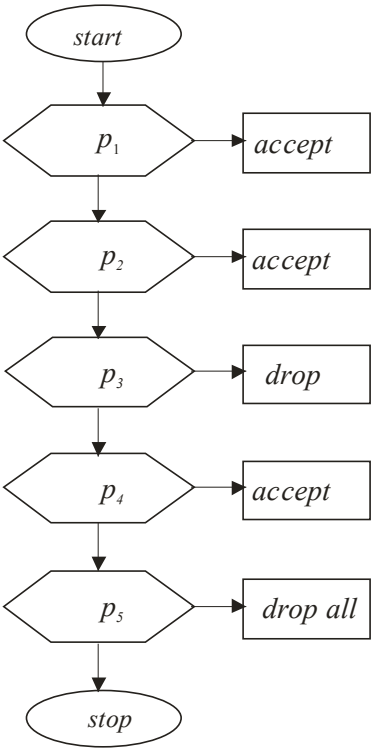

Рис. 2. Простая списковая модель

$$
P \triangleq \left(\begin{array}{ll}
b_1 = ?e, & \\
b_2 = b_1\,?\,not(p_1), & \\
b_3 = b_2\,?\,not(p_2), & \\
b_4 = b_3\,?\,not(p_3), b_2 = b_7(p_1), & \\
b_5 = b_4\,?\,not(p_4), b_3 = b_8(p_2), & \\
b_6 = b_5\,?\,not(p_5), b_4 = b_9(p_3), & \\
b_7 = b_1\,?(p_1), & b_5 = b_{10}(p_4), \\
b_8 = b_2\,?(p_2), & b_6 = b_{11}(p_5), \\
b_9 = b_3\,?(p_3), & \\
b_{10} = b_4\,?(p_4), & !accept(p_1) = b_7, \\
b_{11} = b_5\,?(p_5), & !accept(p_2) = b_8, \\
!e = b_1, & !drop(p_3), = b_9, \\
!e = b_2, & !accept(p_4) = b_{10}, \\
!e = b_3, & !drop\_all(p_5) = b_{11}. \\
!e = b_4, & \\
!e = b_5, & \\
!stop = b_6, &
\end{array}\right)
$$

Рис. 3. Процесс, соответствующий простой списковой модели на рис. 2

Сложные процессные модели для отдельного брандмауэра. Сложная процессная модель поведения брандмауэра, представленная каноническими процессными выражениями и соответствующая сложной списковой модели, показанной на рис. 4, приведена на рис. 5.

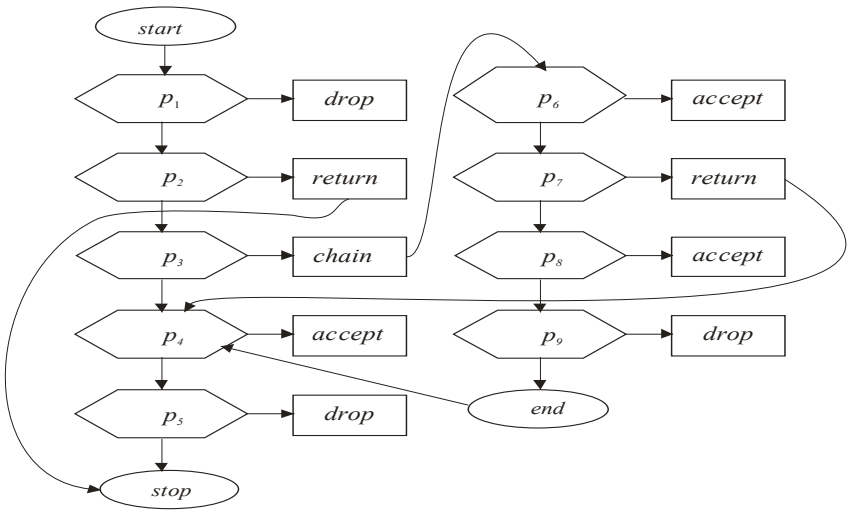

Рис. 4. Сложная списковая модель

Основное отличие сложной списковой модели от простой – это наличие дополнительных действий, позволяющих осуществлять переходы от одной простой списковой модели к другой и обратно. Аналогичные возможности появляются в сложной процессной модели.

Сетевые процессные модели. В типичной сетевой среде несколько брандмауэров чаще всего распределяются по сети и конфигурируются независимо друг от друга. В этом случае защищенность сети зависит от правильности настройки всех сетевых экранов таким образом, чтобы поведение одних сетевых экранов не понижало уровень защищенности, обеспечиваемый другими экранами. Например, если речь идет о защите взаимодействия двух VPN (Virtual Private Network), то независимо от количества сетевых экранов, обеспечивающих защиту этого взаимодействия, уровень защиты должен быть на требуемом уровне. То же самое относится и к взаимодействию Интернета с доверенными внутренними сетями.

Рассмотрим простой пример сетевой среды (рис. 6), содержащей внутреннюю сеть N предприятия, взаимодействующую с Интернетом посредством услуг двух провайдеров $П_1$ и $П_2$ через демилитаризованную зону D. Между каждым провайдером и демилитаризованной зоной расположены соответственно сетевые экраны P_{11} и P_{12}. Сервис, обеспечиваемый демилитаризованной зоной (почта, интернет), в силу своей публичности, более уязвим для атак, по сравнению с внутренней сетью. Поэтому внутренняя сеть дополнительно защищается сетевыми экранами. На рис. 6 приведено два таких экрана P_{21} и P_{22}.

$P \triangleq$
$b_1 = ?e,$
$b_2 = b_1 ?not(p_1),$
$b_7 = b_1 ?(p_1),$
$b_3 = b_2 ?not(p_2),$
$b_8 = b_2 ?(p_2),$
$b_6 = b_8 ?(return),$
$b_4 = b_3 ?not(p_3),$
$b_9 = b_3 ?(p_3),$
$b_5 = b_4 ?not(p_4),$
$b_{10} = b_4 ?(p_4),$
$b_6 = b_5 ?not(p_5),$
$b_{11} = b_5 ?(p_5),$
$b_{12} = b_9 ?(chain),$
$b_{13} = b_{12} ?not(p_6),$
$b_{17} = b_{12} ?(p_6),$
$b_{14} = b_{13} ?not(p_7),$
$b_{18} = b_{13} ?(p_7),$
$b_4 = b_{18} ?(return),$
$b_{15} = b_{14} ?not(p_8),$
$b_{19} = b_{14} ?(p_8),$
$b_{16} = b_{15} ?not(p_8),$
$b_{20} = b_{15} ?(p_8),$
$b_4 = b_{16} ?(end),$
$!stop = b_6,$
$!drop = b_7,$
$!return = b_8,$
$!chain = b_9,$
$!accept = b_{10},$
$!drop = b_{11},$
$!accept = b_{17},$
$!return = b_{18},$
$!accept = b_{19},$
$!drop = b_{20},$
$!end = b_{16}.$

Рис. 5. Сложная процессная модель поведения брандмауэра

Существует несколько путей из Интернета во внутреннюю сеть предприятия. Эффект защиты зависит от пути, по которому проходит пакет. Пакеты не выбирают самостоятельно пути своего перемещения, и в различное время одни и те же пакеты могут проходить по различным путям. В этих условиях сетевые экраны должны действовать согласовано. Недопустимо, чтобы одни и те же пакеты одним сетевым экраном пропускались при выборе пути через него, а другим нет.

Типичные некорректности конфигурирования сетевых экранов. В этом разделе рассмотрим типичные некорректности конфигурирования сетевых экранов. При рассмотрении некорректностей будем

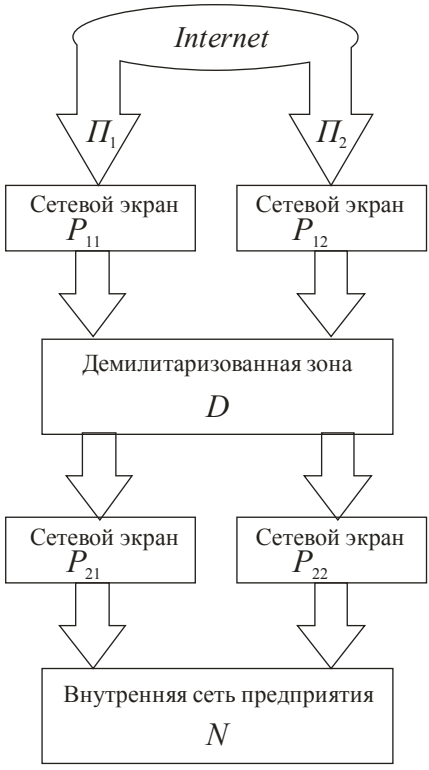

Рис 6. Пример сетевой среды с сетевыми экранами

следовать классификации этих некорректностей, предложенной в работе [3]. Главное, чему будем уделять внимание при рассмотрении некорректностей – это их формальное описание на языке временной модальной логики условий (свойств) некорректности, для того чтобы можно было использовать эти описания для формальной проверки некорректностей в процессных моделях. Для этого введем следующий формат описания пакетов, используемый в процессных моделях.

$< пакет > \rightarrow < протокол >$
$< IP-адрес\ источника >,$
$< порт\_источника >,$
$< IP-адрес\_назначения >,$
$< порт\_назаначения >;$

$< протокол > \rightarrow tcp \mid udp \mid ftp;$

$< IP-адрес\ источника > \rightarrow < число > . < число > . < число > . < число > / < число 1 >;$

$< IP-адрес\ назназначения > \rightarrow < число > . < число > . < число > . < число > / < число 1 >;$

$< число > \rightarrow 0 \mid 1 \mid \ldots\ldots \mid 255;$

$< число 1 > \rightarrow 0 \mid 1 \mid \ldots\ldots \mid 32;$

$< порт\_источника > \rightarrow any \mid < число 2 >;$

$< порт\_назаначения > \rightarrow any \mid < число 2 >;$

$< число 2 > \rightarrow 0 \mid 1 \mid \ldots\ldots \mid 65536;$

Некорректности сетевых экранов. Некорректности в описаниях сетевых экранов могут встречаться во всех трех рассмотренных типах процессных моделей: простых, сложных и сетевых. Настоящий раздел посвящен принципам формального описания этих некорректностей.

Некорректности однопроцессных моделей. Простейший случай затенения возникает, когда действия двух правил в списке, одно из которых предшествует другому правилу, являются противоположными по смыслу, но применяются к одному и тому же пакету *p*. Ясно, что если предшествующее правило разрешает принять этот пакет, а последующее запрещает его прием, то пакет будет принят. И наоборот, если предшествующее правило запрещает принимать этот пакет, а последующее разрешает его прием, то пакет не будет принят. Понятно, что такая ситуация, хотя и разрешается однозначно, является ошибкой, поскольку неизвестно, чем руководствовался автор такого конфигурирования. Возникает большое сомнение в том, что он сделал это сознательно. Скорее всего, это ошибка и ее надо выявлять. Используя временную модальную логику и канонические процессные выражения, одно из правил корректности выглядит следующим образом:

$$\Box[!accept(p))] \supset \neg\Diamond[(!drop(p))].$$

Смысл этого правила следующий: всегда, когда в некотором текущем состоянии совершается действие $!accept$, в которое процесс перешел из некоторого состояния в результате получения (восприятия) пакета p, то никогда в будущем не должно быть другого состояния, в которое процесс перешел из некоторого состояния в результате получения (восприятия) того же пакета p и в котором совершается действие $drop$.

Исключение возникает, когда действия двух правил в списке, одно из которых предшествует другому правилу, являются противоположными по смыслу, но пакет предшествующего правила является подмножеством последующего. Ясно, что если предшествующее правило разрешает принять этот пакет, а последующее запрещает его прием, то предшествующий пакет, входящий в последующий, будет принят. И наоборот, если предшествующее правило запрещает принимать пакет, а последующее разрешает его прием, то пакет, входящий в последующий, не будет принят. Понятно, что такая ситуация, разрешается однозначно, но непонятно, является ли она ошибкой или допущена сознательно, но найти эту ситуацию всегда имеет смысл. Используя временную модальную логику и канонические процессные выражения, правило корректности, истинность которого гарантирует наличие исключения, выглядит следующим образом:

$$\Box[!accept(p_1) \supset \neg\Diamond(!drop(p_2) \wedge (p_1 \subset p_2))] \vee$$

$$\vee\Box[!drop(p_1) \supset \neg\Diamond(!accept(p_2) \wedge (p_1 \subset p_2))].$$

Пересечение возникает, когда действия двух правил в списке, одно из которых предшествует другому правилу, являются противоположными по смыслу, но пакет предшествующего правила пересекается с последующим. Как и в предыдущем случае, ясно, что если предшествующее правило разрешает принять пресечение, а последующее запрещает его прием, то пересечение будет принято. И наоборот, если предшествующее правило запрещает принимать пересечение, а последующее разрешает его прием, то пакет, входящий в последующий, не будет принят. Понятно также, что такая ситуация, разрешается однозначно, но, как и в предыдущем случае, непонятно, является ли она ошибкой или допущена сознательно, но указать на нее необходимо. Используя временную модальную логику и канонические процессные выражения, правило корректности, истинность которого гарантирует отсутствие исключения, выглядит следующим образом:

$$\Box[!accept(p_1) \supset \neg\Diamond((!drop(p_2)) \wedge (p_1 \cap p_2 \neq \varnothing))] \vee$$

$$\vee\Box[!drop(p_1) \supset \neg\Diamond((!accept(p_2)) \wedge (p_1 \cap p_2 \neq \varnothing))].$$

Главными критериями *оптимальности*, как правило, являются: число необходимых правил (процессных выражений) и время проверки при фильтрации пакетов. Понятно, что эти критерии взаимосвязаны, хотя эта взаимосвязь и не очевидна. В однопроцессных моделях сетевых экранов для каждого состояния b вводится не более одного выражения вида $!action = b$ и не более двух выражений вида $b_j = b_i ?(p)$ или $b_j = b_i ?not(p)$, где $b_i \neq b_j$. Если считать, что процесс имеет n состояний, то, исходя из этого, можно заключить, что оценка числа выражений в однопроцессных моделях в зависимости от числа его состояний линейна и равна O(3n). Если считать, что память, необходимая для хранения одного процессного выражения и время его вычисления пропорциональны числу символов l в процессном выражении, то оценки времени вычислений и требуемой памяти для однопроцессорных экранов одинаковы и равны O(3ln). Как правило, $l<<n$, и поэтому главным фактором оптимальности является число состояний n. Для оптимизации числа состояний используется известное [5] отношение R на множестве $?A^*$, определяемое следующим образом: $?a_1^* R ?a_2^*$, если $\varphi^*(?a_1^* ?a^*) = \varphi^*(?a_2^* ?a^*)$ для всех $?a^*$ таких

что $a_1^* ? a^* \varphi^* (a_1^* ? a^*) \in S$, $a_1^* ? a^* \varphi^* (a_2^* ? a^*) \in S$. Отношение R является отношением эквивалентности [5] и разбивает множество $? A^*$ на классы эквивалентности $Q_1, Q_2, ..., Q_{k+1}$, где Q_1 – класс эквивалентности отношения R, которому принадлежит восприятие $? e$. Число этих классов совпадает с числом состояний процесса. В работе [5] алгоритм оптимизации числа состояний (числа процессных выражений) непосредственно по заданным исходным процессным выражениям. Этот алгоритм может быть непосредственно использован оптимизации процесса, описывающего поведение сетевого экрана.

Избыточность какого-либо состояния b_j, достижимого из состояния b_i, является следствием того, что в состоянии b_j совершается то же самое действие, что и в состоянии b_i, и пакет этого состояния b_i включает пакет состояния b_j. Используя временную модальную логику и канонические процессные выражения, правило отсутствия избыточности какого-либо состояния b_j может быть записано следующим образом:

$$\Box[!action_1(p_1) \supset \neg\Diamond((!action_2(p_2) \wedge (!action_1 = !action_2) \wedge (p_2 \subseteq p_1))].$$

Некорректности многопроцессных моделей. Различные сетевые экраны могут обрабатывать одни и те же пакеты по-разному. Как и в случае одного экрана, это может не быть ошибкой, но требует выявления. Когда несколько экранов соединены друг с другом, подвергаются анализу пакеты всех этих экранов. Результат работы экранов, начинающих свою работу позже других, зависит от предшествующей работы связанных с ними экранов. Конфигурирование должно выполняться с учетом этой зависимости. Так, например, на рис. 6 показаны четыре экрана P_{11}, P_{12}, P_{21}, P_{22}. Работе экранов P_{21}, P_{22} предшествует работа экранов P_{11}, P_{12}. Пакеты после обработки экранами P_{11}, P_{12} могут попадать как на экран P_{21}, так и на экран P_{22}.

Затенение между сетевыми экранами возникает тогда, когда действия двух правил, одно из которых принадлежит предшествующему процессу, а другое последующему, являются противоположными по смыслу, но применяются к одному и тому же пакету *p*. Ясно, как и в случае одного экрана, что если предшествующее правило разрешает принять этот пакет, а последующее запрещает его прием, то пакет будет принят. И наоборот, если предшествующее правило запрещает принимать этот пакет, а последующее разрешает его прием, то пакет не будет принят. Понятно, что такая ситуация, хотя и разрешается однозначно, является ошибкой, поскольку неизвестно, чем руководствовался автор такого конфигурирования. Как и прежде, возникает

большое сомнение в том, что он сделал это сознательно. Скорее всего, это ошибка и ее надо выявлять.

Покажем, что затенение между сетевыми экранами можно свести к случаю одного экрана, композируя определенным образом процессы экранов.

Пусть даны два процесса P_1 и P_2, причем процесс P_1 выполняется перед процессом P_2, то есть имеем процесс $P = P_1 \big| P_2$. Каждый из процессов P_1 и P_2 имеют соответственно начальные состояния $b_1^1 = ?e$, $b_1^2 = ?e$ и конечные состояния. $b_i^1 = b_{i-1}^1 ? not(p)$, $b_j^2 = b_{j-1}^2 not(p)$, такие что $!stop = b_i^1$, $!stop = b_j^2$. Введем операцию «◦» композиции последовательно работающих процессов $P_1 \big| P_2$, описывающих последовательную работу каких-либо двух экранов. Она состоит в следующем. В процессе P_1 заменяем выражение $b_i^1 = b_{i-1}^1 ? not(p)$ на выражение $b_1^2 = b_{i-1}^1 ? not(p)$ и удаляем выражение $!stop = b_i^1$. В процессе P_2 удаляем выражение $b_1^2 = ?e$. В результате получим процесс $P = P_1 \circ P_2$, выполнение которого эквивалентно выполнению процесса $P = P_1 \big| P_2$. Например, на рис. 7 показан процесс $P = P_1 \big| P_2$, а на рис. 8 – процесс $P = P_1 \circ P_2$.

$$P = P_1 \big| P_2,$$

$P_1 \triangleq$	$P_2 \triangleq$
$b_1^1 = ?e,$	$b_1^2 = ?e,$
$b_2^1 = b_1^1 ? not(p_1),$	$b_2^2 = b_1^2 ? not(p_1),$
$b_7^1 = b_1^1 ?(p_1),$	$b_7^2 = b_1^2 ?(p_1),$
$b_3^1 = b_2^1 ? not(p_2),$	$b_3^2 = b_2^2 ? not(p_2),$
$b_8^1 = b_2^1 ?(p_2),$	$b_8^2 = b_2^2 ?(p_2),$
$b_4^1 = b_3^1 ? not(p_3),$	$b_4^2 = b_3^2 ? not(p_3),$
$b_9^1 = b_3^1 ?(p_3),$	$b_9^2 = b_3^2 (p_3),$
$b_5^1 = b_4^1 ? not(p_4),$	$b_5^2 = b_4^2 ? not(p_4),$
$b_{10}^1 = b_4^1 ?(p_4),$	$b_{10}^2 = b_4^2 ?(p_4),$
$b_6^1 = b_5^1 ? not(p_5),$	$b_6^2 = b_5^2 not(p_5),$
$b_{11}^1 = b_5^1 ?(p_5),$	$b_{11}^2 = b_5^2 ?(p_5),$
$!accept = b_7^1,$	$!accept = b_7^2,$
$!accept = b_8^1,$	$!accept = b_8^2,$
$!drop = b_9^1,$	$!drop = b_9^2,$
$!accept = b_{10}^1,$	$!accept = b_{10}^2,$
$!drop\_all = b_{11}^1,$	$!drop\_all = b_{11}^2,$
$!stop = b_6^1,$	$!stop = b_6^2.$

Рис. 7. Процесс $P = P_1 \big| P_2$

$$P = P_1 \circ P_2,$$

$P_1 \triangleq$	$P_2 \triangleq$
$b_1^1 = ?e,$	
$b_2^1 = b_1^1 ? not(p_1),$	$b_2^2 = b_1^2 ? not(p_1),$
$b_7^1 = b_1^1 ?(p_1),$	$b_7^2 = b_1^2 ?(p_1),$
$b_3^1 = b_2^1 ? not(p_2),$	$b_3^2 = b_2^2 ? not(p_2),$
$b_8^1 = b_2^1 ?(p_2),$	$b_8^2 = b_2^2 ?(p_2),$
$b_4^1 = b_3^1 ? not(p_3),$	$b_4^2 = b_3^2 ? not(p_3),$
$b_9^1 = b_3^1 ?(p_3),$	$b_9^2 = b_3^2 (p_3),$
$b_5^1 = b_4^1 ? not(p_4),$	$b_5^2 = b_4^2 ? not(p_4),$
$b_{10}^1 = b_4^1 ?(p_4),$	$b_{10}^2 = b_4^2 ?(p_4),$
$b_1^2 = b_5^1 ? not(p_5),$	$b_6^2 = b_5^2 not(p_5),$
$b_{11}^1 = b_5^1 ?(p_5),$	$b_{11}^2 = b_5^2 ?(p_5),$
$!accept = b_7^1,$	$!accept = b_7^2,$
$!accept = b_8^1,$	$!accept = b_8^2,$
$!drop = b_9^1,$	$!drop = b_9^2,$
$!accept = b_{10}^1,$	$!accept = b_{10}^2,$
$!drop\_all = b_{11}^1,$	$!drop\_all = b_{11}^2,$
	$!stop = b_6^2.$

Рис. 8. Процесс $P = P_1 \circ P_2$

Таким образом, анализ затенения между двумя сетевыми экранами P_1, P_2 сводится к анализу затенения одного процесса $P = P_1 \circ P_2$. Оче-

видно, что для выявления всех затенений примера на рис. 6 придется отдельно подвергнуть анализу четыре процесса: $P_{11} \circ P_{21}$, $P_{11} \circ P_{22}$, $P_{12} \circ P_{21}$, $P_{12} \circ P_{21}$, $P_{12} \circ P_{22}$.

Сетевое проникновение. Как показано выше, в случае наличия нескольких экранов возникает возможность прохождения пакетов по различным путям, что может привести к проникновению запрещенных пакетов в закрытые для них зоны. В этом случае нет другого выхода, кроме как анализировать пути возможного перемещения пакетов и смотреть, нет ли среди них таких, которые не должны проникать в запрещенные зоны. Для этого, как и в предыдущем случае, придется построить все процессы, защищающие эти пути, затем задать пакеты, которые не должны проникать в запретные для них зоны, и проверить, нет ли для каждого из них процесса, который позволяет им это сделать. Если обозначить множество пакетов, которым запрещено проникать из некоторой зоны S в некоторую зону E как $p(S,E)$, множество процессов, защищающих пути движения пакетов из зоны S в зону E как $P(S,E)$, факт поступления на процесс P пакета p как $p.P$, то условие отсутствия сетевого проникновения пакета p для всех процессов $P \in P(S,E))$ и всех пакетов $p \in p(S,E))$ выражается следующей формулой модальной логики:

$$\Box[p.P \supset \Diamond(!drop(p) \wedge \neg \blacklozenge(!accept(p)].$$

Смысл следующий: всегда, когда процесс P получает пакет p, то в будущем этот процесс должен перейти в некоторое состояние, в котором совершается действие *drop*, запрещающее пропускать этот пакет и перед этим не должно быть другого состояния, в которое процесс перешел из некоторого состояния в результате получения (восприятия) того же пакета p и в котором совершается действие *accept*.

Принципы перехода от процессного описания модели конфигурирования сетевого экрана к описанию на языке ПРОЛОГ и поиск некорректностей. Переход от процессного описания конфигурирования сетевого экрана к описанию на языке ПРОЛОГ включает следующие шаги:

- построение по процессному описанию модели конфигурирования логической программы на ПРОЛОГЕ (от начального раздела до раздела GOAL);
- построение по описанию на языке модальной логики раздела GOAL;
- проверка корректности.

Построение по процессному описанию модели конфигурирования логической программы на ПРОЛОГЕ до раздела GOAL. Принципы перехода от процессного описания к программе на ПРОЛОГЕ

рассмотрим на примере простой процессной модели, приведенной на рис. 3. В соответствии с форматом описания пакетов (см. раздел 3) в процессных моделях будем полагать, что для рис. 3 заданы следующие пакеты:

p1= accept tcp 10.0.0.0/8 any
p2= accept tcp any any
p3= deny tcp 192.168.1.1/32 any
p4= deny udp 10.1.1.64/26 any
p5= accept udp 10.1.1.26/32 any
p6= accept http 192.168.1.25/16 any
p7= deny ftp 192.168.3.1/15 any
p8= drop tcp 10.0.0./8 any

В соответствии с рис. 3 для каждого процессного выражения в разделе *clauses* формируется факт. Например, для процессного выражения *b7 =b1?(p1)* формируется факт *transition(state_action (state(b1), action(empty)), p(bit(truth), protocol(tcp), ip_source(10,1,1,0,25), port_source(_), ip_destination(_,_,_,_), port_destination(_)), state_action (state(b7), action(accept)))*, означающий, что из состояния *b1* осуществляется переход в состояние *b*7 в результате поступления пакета p_1, над которым осуществляется действие accept. Аналогично формируются остальные факты и сопутствующие им описания в разделах domain и predicates. В результате получаем логическую программу

p=p(bit, protocol, ip_source, port_source, ip_destination, port_destination)
bit=bit(symbol)
protocol= protocol(symbol)
ip_source= ip_source(n,n,n,n,n)
port_source= port_source (n)
ip_destination = ip_destination(n,n,n,n,n)
port_destination= port_destination (n)
n=integer
state_action= state_action(state, action)
state= state(symbol)
action= action(symbol)
Среди каждой области правил фильтра следующим образом:
protocol – Имя протоколов
ip_source – ip-адрес источника
port_source – порт-источника
ip_destination – ip-адрес назначения
port_destination – порт- назначения

predicates
nondeterm transition(state_action, p, state_action)
clauses

transition(state_action (state(b1), action(empty)), p(bit(truth), protocol(tcp), ip_source(10,1,1,0,25), port_source(_), ip_destination(_,_,_,_,_), port_destination(_)), state_action (state(b7), action(accept))).
transition(state_action (state(b2), action(empty)), p(bit(false), protocol(tcp), ip_source(10,1,1,0,25), port_source(_), ip_destination(_,_,_,_,_), port_destination(_)), state_action (state(b8), action(empty))).
transition(state_action (state(b3), action(empty)),a(bit(truth), protocol(udp), ip_source(_,_,_,_,_), port_source (_), ip_destination(192,168,1,0,24), port_destination(_)), state_action (state(b9), action(accept))).
transition(state_action (state(b4), action(empty)), p(bit(false), protocol(udp), ip_source(_,_,_,_,_), port_source (_), ip_destination(192,168,1,0,24), port_destination(_)),state_action (state(b10), action(empty))).
transition(state_action (state(b5), action(empty)), p(bit(truth), protocol(tcp), ip_source(10,1,1,128,25), port_source (_), ip_destination(_,_,_,_,_), port_destination(_)), state_action (state(b11), action(deny))).
accessible(B1, [X], B2) :- transition(B1, X, B2).
accessible(B1, [X|Rest], B2) :- transition(B1, X, B3), accessible(B3, Rest, B2).

Построение по описанию на языке модальной логики раздела GOAL. В рамках настоящей работы не ставилось цели описания на языке модальной логики всех свойств или требований, которым должны удовлетворять межсетевые экраны. Они могут быть самыми различными и достаточно подробно перечислены в литературных источниках [6,7]. Принципы использования модальной логики для описания указанных свойств также хорошо известны [8]. Например, если после начала выполнения процесса *P* на рис. 3 не должно быть ни одного затенения, то это требование можно выразить простой модальной формулой $\Box[!accept(p))] \supset \neg\Diamond[(!drop(p))]$ (см. раздел 3.1.1.1). Это означает, что должен существовать хотя бы один путь, ведущий из какого-либо текущего, в котором над пакетом *p* совершается действие *accept* в другое состояние, в котором над этим же пакетом совершается действие *drop*. На языке ПРОЛОГ это можно выразить в виде следующего раздела *goal* (цели):

goal
accessible (state_action (state(X), action(accept)), [p(bit(truth), protocol(tcp), ip_source(N1,N2,N3,N4,_),port_source (_), ip_destination(_,_,_,_,_), port_destination(_))], state_action (state(X), action(_))),
accessible(state_action (state(Y), action(drop)),[p(bit(false), protocol(_), ip_source(N1,N2,N3,N4,_), port_source (_), ip_destination(_,_,_,_,_), port_destination(_)),state_action (state(Y), action(_))), X< >Y.

Проверка затенения для конфигурирования сетевого экрана. В результате работы полученной по процессному описанию логической программы на ПРОЛОГе при заданной цели получим результат

$$N1=10,\ N2=1,\ N3=1,\ N4=0,\ C=b8,\ B=b2,$$

означающий, что в состоянии действия состояния, *b8* затеняет действие состояния *b2*.

Заключение. Предложен новый статический метод для проверки конфигурации брандмауэров, основанный на использовании процессных моделей и языка модальной логики для описания требований к их корректности. Метод основан на символической проверке моделей с языка логического программирования ПРОЛОГ. По сравнению с другими, этот метод является масштабируемым и реализует полный охват прохождения всех возможных IP-пакетов в различных конфигурациях сетевых экранов. Предлагаемый метод является полезным и практичным не только для сетевых администраторов, но и для пользователей брандмауэров.

ЛИТЕРАТУРА

[1] Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6), 2004.

[2] *Firewall wizards security mailing list* URL: http://honor.icsalabs.com/mailman/listinfo/firewall-wizards.

[3] Lihua Yuan, Jianning Mai, Chen-Nee Chuah. FIREMAN: A Toolkit for FIREwall Modeling and Analysis.

[4] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*: Progress. Draft, 1996.

[5] Девятков В.В. Построение, оптимизация и модификация процессов. *Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение*, 2012, № 2, с. 60–79

[6] Девятков В.В. *Системы искусственного интеллекта*. Москва, Изд-во МГТУ им. Н.Э. Баумана, 2001, 352 с.

[7] Леммон Е. *Алгебраическая семантика для модальных логик*. В кн.: *Семантика модальных и интенсиональных логик*. Москва, 1981.

Статья поступила в редакцию 28.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Девятков В.В., Мьо Тан Тун. Логический анализ корректности конфигурирования межсетевых экранов. *Инженерный журнал: Математическое моделирование*, 2013, вып. 11. URL: http://engjournal.ru/catalog/it/security/988.html

Девятков Владимир Валентинович родился в 1939 г., окончил Ленинградский институт точной механики и оптики в 1963 г. Д-р техн. наук, профессор, заведующий кафедрой «Информационные системы и телекоммуникации» МГТУ им. Н.Э. Баумана, Автор более 120 научных работ (из них 3 монографии). e-mail: deviatkov@bmstu.ru

Мьо Тан Тун родился в 1984 г., окончил МГУ им. М. В. Ломоносова в 2010 г. Аспирант кафедры «Информационные системы и телекоммуникации» МГТУ им. Н.Э. Баумана. e-mail: myothanhtun@gmail.com