Шурыгин Борис
012 группа

Средства защиты, основанные на подмене системных вызовов.

Востребованность.

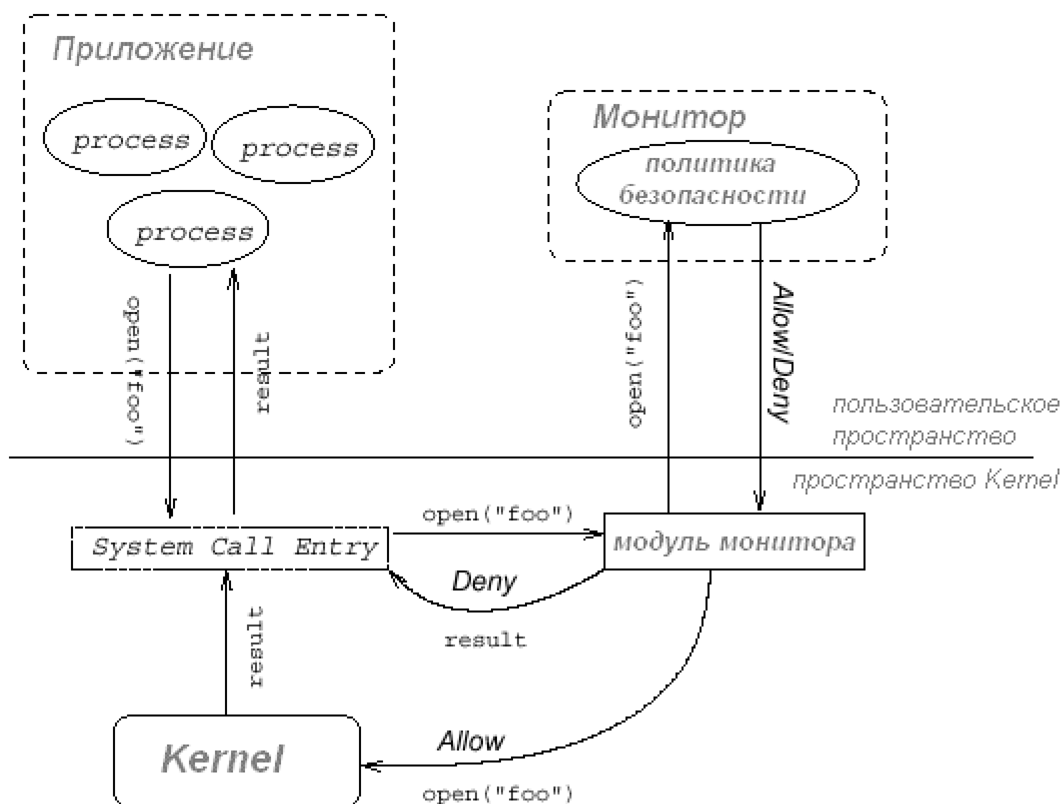
В настоящее время информационная безопасность стала одним из основных факторов, которые принимаются в рассмотрение при использовании приложений. Очевидно, что запуск на исполнение приложения, которое получено из непроверенного источника, создает опасность деструктивных действий приложения. Однако на практике нередко приходится пользоваться такими непроверенными приложениями, в этом случае, конечно, хотелось бы свести риск возможных нежелательных последствий к минимуму. Именно для этих целей используются средства защиты, позволяющие контролировать поведение приложения. Мощным методом наблюдения и контроля поведения приложения является применение средств, интегрированных в ядро операционной системы. Интерфейс системных вызовов является границей между приложением и операционной системой. Его мониторинг позволяет наблюдать все действия приложения при взаимодействии с сетью, файловой системой и другими системными ресурсами. Контроль поступления системных вызовов в ядро делает возможным контроль действий приложения. Например, очевидно, что для Unix-подобных ОС запрет, каких бы то ни было системных вызовов для непроверенного приложения, гарантированно не позволит ему нарушить политику безопасности. В то же время такой запрет может нарушить функциональность приложения, например, если для нормальной работы приложению требуется доступ к файловой системе. Т.е. проблема заключается в том, чтобы позволить приложению функционировать, когда оно не пытается произвести некорректных действий, и не допускать системных вызовов, которые возможно приведут к нежелательным последствиям.

Существует два основных повода сомневаться в поведении приложения еще до запуска, они и являются основными причинами помещения приложения под наблюдение и контроль. Первый – получение приложения из непроверенного источника. Интернет представляет в этом смысле очень богатое источниками программного обеспечения пространство, к сожалению, огромное их количество не поддается проверке. Второй причина заключается в том, что приложения могут нарушить функциональность системы, причем это нарушение функциональности не является целью создателей программы, а является программной ошибкой. Самый простой пример – ошибка, приводящая к большому потреблению системных ресурсов и существенно снижающая быстродействие или полностью приводящая к зависанию системы. В частности, для сервера это может

быть недопустимо. В любом случае если запуск подобных приложений на исполнение все же желателен, хотелось бы иметь защиту от потенциально нежелательных последствий.

Принцип действия и архитектура.

Назовем программу, осуществляющую наблюдение и контроль действий приложения, *монитором* (не путать с устройством). Сразу оговоримся – далее речь идет о реализациях мониторов для ядер операционных систем Linux и Solaris.



Монитор включает:

1) *Модуль монитора*. Является загружаемым модулем ядра. Главной особенностью загружаемого модуля ядра является возможность динамической загрузки и выгрузки, без необходимости перезагрузки всей системы или перекомпиляции ядра.
2) Сам *монитор* как приложение. Это приложение и занимается принятием решений по заданной пользователем политике безопасности.

Общий порядок действий монитора и архитектура:

1) Загружаемый модуль ядра отвечает за перенаправление и задержку системных вызовов до принятия решения приложением-монитором. Он передает монитору данные о типе системного вызова и его аргументах.
2) Основываясь на этих данных и определяемой пользователем политике безопасности, приложение-монитор принимает решение об отклонении или передаче в ядро системного вызова.
3) В случае принятия решения об отклонении вызова модуль ядра возвращает сообщение об отклонении вызова, что для приложения выглядит так, как будто бы вызов отклонен системой. В случае корректно написанного кода приложения эта ситуация должна обрабатываться самим приложением (например выдачей сообщения об ошибке, или новой попыткой произвести вызов), тогда мы не рискуем серьезно нарушить функциональность приложения.
4) В случае принятия решения о передаче вызова в ядро модуль предает вызов дальше в ядро, где этот вызов обрабатывается. Результат ядро возвращает приложению.

Такая организация работы позволяет сделать мониторинг полностью прозрачным для отслеживаемого приложения. Это позволяет не требовать от приложений никаких дополнительных возможностей и режимов работы. Также использование не встроенного, а загружаемого модуля позволяет существенно снизить требования к операционной системе, на которой выполняется мониторинг. Тем не менее, есть два требования, которые предъявляются к операционным системам для успешной реализации монитора. Первое требование состоит в том, чтобы система обладала возможностью, позволить некому посреднику (т.е. программе, не входящей в ядро) контролировать взаимодействие между процессом и его окружением. Второе состоит в том, чтобы система позволяла нам сузить количество доступных для приложения операций. К сожалению, разработка приложений под Win95,98 затруднена из-за необходимости программной изоляции ошибок (Software Fault Isolation - SFI), именно такие средства разрабатываются и реализуются для Windows на практике. Принцип работы SFI отчасти схож с работой монитора: SFI работает на уровне трансляции в модуле кода, через комбинацию статического анализа и динамических проверок для создания ограниченных сред. Система выделяет каждому модулю свое отдельное пространство памяти, в котором он и существует, как изолированная часть большой программы. Статические проверки гарантируют, что все изменения происходят в пределах одного модуля, а все внешние обращения связаны лишь с разрешенными функциями (Подробнее об этом см. [4]). Из-за непрерывного контроля

над всеми операциями, ухудшается производительность работы, но на это приходится идти. Большинство современных операционных систем имеют механизм отслеживания процессов. Для операционных систем Linux и Solaris посредничество между приложением и его окружением осуществляется по методу описанному выше. У этих операционных систем есть встроенная поддержка механизма отслеживания процессов. Во-первых, используется *ptrace*, такой подход применяется не только в мониторах, но и в средствах отладки программ. Во-вторых, существует специальная файловая подсистема /proc , которая предлагает более гибкий и удобный способ слежения за процессами (кстати, ptrace, поддерживаемый Solaris, реализован через /proc). И ptrace и /proc интерфейсы позволяют получать у ядра kernel информацию о моменте системного вызова. Интерфейс ptrace использует механизм сигналов Unix: в момент каждого события наблюдаемый процесс блокируется сигналом SIGTRAP, и программа наблюдения может обнаружить это, отслеживая этот сигнал с помощью wait. Интерфейс /proc использует ioctls.

Некоторые отслеживаемые ситуации.

Монитор отслеживает не только и не столько системные вызовы сами по себе, сколько комбинации этих сигналов и модели поведения непроверенных приложений. Комбинации допустимых по отдельности системных вызовов могут нарушать защиту! Вот несколько примеров:

1) Монитор следит за созданием наблюдаемым приложением новых процессов, т.е. после вызовов fork и exec новые процессы обязательно должны попадать под мониторинг. При этом возможен мониторинг этих процессов с другой политикой безопасности. Здесь кроется одна из возможных уязвимостей защиты. Разные политики безопасности позволяют приложениям, используя механизмы синхронизации и передачи данных между процессами, нарушать самым серьезным образом безопасность всей системы. Пример: запущенны два процесса и один имеет доступ к сети, а другой – к файловой системе. Взаимодействуя и используя корректные вызовы, соответствующие индивидуальной политике безопасности каждого из приложений, они могут легко передать через сеть, например все содержимое жесткого диска или наоборот закачивать из Интернета файлы, не спрашивая разрешения пользователя. В целях предотвращения монитор должен следить за передачей данных между процессами, сообразуясь с собственными установками ограничений на взаимодействующих процессах.

2)Монитор обязан принимать все меры по защите файловой системы от нежелательного доступа. Запрет получения дескрипторов на важные системные файлы – одна из основных задач монитора. Например, недопустим системный вызов вида “ *open*("/etc/passwd",...) ”. В этой области также существует ряд проблем, эти проблемы в основном связаны с разного рода «гонками» (File System races, arguments races, etc.). Разберем для ясности простейший случай. Пусть имеется символическая ссылка на некоторый файл, не являющийся важным с точки зрения безопасности. В таком случае если один процесс попытается, скажем, открыть файл, на который указывает ссылка, на чтение и запись, то это будет корректным действием с точки зрения монитора. При этом для проверки вызова всегда требуется время, на это время вызывающий процесс приостанавливается и монитор анализирует вызов, этого времени может вполне оказаться достаточным, чтобы другой процесс подменил *эту ссылку* на *ссылку на другой файл,* возможно важный. Тогда монитор, проанализировав старые данные о ссылке, вернет разрешающий ответ модулю, модуль передаст запрос в ядро и запрашивающий процесс получит доступ к важному ресурсу. Это открывает перед монитором целый класс проблем, которые, однако, почти всегда имеют решение. Продемонстрированный случай SymLink Race имеет, например, такое простое решение: запрет создавать символические ссылки на важную информацию для всех непроверенных процессов под наблюдением. Остальные случаи не так просты, в частности случай «гонок» на файловой системе или «гонок» аргументов более сложны и требуют объемных исследований интерфейса программирования приложений(API) на уязвимость для каждой конкретной операционной системы.

3) Существует ряд других задач монитора: слежение за состоянием окружения приложения, системы. При этом надо помнить, что в целях сохранения эффективности монитор ни в коем случае не должен пытаться симулировать работу ядра системы и избегать предположений о состоянии системы.

Заключение

Средства защиты, основанные на подмене системных вызовов, являются мощным инструментом для сведения к минимуму нежелательных последствий работы непроверенных приложений. Они находят применение в наблюдении за действиями почтовых программ, виртуальной машины Java, проверки скриптов, модулей Active X и других целей в основном связанных с проверкой сетевых приложений и отражением атак «червей». Для Windows описанные методы реализуются вместе с использованием SFI, для операционных систем на основе Unix они реализуются в виде двух частей –

загружаемого модуля ядра и монитора, принимающего решения. Существует реализация под ОС Solaris под названием Janus (см. [2]). Анонсируется выпуск Janus под Linux. Доступна для скачивания альфа-версия.

Ссылки:

[1] Владимир Мешков. *Перехват системных вызовов в ОС Linux.*

http://www.samag.ru/img/uploaded/2003/3/p3.pdf

[2] David A. Wagner. *Janus: an approach for confinement of untrusted applications*.

http://sunsite.berkeley.edu/TechRepPages/CSD-99-1056 - Janus

[3] Tal Garfinkel. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.*
http://www.stanford.edu/~talg/papers/traps/abstract.html

[4] *Технология для отражения неизвестных атак. Программная изоляция ошибки.*

http://www.i2r.ru/static/452/out_16208.shtml

[5]Jim Mauro, Richard McDougall. *Solaris Internals: Core Kernel Components.*

Sun Microsystems Press, 2001.