Простота и надёжность встраиваемых систем

Роман Филиппов (filippov.r@prosoft.ru)

В статье обсуждаются достоинства и недостатки архитектур современных операционных систем с точки зрения надёжности. Рассмотрен подход с использованием виртуализации, а также микроядерная архитектура и её модификации.

При современном широком распространении встроенных систем в промышленной автоматизации и транспорте значение надёжности и безопасности этих систем трудно переоценить. Необходимо учитывать также тот факт, что многие из этих устройств управляются дистанционно, а значит, их отказы могут быть спровоцированы не только случайностью или недостаточным тестированием, но также и намеренными действиями злоумышленников и вредоносного программного обеспечения (ПО).

Встроенное ПО, реализующее основные функции устройства, может быть достаточно компактным и тщательно протестированным, но помимо него устройство обычно содержит также большое количество инфраструктурного кода, от которого также зависит надёжность устройства в целом. В качестве примеров таких инфраструктурных компонентов можно привести различные стеки коммуникационных протоколов, файловые системы и тому подобное. Из-за сложности и больших объёмов кода, содержащихся в современных операционных системах (ОС), разработчик встроенного ПО нередко использует ОС и прочие программные компоненты как «чёрный ящик», что может привести к ситуации, когда сделанные при разработке допущения оказываются неверными.

Главный тезис данной статьи заключается в том, что средством обеспечения надёжности и безопасности встроенных систем является их простота и детальное понимание разработчиком всех аспектов работы системы. Это позволяет предвидеть многие векторы атак и заранее предпринять меры, затрудняющие использование уязвимостей. Кроме того, простую и понятную систему проще тестировать, модифицировать и поддерживать.

Если обратиться к истории, то именно борьба со сложностью является основной движущей силой программирования как инженерной дисциплины. Стремление добиться простоты во многом привело к созданию ОС UNIX, которая изначально появилась как попытка избавиться от чрезмерной сложности системы Multics. Изначально UNIX была простой в освоении системой, но сегодняшние реализации UNIX-подобных ОС уже не могут похвастаться простотой своего прародителя.

Проблема развития ОС

Наиболее широко используемой ОС во встроенных системах (если не рассматривать микроконтроллеры) являются различные варианты Linux – как с RT-расширениями, так и без них. Linux является достаточно зрелой ОС, которая разрабатывается уже более двух десятилетий. Однако, несмотря на долгое развитие и стабильность проекта, в ней до сих пор находят уязвимости [1]. Парадоксально, но главной проблемой является именно развитие проекта. По мере развития в ОС добавляется много нового кода, содержащего новые ошибки. Эти ошибки исправляются, но к этому времени добавляется уже новый функционал и так далее. Эти два состязающихся между собой процесса: исправления ошибок и внесения новых, могут принципиально не сходиться к надёжной системе. В среднем, количество ошибок и связанных с ними уязвимостей плавно возрастает от версии к версии.

Дополнительной проблемой является большой объём кода и необходимость использования в виде «чёрного ящика» не только самого ядра, но также и всех драйверов. Полностью разобраться в исходных текстах ядра довольно затруднительно из-за высокой степени асинхронности его кода и сложной синхронизации.

Встроенные системы не всегда имеют доступ к Интернету, поэтому обновления не всегда могут быть установлены вовремя, даже в том случае, если в новой версии ядра ошибки были исправлены. К тому же и сам переход на новую версию ядра зачастую требует дополнительных усилий от разработчиков встроенного ПО. Фактически, это приводит к тому, что если не используется самая актуальная версия ядра в любой момент времени, то система является потенциально уязвимой для внешней эксплуатации, так как ошибки публикуются в централизованной базе данных CVE. Эксплуатация происходит практически по шаблону: определить версию используемой ОС; найти подходящие уязвимости этой версии в базе CVE; разработать или даже взять готовый эксплоит. Всё это даёт основания заключить, что монолитные системы, такие как Linux, несмотря на все свои достоинства, всё-таки разрабатывались не для этого и плохо подходят для ответственных встроенных систем.

Виртуализация

Большинство используемых сегодня ОС начали разрабатываться в конце прошлого века, поэтому их архитектура во многом продиктована соображениями эффективности в ущерб надёжности. В частности, Linux имеет монолитную архитектуру ядра, в том числе потому, что с другой архитектурой было бы сложно добиться приемлемого уровня производительности на аппаратуре начала 90-х годов. Сегодня, когда проблема надёжности и безопасности может быть важнее производительности, одним из вариантов решения проблемы унаследованного кода является использование виртуализации и гипервизоров.

Технологии аппаратной виртуализации позволяют запускать несколько экземпляров ОС на одном физическом процессоре, что позволяет рассматривать ОС вместе со всеми её приложениями в виде одного большого «приложения», управляемого гипервизором, который можно рассматривать как «ядро». Если приложение с особыми требованиями к надёжности и потенциально ненадёжная ОС, содержащая инфраструктурные компоненты, реализованы в виде отдельных виртуальных машин, то гипервизор обеспечивает их надёжную изоляцию друг от друга.

Данное решение в большинстве случаев вполне приемлемо, но нельзя не отметить следующие недостатки: повышение сложности из-за дополнительных слоёв ПО, в которых также могут содержаться ошибки; необходимость использования довольно продвинутых и дорогих про-

цессоров, поддерживающих аппаратную виртуализацию; добавление гипервизора в список доверенных частей ПО, так как все прочие компоненты системы зависят от него. Также для такой системы сложнее разрабатывать прикладное ПО: виртуальные машины по степени изоляции сравнимы с физически раздельными компьютерами, поэтому взаимодействие между ними является более сложным, нежели взаимодействие процессов в рамках одной ОС.

Всё это позволяет заключить, что виртуализация не даётся бесплатно, к тому же возрастающая сложность процессоров также не помогает в вопросах повышения надёжности и огромное количество аппаратных ошибок и уязвимостей в процессорах в последнее время появляется именно из-за их высокой сложности.

Одним словом, решение хоть и относительно хорошее, но применимое не во всех ситуациях, особенно в системах жёсткого реального времени с особыми требованиями к надёжности.

Между тем существующие технологии защиты памяти в виде разделения режимов работы процессора и установки прав доступа к памяти в виде MPU/MMU способны обеспечить основу для построения защищённых и надёжных приложений. На этой основе можно строить сколь угодно сложные системы и изолировать приложения друг от друга. При этом от процессора не требуется аппаратной поддержки механизмов виртуализации. Разумеется, если причиной использования Linux является его набор драйверов, отсутствующий в других ОС, то вариант с гипервизором остаётся единственно возможным. Если же вопрос с драйверами стоит менее остро, то целесообразно рассмотреть другие варианты обеспечения надёжности, в целях снижения сложности.

Микроядро

Попытки разделить функционал ядра ОС на более удобные для понимания и изолированные друг от друга компоненты начались практически одновременно с разработкой ранних ОС. Большой объём инженерных усилий был затрачен на разработку так называемых микроядер, первые представители которых появились в 70-е годы.

Микроядро предполагает выполнение таких традиционных компонентов ядра, как файловые системы, драйверы и сетевые протоколы в виде обычных приложений, называемых также серверами, потому что они, по аналогии с сетевыми серверами, предоставляют услуги программам пользователя. В микроядерной архитектуре приложение взаимодействует с серверами, используя механизмы межпроцессного взаимодействия, например, с помощью обмена сообщениями (см. рис. 1).

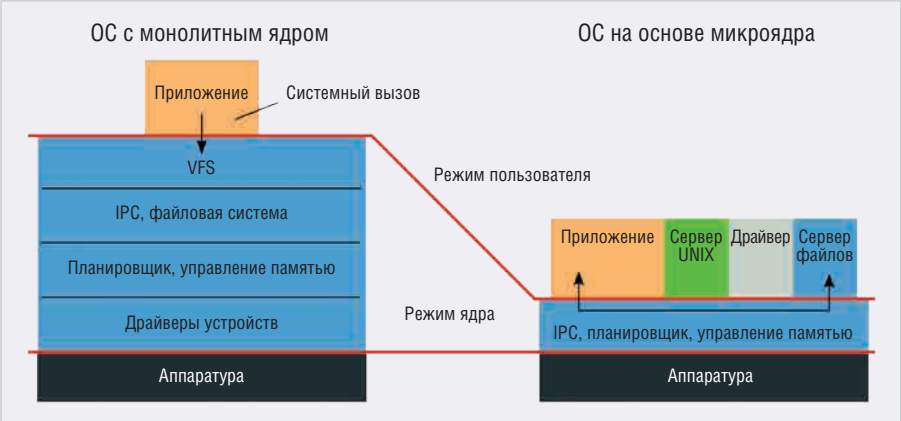

Рис. 1. Архитектура ядра ОС

На ядро возлагаются только базовые обязанности вроде планировщика и реализации механизмов межпроцессного взаимодействия. Серверы изолированы друг от друга и от ядра, а кроме того, обладают (в большинстве случаев) структурой в виде цикла обработки сообщений. За счёт такой архитектуры микроядерные системы, как правило, более просты и надёжны.

Стоит особо подчеркнуть, что популярная в последнее время практика отнесения ОС к микроядерным на основании только размера ядра является ошибочной. Микроядро – это прежде всего именно архитектура, в которой компоненты самой ОС выполняются в пользовательском режиме и обслуживают запросы приложений через механизмы межпроцессного взаимодействия. Компактный размер самого ядра является побочным эффектом от такой архитектуры, а не критерием отнесения ОС к тому или иному типу.

Отсутствие функционала в ядре, требующего изменений для поддержки нового оборудования или сетевых протоколов, выгодно отличает микроядерную архитектуру от монолитных ядер. По мере исправления ошибок новые ошибки практически не вносятся, так как отсутствует параллельный процесс развития ядра. Это даёт надежду на то, что рано или поздно этот процесс разработки сойдётся к программе, в которой нет эксплуатируемых ошибок.

Микроядерный UNIX

Проблемой, ограничивающей применение микроядерной архитектуры

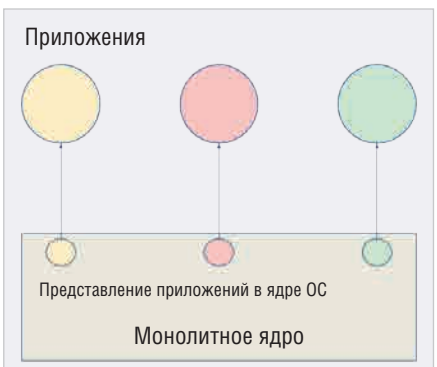

Рис. 2. Информация о состоянии приложений в монолитном ядре

в надёжных системах, является огромное количество унаследованного кода, который использует интерфейсы UNIX (POSIX). Преодолеть эту проблему призваны попытки реализовать UNIX-подобный интерфейс поверх микроядра. Это позволило бы использовать достоинства микроядер с существующей инфраструктурой.

Среди применяемых ОС с такой архитектурой можно упомянуть MINIX, а также операционную систему реального времени (ОСРВ) QNX. Архитектура этих ОС построена вокруг микроядра, которое управляет процессами и их взаимодействием, а все драйверы, стеки протоколов и файловые системы вынесены в пользовательское пространство и могут быть перезапущены во время работы.

К сожалению, попытки реализовать ОС UNIX, которая не была бы монолитной системой, сталкиваются с фундаментальными проблемами, присущими самому UNIX-подобному интерфейсу. Если в монолитных ОС состояние процесса, такое как открытые файлы, сетевые сокеты и прочее находится в некоторых структурах данных внутри ядра (см. рис. 2), то в микроядрах это состояние хранится уже не в одном месте, а в

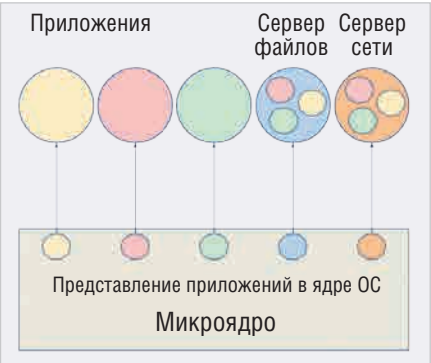

Рис. 3. Информация о состоянии приложений в микроядре

виде множества частей, каждая в своём сервере. Открытые файлы и связанные с ними атрибуты хранятся в сервере, реализующем VFS (Virtual File System), сокеты – в сетевом сервере и так далее (см. рис. 3).

Например, системный вызов fork, который является единственным способом создания нового процесса в классическом UNIX, должен создать полную копию выполняемого процесса за исключением некоторых специально оговорённых вещей. То есть новый процесс должен выполнять тот же код, иметь доступ к тем же данным и иметь те же самые открытые файлы.

Помимо очевидных проблем с предсказуемостью: данные потомка должны копироваться из предка при первом обращении к ним, данный системный вызов требует создания полной копии всего состояния. Очевидно, что в случае классического микроядра это требует дублирования состояния процесса в каждом из серверов. Учитывая, что сами серверы работают асинхронно, а к тому же являются ещё и потенциально ненадёжными, реализация такого функционала выливается в серьёзную проблему, тогда как в классических монолитных ОС это делается довольно просто.

Системные процессы

Способ, который применяется для решения этой проблемы, заключается в том, что состояние пользовательских процессов хранится в специальном системном процессе, который выполняет роль UNIX-сервера.

В этом случае состояние процессов хранится в одном месте и его относительно просто скопировать. В этом же процессе может храниться дерево файловой системы (которое является глобальным для всех процессов в UNIX) и тому подобное. В QNX роль такого процесса выполняет Process manager (procnto), в задачи которого входит управление процессами, памятью, загрузкой образов исполняемых файлов и поддержка глобального пространства имён [3]. В MINIX также есть аналогичный процесс (Process manager), являющийся, наряду с файловой системой, обязательным системным компонентом [4].

Подобные системы являются более надёжными, нежели традиционные монолитные ядра, хотя бы по той причине, что драйверы устройств являются основным источником ошибок, и они в данных архитектурах полностью изолированы. Тем не менее, побочным эффектом такой архитектуры с системным процессом является также и то, что этот процесс входит в доверенную базу системы наравне с микроядром. Ошибки и уязвимости в этом процессе могут вести к компрометации всей системы. А учитывая, что он содержит практически весь функционал традиционного UNIX, можно предположить, что основной источник ошибок будет содержаться именно в этом критическом процессе. Статистика показывает верность этого предположения [5, 6]. По данным MWRLabs, уязвимости самого микроядра QNX составляют менее 10% всех найденных уязвимостей. Львиная доля ошибок заключена в реализации UNIX-функций вроде setuid, загрузчике образов и т.д. Таким образом, даже при наличии полностью корректного микроядра, нельзя делать выводы относительно надёжности систем на его основе.

Что касается различных вариантов верифицированных микроядер L4, то, помимо наличия критических процессов, для упрощения формальной верификации используется подход, при котором ядро является вытесняемым лишь частично, поэтому не может использоваться там, где важна предсказуемость и низкая латентность прерываний.

FX-RTOS

С 2010 года компания Eremex разрабатывает собственную микроядерную ОСРВ FX-RTOS, которая призвана в какой-то степени решить эти проблемы. Подход, предлагаемый FX-RTOS, основан на известном правиле «80/20», которое переформулируется в виде «реализация 80% функционала традиционного UNIX используя 20% кода» плюс отказ от функций, негативно влияющих на безопасность. Отказ от хранимого в единственном месте глобального состояния приложений позволил реализовать систему без критических процессов. Реализовав приложение или сервер, работающие поверх микроядра и выполняющие некоторую важную функцию, пользователь может быть уверен, что даже в случае аварийного завершения всех прочих процессов, это приложение не только продолжит работать, но и будут соблюдены его требования к работе в реальном времени. Нет необходимости принимать на веру надёжность инфраструктурных компонентов, если они не используются. Более того, в рамках одной системы можно допустить сосуществование «надёжной» и «ненадёжной» инфраструктуры. Например, ответственное приложение может использовать свой собственный выделенный стек UDP/IP (который гораздо проще полноценного TCP/IP), работающий на отдельном аппаратном интерфейсе, тогда как прочие приложения могут использовать стандартный сетевой стек. Таким образом, достигаемый уровень изоляции и независимости приложений сравним с таковым при использовании виртуализации, но при этом реализуемый силами микроядра, без использования гипервизоров и аппаратной поддержки соответствующих технологий.

Особое внимание уделяется также защите от атак, приводящих к отказу в обслуживании. В частности, все ресурсы процесса, которые могут быть выделены статически, выделяются статически на этапе его запуска и впоследствии не изменяются. Тем самым гарантируется наличие этих ресурсов в случае аварийного перезапуска, в отличие от традиционных систем, где выделение ресурсов происходит по запросу и в принципе может закончиться неудачно. Например, для каждого процесса должен быть определён максимальный размер памяти, который ему разрешается использовать. При запуске процесса эта память выделяется сразу, после чего стеки потоков и куча выделяются уже внутри процесса. На это можно возразить, что в разные моменты времени разным процессам может потребоваться вообще вся доступная в системе оперативная память и такой подход слишком ограничивает пользовательские программы. Но, если в системе возможно существование нескольких

процессов, которые находятся на пике своего использования памяти, получается, что о надёжности в такой системе говорить не приходится, поскольку после исчерпания памяти одним из процессов, остальные работать не смогут. Резервирование позволяет гарантировать, что если процесс запустился, то он сможет работать, несмотря на действия всех других процессов, поэтому такую систему в целом проще протестировать.

Привилегии процессов настраиваются во время сборки образа ОС и отсутствует возможность повышения привилегий во время работы. Процесс-драйвер не может обращаться к любым устройствам, а только к тем, работа с которыми разрешена для него, поэтому даже в случае наличия в нём уязвимостей, взять под контроль всю систему невозможно. Подобные принципы распространяются и на другие компоненты ОС.

Литература

1. Two DDoS Friendly Bugs Fixed in Linux Kernel. https://www.bleepingcomputer.com/news/linux/two-ddos-friendly-bugs-fixed-in-linux-kernel/.
2. Linux Kernel: CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.
3. Process Manager. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/proc.html.
4. *Таненбаум Э.* Операционные системы. Разработка и реализация. 3-е издание. СПб.: Питер, 2007.
5. *Plaskett A., Geshev G.* QNX: 99 Problems but a Microkernel ain'tone! https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-qnx-troopers-99-problems-but-a-microkernel-aint-one.pdf.
6. *Plaskett Alex.* QNX Security Architecture Whitepaper. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-qnx-security-whitepaper-2016-03-14.pdf.

НОВОСТИ МИРА

ASTRALINUX и «КриптоПро» совместимы на процессорах «Эльбрус» и «Байкал»

Компания ASTRALINUX сообщает, что впервые средство электронной подписи «КриптоПро CSP» сертифицировано для применения в российской операционной системе для процессорной архитектуры «Эльбрус» и «Байкал» (MIPS). Компанией «КриптоПро» получены сертификаты соответствия на «КриптоПро CSP 4.0 R4», сборка 4.0.9963 (Abel). При этом подтверждена работа данного средства криптографической защиты информации (СКЗИ) в следующих программно-аппаратных средах:

- вариант исполнения 1-Base: в среде ASTRALINUX на процессорных архитектурах x64, «Эльбрус», MIPS;
- вариант исполнения 2-Base: в среде ASTRALINUX на процессорных архитектурах x64, «Эльбрус».

Для работы с системами электронного правительства сотрудникам госорганов и бюджетных учреждений необходимо использовать браузеры, которые поддерживают защищённые https-соединения (TLS) и обеспечивают возможность работы с отечественными сертифицированными средствами формирования и проверки усиленной квалифицированной электронной подписи.

www.astralinux.ru