Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»

Физтех-школа Прикладной Математики и Информатики
Факультет Управления и Прикладной Математики
Кафедра информатики и вычислительной математики

Направление подготовки / специальность: 03.03.01 Прикладные математика и физика
Направленность (профиль) подготовки: Компьютерные технологии и интеллектуальный анализ данных
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
_____/_____
(подпись) *расшифровка подписи*
«___» июня 2018 г.

Исследование технологий модификации кода в исполняющейся программе

(бакалаврская работа)

Обучающийся: Емчинов Александр Владимирович
_____
(подпись обучающегося)
Научный руководитель: Кинсбурский Станислав Александрович, к.т.н.
_____
(подпись научного руководителя)

Москва 2018

Аннотация

Целью этой работы является исследование технологии динамической модификации кода запущенных программ. Объектом исследования является проект virtuozzo/nsb. Задача исследования - портирование virtuozzo/nsb для применения его к 32-битным приложениям. Результатом выполненной работы является расширение функциональности virtuozzo/nsb, позволяющей использовать этот проект для модификации запущенных приложений, собранных для архитектуры x86.

Аннотация 2

1. Введение 5

2. Описание технологии Live Patching 7

2.1 Необходимые понятия 7

2.1.1 Патч 7

2.1.2 Live Patching 8

2.2 Устройство DSU-систем 9

2.2.1 Общий принцип работы DSU-систем 9

2.2.2 Требования к DSU-системам 11

2.2.3 Преобразование кода 12

3. Существующие системы обеспечивающие технологию Live Patching 12

3.1. Ksplice, kpatch, kGraft 13

3.2 Устройство kPatch 14

3.3 Устройство Kgraft 15

3.4 Устройство kSplice 18

3.5 Устройство libcare 20

4. Технология Live Patching средствами проекта virtuozzo/nsb 21

4.1 Необходимые понятия 21

4.1.1 Ptrace 21

4.1.2 ELF формат 21

4.2 Обзор nsb 23

4.3 Принцип прикладывания патчей средствами nsb 24

4.3.1 Прекращение выполнения процесса в безопасной точке извне 25

4.3.2 Загрузка и применение патча к исполняемому коду процесса 25

4.3.3 Освобождение процесса 25

4.4 Использование diff для оценки возможности создания живых патчей 26

4.5 Создание живых патчей 26

4.5.1 Создание патчей вручную 27

4.5.2 Автоматическое создание патчей 27

4.6 Применение патчей к процессам 27

4.7 Ограничения по технологии Live Patching 29

5. Постановка проблемы 30

5.1 Формулирование проблемы 30
5.2 Постановка задачи 30

6. Решение задачи 30
6.1 Исходная платформа 30
6.2 Ход решения задачи 30
6.2.1 Установка и запуск nsb 31
6.2.2 Написание мануальных тестов для исследования принципов работы nsb 32
6.2.3 Тестирование исходного проекта на архитектуре x86 32
6.2.4 Определение хода работы для поддержки x86 архитектуры 32
6.3 Построение алгоритма и его реализация 33
6.3.1 Работа с generator и protobuf 33
6.3.1.1 Изменения generator для работы с файлами, собранными на архитектуре x86 33
6.3.2 Работа с patcher 33
6.3.2.1 Обработка данных, полученных с генератора 33
6.3.2.2 Принцип введения ветвления 33
6.3.2.3 Предварительные предположения об архитектурно-зависимых функциях 34
6.3.2.4 Поиск архитектурно-зависимых функций и их обработка 34

7. Заключение 35

Список использованных источников **36**

1. Введение

Как правило промышленная разработка программных продуктов итеративна и сопровождается поддержкой, в которую можно включить исправление найденных за время эксплуатации ошибок, улучшение и обновление устаревших модулей, добавление новой функциональности, обеспечение совместимостей с другими программными продуктами. Такая поддержка может осуществляться непосредственным переписыванием исходного кода или использованием специальных патчей, предназначенных для внесения нужных изменений в файлы программы.

Приложения, обладающие длительным сроком жизнедеятельности могут быть чувствительны к остановке. К таким приложениям можно отнести СУБД, файловые хранилища и другие.

Например, систему управления базами данных можно принудительно остановить. В таком случае время на повторную загрузку может быть достаточно велико для того, чтобы доставить неудобства пользователям, т.к. СУБД при старте как правило создают и инициализируют кеш, позволяющий ускорить ее работу в дальнейшем.. К тому же за время формирования кэша производительность остальных систем может быть ограничена.

Сюда же можно отнести сервисы, которые обслуживают длительные сетевые соединения. Зачастую необходимо поддерживать очень долгое соединение с клиентом. При перезагрузке подобных приложений есть риск потерять соединение с сервисом.

Решением подобного рода проблем, является технология Userspace Live Patching. Эта технология позволяет изменять логику работу приложений без их перезагрузки путем прикладывания к целевому запущенному приложению “живых патчей”.

В данной работе рассмотрен проект virtuozzo/nsb. Он представляет собой автоматизированное средство для создания и применения живых патчей. Он основан на использовании библиотеки Compel.

В библиотеку Compel вынесена возможность исполнения паразитного кода в процессе. На основе Compel работает система Live Patching(живой патчинг приложений).

Таким образом проект virtuozzo/nsb представляет собой программное обеспечение, позволяющее извне изменять код запущенных программ (Live Patching) для микропроцессорной архитектуры x86-64.

nsb основан на инструментировании, а не на анализе бинарного кода и изначально разрабатывался, как проект портируемый на другие архитектуры. Данная его особенность является его основным отличием перед другими аналогами, предоставляющими подобную функциональность и основанными на анализе бинарного кода.

Задачей данной работы является исследование проекта nsb, принципов его работы, исследование возможности портирования данного проекта для архитектуры x86 и его реализация.

В результате проделанной работы nsb был дополнен функциональностью, позволяющей легко его портировать на другие архитектуры. Эта функциональность применима для любых типов архитектур, т.к. представляет собой некоторое абстрактное представление архитектуры внутри проекта. Основным результатом этой работы является портирование исходного проекта virtuozzo/nsb для живого патчинга программ, собранных для архитектуры x86. Достигнутый результат полезен и интересен по ряду причин. Во-первых, он представляет собой практическое доказательство преимущества nsb с точки зрения портирования для других архитектур. Во-вторых, возможности проекта virtuozzo/nsb расширены за счёт добавления поддержки новой архитектуры x86.

2. Описание технологии Live Patching

2.1 Необходимые понятия

2.1.1 Патч

Патч - это часть программного обеспечения, предназначенного для обновления компьютерной программы, для ее поддержки, для исправления логики ее работы или улучшения. Он как правило используется для исправлений уязвимостей безопасности и других ошибок, причем такие исправления обычно называются исправлениями или исправлениями ошибок и улучшают удобство использования или производительность. В некоторых особых случаях патчи могут сознательно нарушать функциональность, например, путем удаления компонентов, для которых поставщик обновлений больше не лицензируется.

Управление патчами является важной частью управления жизненным циклом программного обеспечения и представляет собой важный процесс его дальнейшей поддержки и обновления.

2.1.2 Live Patching

Hot Patching, также известное как Live Patching или динамическое обновление программного обеспечения - это применение патчей без выключения и перезапуска системы или соответствующей программы. Такое обновление устраняет проблемы, связанные с недоступностью службы на время рестарта или инициализации, предоставляемой системой или программой. Патч, который может быть применен таким образом, называется “горячим патчем” или “живым патчем”. Эта технология становится обычной практикой в пространстве мобильных

приложений. Такие компании, как Rollout.io, используют метод swizzling для доставки горячих исправлений в экосистему iOS(https://rollout.io/blog/swift-method-swizzling/). Другим методом для горячих исправлений iOS-приложений является JSPatch(https://github.com/bang590/JSPatch).

В науке область исследований, относящихся к обновлению программ во время их работы называется "динамическим обновлением программного обеспечения" (Dynamic Software Updating - DSU). В настоящее время DSU широко используется в промышленности. Исследователи разработали широкий спектр систем и методов для внедрения DSU. Эти системы обычно тестируются в реальных программах.

Большинство современного программного обеспечения и операционных систем не включает в себя технологии поддержки DSU. По этой причине, реализация DSU обычно либо использует сторонние инструменты, либо реализуется с помощью специализированных компиляторов. Эти компиляторы сохраняют семантику исходной программы, но используют либо исходный код, либо код объекта для создания динамически обновляемой программы.

2.2 Устройство DSU-систем

2.2.1 Общий принцип работы DSU-систем

Любая работающая программа может быть рассмотрена в виде кортежа ***(s, P)***, где ***s*** - текущее состояние программы, а ***P*** - текущий программный код. Динамические системы обновления программного обеспечения преобразуют запущенную программу ***(s, P)*** в новую версию

(s', P'). Для этого состояние должно быть преобразовано в представление ***P'***. Для этого требуется функция трансформатора состояния - ***f***. Таким образом, DSU преобразует программу ***(s, P)*** в ***(f(s), P')***. Обновление считается действительным тогда и только тогда, когда запущенная программа ***(f(s), P ')*** может быть сведена к точечному набору ***(s, P')***, который доступен из начальной точки новой версии программы ***(s(init), P')***.

Место в программе, где происходит динамическое обновление, называется точкой обновления. Существующие реализации DSU широко варьируются в реализации обработки точек обновления. В некоторых системах, таких как UpStare(http://www.mkgnu.net/upstare) и PoLUS, обновление может происходить в любое время во время выполнения. Компилятор Ginseng попытается сделать вывод о хороших местах для точек обновления, но также может использовать точки обновления, установленные программистом. Kitsune и Ekiden требуют от разработчиков вручную указывать все точки обновления.

Системы обновления различаются по типам программных изменений, которые они поддерживают. Например, Ksplice поддерживает только изменения кода в функциях и не поддерживает изменения в представлении состояния. Это связано с тем, что Ksplice в первую очередь нацелен на изменения безопасности, а не на общие обновления. Напротив, Ekiden может обновлять программу до любой другой программы, которая может быть выполнена. Сюда, в частности включаются программы, написанные на другом языке программирования. Разработчики систем могут извлечь ценную производительность или гарантировать безопасность программы, ограничив объем обновлений. Например, любая проверка безопасности обновлений ограничивает их объем только теми, которые проходят проверку безопасности. Механизм,

используемый для преобразования кода и состояния, влияет на то, какие обновления будет поддерживать система.

Системы DSU, как и инструменты их использования, могут быть оценены на простоте их использования и ясности для разработчиков. Многие системы DSU, такие как Ginseng, требуют, чтобы программы подвергались различным статическим анализам. Несмотря на то, что эти анализы выявляют свойства программ, которые ценны для DSU, они являются сложными для понимания. Системы DSU, которые не используют статический анализ, могут потребовать использования специализированного компилятора.

Программы, которые обновляются системой DSU, называются целевыми программами. Академические публикации систем DSU обычно включают несколько целевых программ в качестве тематических исследований. vsftpd, OpenSSH, PostgreSQL, Tor, Apache, GNU Zebra, memcached и Redis - это все цели динамического обновления для различных систем. Поскольку несколько программ написаны с учетом необходимости динамического обновления, переоснащение существующих программ является ценным средством оценки системы DSU для практического использования.

2.2.2 Требования к DSU-системам

Системы DSU должны загружать новый код в запущенную программу и преобразовывать существующее состояние в формат, понятный новому коду. Поскольку многие мотивы использования DSU являются критическими по времени (например, развертывание исправления безопасности в живой и уязвимой системе), системы DSU должны обеспечивать достаточную доступность обновлений. Некоторые системы

DSU также пытаются обеспечить безопасность обновлений перед их применением.

Канонического решения ни одной из этих проблем не существует. Как правило, система DSU, которая хорошо работает в одной проблемной области, делает это в компромиссе с другими. Например, эмпирическое тестирование динамических обновлений указывает на то, что увеличение количества точек обновления приводит к увеличению числа небезопасных обновлений.

При разработке систем DSU необходимо учитывать, проблемное пространство, решаемое динамическим обновлением, которое можно рассматривать как пересечение нескольких других. Примеры включают контрольную точку, динамическое связывание и устойчивость. Например, база данных, которая должна быть обратно совместима с предыдущими версиями своего формата на диске, должна выполнить тот же тип преобразования состояния, ожидаемый от системы динамического обновления. Аналогично, программа, имеющая архитектуру плагина, должна иметь возможность загружать и выполнять новый код во время выполнения.

Подобные методы иногда также используются для целей динамического устранения мертвого кода, для удаления условно мертвого или недостижимого кода при загрузке или исполнении, они рекомбинируют оставшийся код, чтобы минимизировать объем памяти или повысить скорость.

2.2.3 Преобразование кода

Большинство систем DSU используют подпрограммы в качестве единицы кода для обновлений, однако новые системы DSU реализуют обновления всей программы.

В таких языках, как C / C ++, системы DSU могут использовать специализированные компиляторы, которые вставляют косвенные функции в программу. Если система DSU не использует компилятор для вставки указателей статически, то они вставляются во время выполнения с бинарным переписыванием. Бинарное переписывание - это процесс написания низкоуровневого кода в образ памяти исполняемой программы для перенаправления функций. Хотя такой способ модификации не требует статического анализа программы, он сильно зависит от архитектуры.

Ekiden и Kitsune загружают новый программный код через запуск совершенно новой программы либо через fork-exec, либо с динамической загрузкой. Затем существующее состояние программы переносится в новое программное пространство.

3. Существующие системы обеспечивающие технологию Live Patching

3.1. Ksplice, kpatch, kGraft

kpatch - это функция ядра Linux, которая реализует обновление в реальном времени запущенного ядра, что позволяет применять патчи во время его работы. Избегая необходимости перезагрузки системы с новым ядром, содержащим требуемые исправления, kpatch стремится максимизировать время безотказной работы и доступность системы. В то же время kpatch позволяет применять обновления безопасности, связанные с ядрами, не откладывая их до запланированных простоев.

Внутренне, kpatch позволяет целым функциям в запущенном ядре замениться их исправленными версиями, делая это безопасно, останавливая все запущенные процессы, пока выполняется обновление в реальном времени.

Ksplice - это расширение ядра Linux с открытым исходным кодом, которое позволяет применять исправления безопасности к работающему ядру без необходимости перезагрузки, избегая простоев и повышения доступности (метод, широко называемый динамическим обновлением программного обеспечения). Ksplice поддерживает только патчи, которые не вносят существенных семантических изменений в структуры данных ядра.

kGraft - это особенность ядра Linux, которая реализует обновление в реальном времени запущенного ядра, что позволяет применять патчи ядра во время работы ядра. Точно также как и kpatch kgraft избегает необходимости перезагрузки системы с новым ядром, содержащим требуемые исправления, он стремится максимизировать время безотказной работы и доступность системы. Обновления безопасности, kGraft позволяет применять, не откладывая их до запланированных простоев. Внутренне kGraft аналогично kpatch позволяет целым функциям в запущенном ядре замениться на их исправленные версии, делая это безопасно, выборочно используя оригинальные версии функций, чтобы обеспечить согласованность каждого процесса, когда выполняется обновление в реальном времени.

libcare - в отличие от перечисленных выше технологий, предназначена для Live Patching пользовательских приложений, а не ядра.

3.2 Устройство kPatch

По существу, kpatch состоит из двух частей - модуль ядра выполняет механизм Live Patching, изменяя внутреннюю работу ядра, в то время как набор утилит пользовательского пространства подготавливает отдельные модули ядра hot patch из исходных diff и управляет их приложением. Live Patching ядра выполняется на уровне функции, что означает, что kpatch может заменить целые функции в запущенном ядре на их исправленные версии, используя средства, предоставляемые ftrace для «маршрутизации» старых версий функций. Таким образом, горячие патчи также можно легко отменить. Никакие изменения во внутренних структурах ядра не возможны; однако патчи безопасности, которые являются одним из естественных кандидатов, которые будут использоваться с kpatch, редко содержат изменения в структурах данных ядра.

kpatch гарантирует, что горячие исправления применяются атомарно и безопасно, останавливая все запущенные процессы, пока применяется горячий патч, и гарантируя, что ни один из остановленных процессов не запущен внутри функций, которые должны быть исправлены. Такой подход упрощает весь механизм исправления в реальном времени и предотвращает определенные проблемы, связанные с тем, как структуры данных используются исходными и исправленными версиями функций. Как недостаток, этот подход также оставляет возможность для горячего патча терпеть неудачу и вводит небольшую задержку, необходимую для остановки всех запущенных процессов.

На рисунке 1 изображена принципиальная схема работы kPatch на примере патчинга функции. Наверху изображен стандартный вызов функции при исполнении, внизу отражена схема патчинга с использованием ftrace-маршрутизатора.

На схеме, соответствующей работе после патчинга показано взаимодействие маршрутизатора с другими частями.

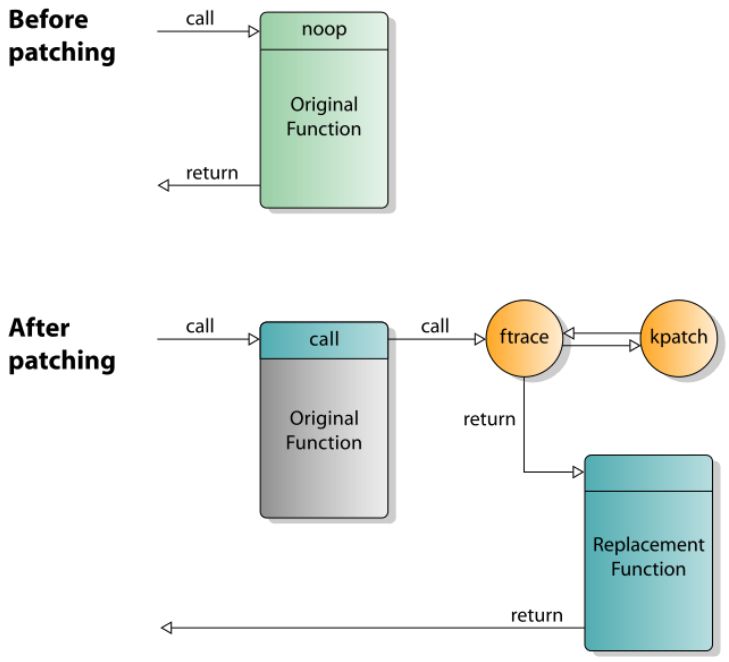

Рисунок 1 - устройство kPatch

3.3 Устройство Kgraft

Kgraft по своему внутреннему устройству аналогичен kPatch. Но в его реализации присутствуют принципиальные отличия.

При применении горячих патчей kGraft не требует останова запущенного ядра для исправленных версий функций, которые будут введены в него. Вместо замены функций атомарно kGraft обеспечивает согласованные «юниверсы» для процессов пользовательского пространства, потоков ядра и обработчиков прерываний, которые контролируются во время их выполнения, поэтому исходные версии исправленных функций ядра могут по-прежнему использоваться. Для поддержки этого kGraft поддерживает оригинальные версии исправленных функций в модуле «чтение-копирование-обновление» (Read-Copy-Update

или RCU) и динамически выбирает между исходными и исправленными версиями, в зависимости от того, какой процесс выполняет поток ядра или обработчик прерываний. Более конкретно, исходные версии функций продолжают использоваться - в то время, когда применяется горячий патч.

Благодаря своей конструкции kGraft не вводит дополнительную задержку при применении горячих патчей.Главный недостаток kGraft - сложность в реализации, а следовательно его меньшая надежность. Также в качестве недостатков можно заметить требование того, что исходные версии исправленных функций ядра должны поддерживаться в течение длительных периодов времени в случае, если есть процессы, которые слишком долго остаются в пространстве ядра; например, процесс может ожидать ввода-вывода в сетевом сокете. Кроме того, поскольку как исходные, так и исправленные версии функций разрешены для выполнения параллельно, могут возникнуть проблемы, если они используют внутренние структуры данных ядра по-разному.

На схемах последовательно отражена схема применения технологии kGraft и отражена логика использования RCU-модуля.

На рисунке 2 отражена проблема отсутствия дополнительной прослойки в логике, которая может привести к несогласованности

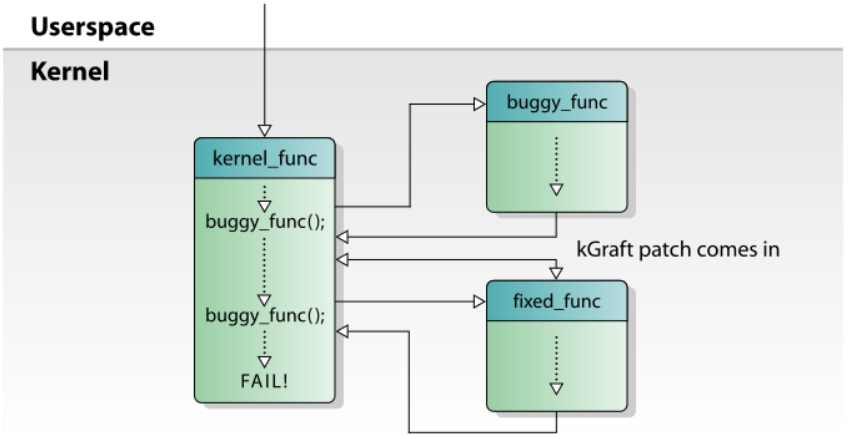

Рисунок 2.

На рисунке 3 отражена идея контролирования всех процессов, т.е каждый процесс контролируется, поэтому он выполняет исправленную функцию последовательно в рамках одного системного вызова.

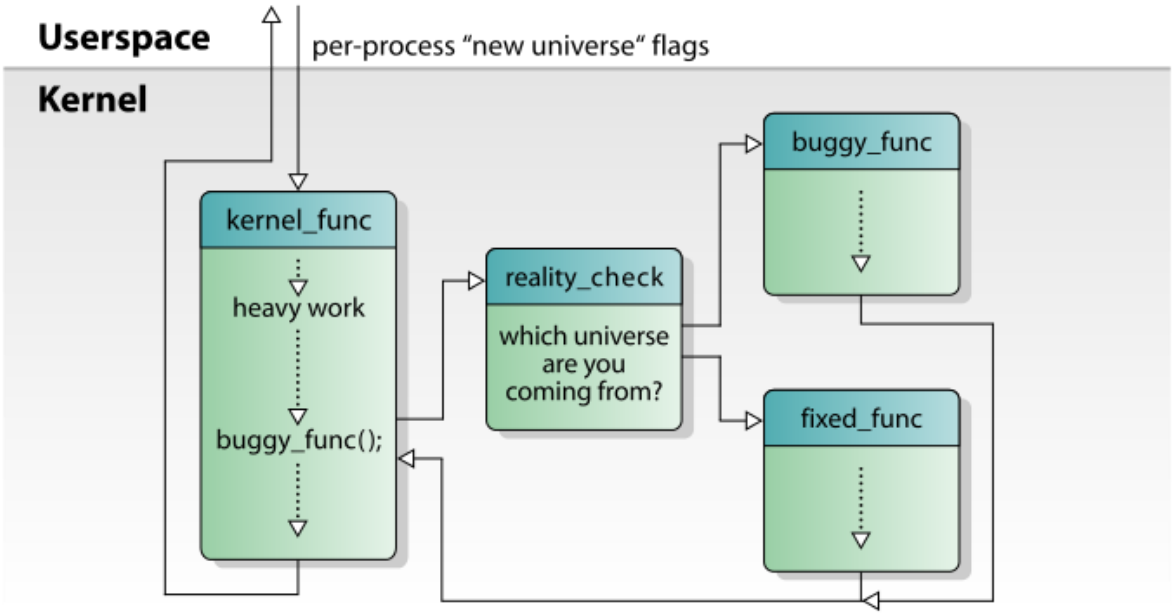

Рисунок 3

После осуществления полной миграции в новый юниверс, прослойка больше не нужна. Эта идея отражена на рисунке 4.

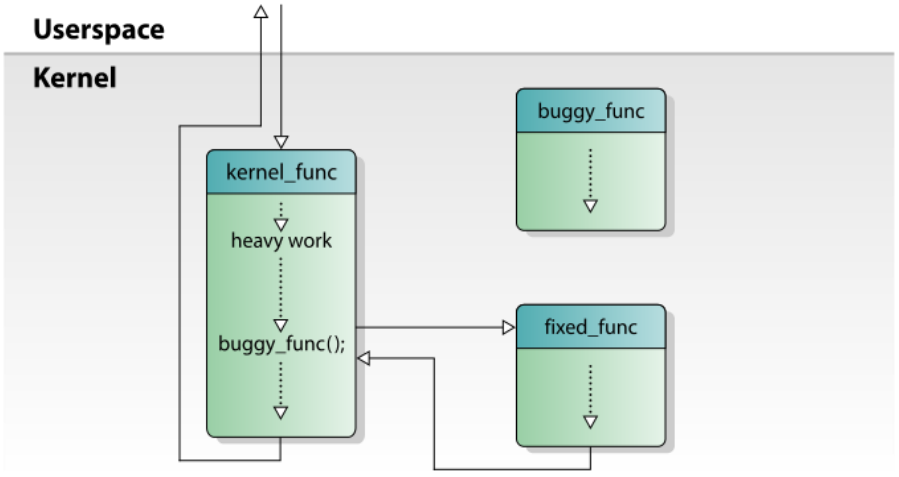

Рисунок 4.

3.4 Устройство kSplice

Ksplice принимает входной унифицированный diff и исходный код ядра, и он обновляет текущее ядро в памяти. Использование Ksplice не требует какой-либо подготовки до того, как система будет загружена изначально (например, работающему ядру не требуется специальная предварительная компиляция). Чтобы создать обновление, Ksplice должен определить, какой код в ядре был изменен патчем. Ksplice выполняет этот анализ на уровне объектного кода Executable и Linkable Format (ELF).

Чтобы применить патч, Ksplice сначала замораживает выполнение компьютера, до состояния того будто целевая программа единственна. Система проверяет, что процессоры не были в середине выполнения функций, которые будут изменены патчем. Ksplice модифицирует начало измененных функций, чтобы вместо этого указывать на новые обновленные версии этих функций и изменять данные и структуры в памяти, которые необходимо изменить. Наконец, Ksplice возобновляет работу каждого процессора, где он остановился.

Чтобы быть полностью автоматическим, Ksplice первоначально был ограничен патчами, которые не вносили семантических изменений в структуры данных, поскольку большинство патчей безопасности ядра Linux не вносят такого рода изменения. В результате оценки уязвимостей безопасности ядра Linux с мая 2005 года по май 2008 года было обнаружено, что Ksplice удалось применить исправления для всех уязвимостей ядра, обнаруженных в этом интервале. В 2009 году крупные поставщики Linux попросили своих клиентов установить обновление ядра более одного раза в месяц. Для патчей, которые вводят семантические изменения в структуры данных, Ksplice требует, чтобы программист написал короткий объем дополнительного кода, чтобы помочь применить

патч. Это было необходимо примерно для 12% обновлений за этот период времени.

Принципиальная схема работы Ksplice отражена на рисунке ниже. Данная схема являет собой высокоуровневый обзор генерации горячих патчей, основанный на изменениях исходного кода ядра.

Из рисунка 5 видно, что в результате сравнения файлов, собранных с использованием оригинальных и пропатченных исходного кодов - формируется список пропатченных функций. Дальнейшая обработка производится уже с ними.

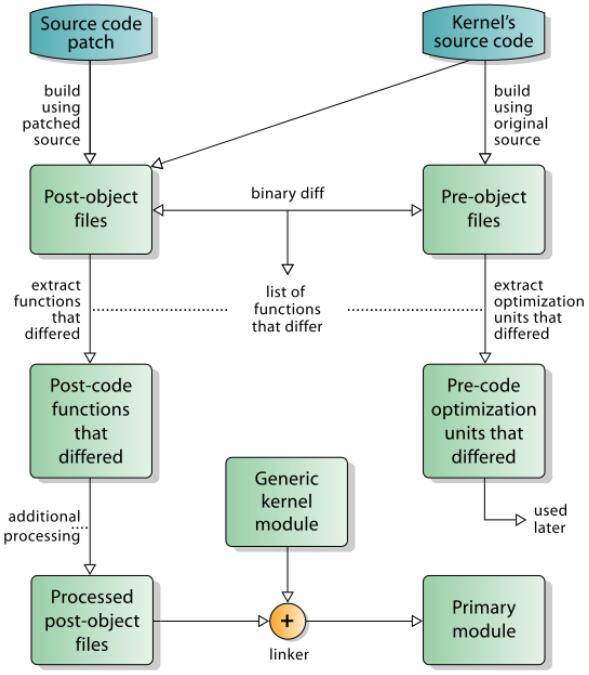

Рисунок 5.

3.5 Устройство libcare

LibCare прикладывает “горячие патчи” к любым исполняемым файлам или библиотекам Linux во время выполнения без необходимости перезапуска ваших приложений. В отличие от перечисленных выше технологий, он чаще применяется к пользовательским приложениям. Но кроме этого, он также используется и для живого обновления ядра Linux (https://kernelcare.com)

Принцип работы libcare следующий:

- как исходный, так и исправленный исходный код переводится на ассемблер
- сравниваются соответствующие файлы ассемблера и генерируется новый инструментированный ассемблерный код, где патчи хранятся в специальных разделах ELF
- инструментированный ассемблерный код компилируется с использованием системы построения целевого проекта, в то время как секции патчей ELF собираются в бинарных файлах
- бинарные файлы патчей извлекаются из секций ELF

4. Технология Live Patching средствами проекта virtuozzo/nsb

4.1 Необходимые понятия

4.1.1 Ptrace

Системный вызов ptrace () представляет собой способ, посредством которого один процесс («трассировщик») может наблюдать и контролировать выполнение другого процесса («трассируемого»). Также при помощи этого системного вызова можно проверять и изменять трассируемые память и регистры. Как правило этот системный вызов используют для создания точек прерывания и контроля исполнения системных вызовов.

Для трассирования необходимо привязаться к целевому потоку. Вложения и последующие команды относятся к потоку: в многопоточном процессе каждый поток может быть индивидуально подключен к (потенциально другому) трассеру или оставлен без привязки и, следовательно, не отлажен. Поэтому «tracee» всегда означает «(один) поток« никогда »(возможно, многопоточный) процесс». Команды Ptrace

всегда трассирует, используя вызов формы ptrace (PTRACE_foo, pid, ...), где pid - это идентификатор потока соответствующего потока Linux.

4.1.2 ELF формат

ELF(Executable and Linkable Format) - формат исполняемого файла, используемый в большинстве современных UNIX-подобных операционных систем.

На рисунке 1 отражена схема исполняемого образа программы формата ELF.

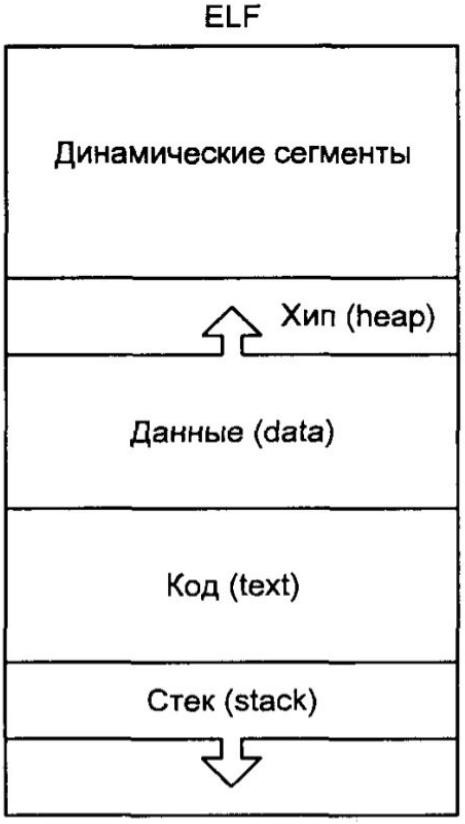

Рисунок 1

Данный формат различает несколько типов файлов:

- Исполняемый файл
- Разделяемый объектный файл
- Перемещаемый файл

Они различаются по типу хранимой в них информации.

Таким образом, исполняемый файл хранит в себе полное описание, необходимое системе для воссоздания полного образа процесса.

Перемещаемый и разделяемый объектный файлы хранят в себе инструкции и данные, которые могут быть связаны с другими объектными

файлами. В результате этого связывания могут быть сформированы исполняемые или объектные файлы. Но в случае с с разделяемым объектным файлом возможна следующая ситуация: уже на этапе запуска некоторой программы операционная система свяжет его с исполняемым файлом динамически, результатом связывание станет новый исполняемый образ программы.

Файл данного формата имеет определенную структуру. Она отражена на рисунке.

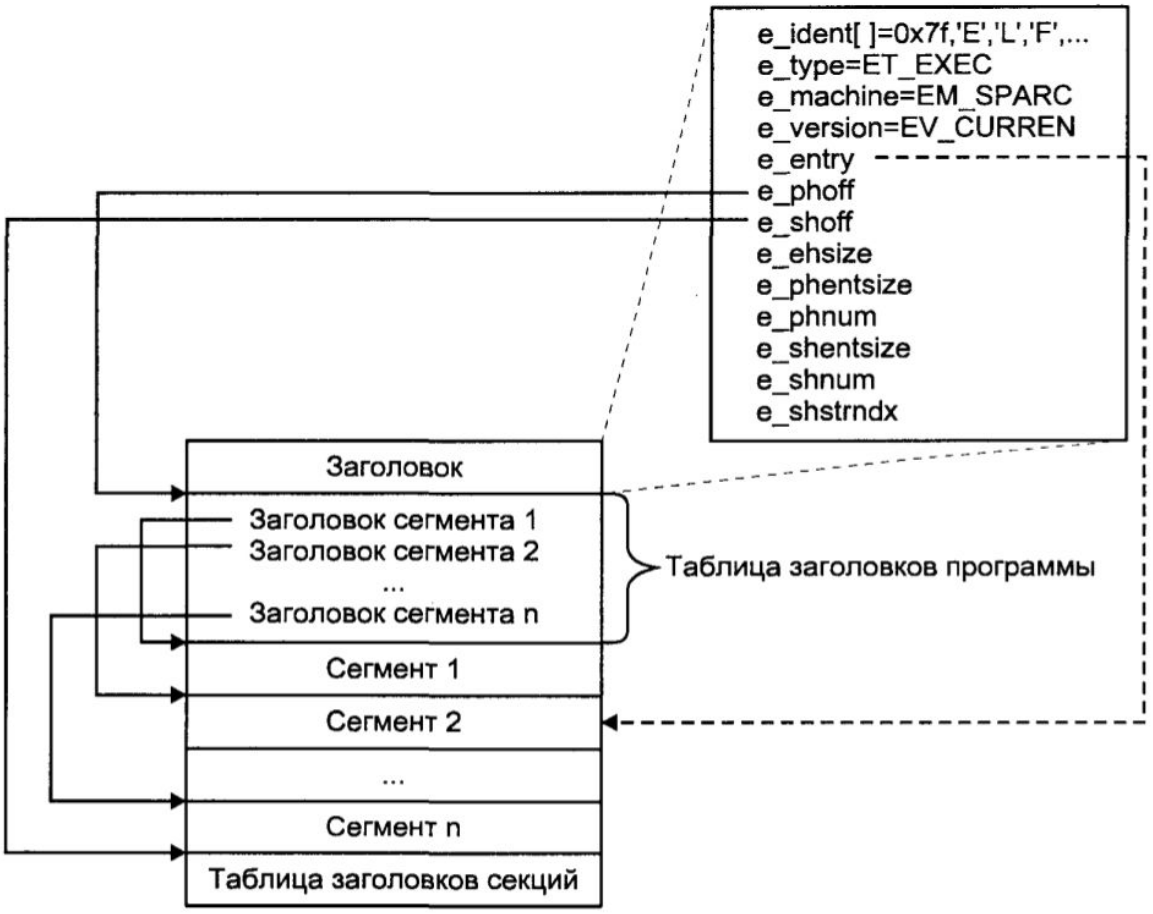

Рисунок 2.

Структура файла в формате ELF отражена в заголовке, который расположен в фиксированном стандартом месте. Таким образом, заголовок задает общую структуру файла, расположение отдельных частей и их размеры.

Структура заголовка отражена на рисунке 2.

4.2 Обзор nsb

Технология ReadyKernel Userspace представлена набором инструментов под общим названием nsb для создания и применения живых патчей. Живой патч - это разделяемый объект, хотя и необычный, он не может быть загружен динамическим компоновщиком. По этой причине активные патчи загружаются из внешних процессов с помощью инструмента nsb. Поддерживая архитектуру x86_64, эти инструменты могут заменять части логики процесса, то есть выполняемой одно- или многопоточной программы, на лету, не останавливая или не перезапуская ее. Живые патчи могут применяться к любому исполняемому процессу, включая:

- статически слинкованные бинарные файлы,
- динамически слинкованные бинарные файлы,
- разделяемые библиотеки.

Логику процесса можно заменить на функциональном или на более высоком уровне. То есть одна или несколько функций процесса могут быть изменены в соответствии со следующим алгоритмом:

1. Новые функции добавляются в адресное пространство процесса.
2. Ссылки на текущие данные процесса восстанавливаются в новых функциях.
3. Выполнение процесса перенаправляется от старых функций к новым с помощью инструкций JMP.
4. Исходя из вышесказанного, пользовательское пространство в реальном времени исправляет следующие требования к коду:
    a. Атомарная (для процесса) замена исполняемого кода.

b. Обслуживание ссылок на статические и глобальные переменные, существующие в процессе.

4.3 Принцип прикладывания патчей средствами nsb

Процесс патчинга процесса имеет несколько этапов:

- Останов процесса с помощью ptrace в безопасном месте
- Загрузка и применение патча к исполняемому коду процесса
- Восстановление работы процесса с точки останова

4.3.1 Прекращение выполнения процесса в безопасной точке извне

После прекращения процесса (то есть все его потоки остановлены), nsb проверяет стеки всех потоков через API libunwind, чтобы убедиться, что части кода, которые будут изменены патчем, не выполняются. В противном случае nsb отпускает все потоки процессов, ждет короткое время и останавливает их снова. Количество попыток ограничено, а задержка увеличивается после каждой попытки повтора.

4.3.2 Загрузка и применение патча к исполняемому коду процесса

Патч загружается путем вызова системных вызовов mmap в контекст целевого процесса. Паразитный код вставляется в процесс через библиотеку libcompel.

После загрузки патч связывается с целевым процессом на основе информации, добавленной к патчу во время его создания.

Связывание означает, что выполнение процесса перенаправляется от старых функций к новым с помощью инструкций JMP, а также любого из следующего:

- Перенаправление ссылок на глобальные функции и переменные в соответствии с алгоритмом динамического компоновщика.
- Перенаправление ссылок для обработки статических данных.
- Перенаправление ссылок на статические функции в исполняемом коде процесса.

4.3.3 Освобождение процесса

После загрузки патча выполнение кода перенаправляется, и все ссылки в патче инициализируются, выполнение процесса возобновляется с точки остановки через системный вызов ptrace. На выходе имеется исправленная, т.е. пропатченная версия исходной программы.

4.4 Использование diff для оценки возможности создания живых патчей

diff, необходимый для создания патча, в большинстве случаев, может быть любым изменением логики процесса. Тем не менее, есть определенные кейсы, которые не могут быть обработаны:

- Добавление новых статических или глобальных переменных. Причина: новая переменная может содержать состояние, которое требуется новой логике, но которое не существует в текущем процессе.
- Изменение размеров статических или глобальных переменных или структур данных. Причина: после исправления часть состояния процесса, накопленная во время выполнения, может быть потеряна.

- Переупорядочивание вложенных переменных в структурах. Причина: аллоцированные данные представлены в другом порядке в процессе.

4.5 Создание живых патчей

Живые патчи создаются из объектных файлов с измененным кодом с помощью генератора. Этот инструмент добавляет в объектный файл раздел ELF с именем «vzpatch» с информацией, необходимой для применения патча. Важно отметить, что генератор рассматривает все функции в патче как новые и будет создавать переходы для всех из них. Поддерживаются два типа живых патчей: «ручной» и «автоматический».

4.5.1 Создание патчей вручную

Этот тип полезен опытным разработчикам, которым необходимо быстро создать компактный патч.

Чтобы создать ручной патч, разработчику необходимо предоставить генератору следующее:

- целевой объектный файл должен содержать информацию об отладке
- патч должен быть собран как shared library.

4.5.2 Автоматическое создание патчей

Патчи этого типа создаются автоматически, хотя разработчику по-прежнему необходимо просмотреть diff, чтобы узнать, можно ли создать патч из него. Автоматические патчи более крупнозернистые, т. е. создаются из единицы компиляции - объектного файла. Автоматический патч включает весь исходный файл, даже если была изменена только одна функция. Чтобы создать автоматический патч, разработчику необходимо предоставить генератору следующее:

- объектный целевой файл (должен содержать информацию об отладке),
- патч, построенный как разделяемая библиотека
- список всех объектных файлов, из которых построен патч; все эти объектные файлы должны быть построены с параметрами «-g -ffunction-sections -fdata-sections».

4.6 Применение патчей к процессам

Процедура применения патча включает следующие шаги:

- Процесс (все его потоки) останавливается
- Патч добавляется в адресное пространство процесса.
- Патч настроен:
    - Инициализируются ссылки на внешние глобальные переменные и функции.
    - Ссылки на оригинальные статические переменные и функции инициализируются (только для патчей, созданных вручную).
    - Ссылки на внешние статические переменные и функции вставляются в код (только для патчей созданных автоматически).

В начало старых функций вставляются инструкции JMP от старых функций к новым.

В настоящее время инструмент nsb поддерживает следующие команды:

- **patch**, применить патч к процессу,
- **revert**, удалить патч из процесса,
- **list**, список применяемых исправлений,
- **check**, может ли патч применяться к процессу.

Следующие процессы не поддерживаются (работоспособность процессов не гарантируется после применения патча):

- C-сопрограммы (которые используют setcontext, makecontext и т. Д.),
- процессы, которые используют GIMPLE (в частности, созданные с помощью применения оптимизаций во время связывания объектных файлов)
- процессы, которые совместно используют адресное пространство с другими процессами.

4.7 Ограничения по технологии Live Patching

Несмотря на то, что патч всегда можно создать и применить к процессу, важно понять, что некоторые процессы не гарантируют работу после исправления. Некоторые ограничения связаны с созданием патча, а некоторые - с приложением.

4.7.1 Ограничения создания патчей

Следующие ограничения могут быть автоматически обнаружены в информации об отладке при создании патча:

- изменение размера переменной / структуры,
- изменение порядка атрибутов структуры,
- изменение переменной переменной из-за оптимизации компилятора («var -> const» и «const -> var»)
- использование плагинов gcc GIMPLE (в частности, те, которые построены с оптимизацией времени связи).

4.7.2 Ограничения применения патчей

Следующие ограничения могут быть обнаружены автоматически при применении патча:

- невозможность остановить процесс за пределами исправленных функций.

Следующие ограничения не могут быть обнаружены при применении патча (строгие ограничения, которые должны соблюдаться системным администратором):

- использование сопроцессов в этом процессе.

5. Постановка проблемы

5.1 Формулирование проблемы

На данный момент nsb реализован для патчинга 64-битных процессов собранных под архитектуру x86_64. Как уже упоминалось, nsb основан на инструментировании, что позволяет относительно легко портировать его на другие типы архитектур. На данный момент отсутствует реализация nsb для 32-битных процессов собранных под архитектуру x86.

5.2 Постановка задачи

Задачей является создание удобного интерфейса для портирования nsb на другие типы архитектур и расширение возможностей nsb для работы с 32-битными приложениями.

6. Решение задачи

6.1 Исходная платформа

В качестве исходного проекта, выступил проект nsb/virtuozzo. Он представляет собой проект предназначенный для инструментированного живого патчинга программ. На момент приступления к решению задачи поддерживал архитектуру x86-64

6.2 Ход решения задачи

1. Установка и запуск nsb
2. Написание мануальных тестов для исследования принципов работы nsb
3. Тестирование имеющегося проекта на архитектуре x86
4. Определение хода работы для портирования
5. Построение алгоритма портирования и его реализация

6.2.1 Установка и запуск nsb

Для установки проекта необходимо предустановить на компьютер следующие библиотеки и приложения:
Generic:

- libtool
- automake
- gcc
- make

C:

- libprotobuf-c
- libunwind
- libcompel (part of CRIU: https://github.com/xemul/criu)
- libelf
- protobuf-c-compiler

Python (2.7):

- pyelftools (>=0.24)
- protobuf
- protobuf-compiler

После установки nsb необходимо запустить приложение, перейдя в корневую директорию проекта и запустив из терминала команду: make check.

По запуску этой команды проект скомпилируется и запустится тестовый движок, применяющий nsb к заранее, подготовленным патчам и целевым программам.

После формируется статистика, показывающая какие тесты прошли, а какие нет. Кроме статистики будут сформированы log-файлы, в них содержится информация о запущенных командах и результатах их работы.

6.2.2 Написание мануальных тестов для исследования принципов работы nsb

Для изучения принципов работы nsb и для дальнейшего тестирования были написаны мануальные тесты, не входящие в состав изначального тестового модуля. Они представляют из себя исполняемый файл(целевая программа) и разделяемую библиотеку(патч). Для того чтобы запустить их вручную из консоли, необходимо выполнить определенные команды. Их нужно достать из log-файла, взяв команды с пометкой “Execute”. Это легко сделать, воспользовавшись утилитой grep.

6.2.3 Тестирование исходного проекта на архитектуре x86

Для того, чтобы определить алгоритм, согласно которого исходный проект будет портирован на x86 архитектуру, исходный проект запускается на виртуальной машине, в точности на 32-битном Centos.

В процессе предварительной установки необходимых компонент, была обнаружена невозможность портирования исходного проекта на 32-битные машины. Невозможность портирования, объясняется отсутсвием в libcompel поддержки x86 архитектуры.

В связи с отсутствием возможности поддержать x86 архитектуру, было принято решение осуществить поддержку патчинга 32-битных приложений.

Дальнейшее тестирование исходного проекта, продолжалось для 32-битных целевых программ и патчей.

Алгоритм поддержки x86 архитектуры основывался на вышеописанном методе тестирования и дальнейшего исправлении получаемых, в ходе этого тестирования, ошибок.

6.2.4 Определение хода работы для поддержки x86 архитектуры

Базовый метод определения алгоритма, основывался на тестировании и исправлении ошибок.

6.3 Построение алгоритма и его реализация

6.3.1 Работа с generator и protobuf

Решена проблема внедрения ветвления поведения nsb в зависимости от архитектуры.

Эта проблема была решена следующим образом: в proto-файл, содержащий информацию об исходной целевой программе и патче были добавлены поля - отвечающие за тип архитектуры, на котором собраны исходные файлы цели и патча.

Эти поля заполняются в коде генератора, с использованием библиотеки pyelftools, а возвращаются они из секции header ELF-файлов.

6.3.1.1 Изменения generator для работы с файлами, собранными на архитектуре x86

При анализе исполняемого файла в формате ELF, собранного на архитектуре x86 была выявлена проблема лишних считываемых символов. Экспериментальным и теоретическим путем было установлено предназначение этих символов. Был сформирован список игнорируемых символов, и ненужные символы, не предназначенные для дальнейшей работы были занесены в этот список. Как следствие эти символы игнорировались.

6.3.2 Работа с patcher

6.3.2.1 Обработка данных, полученных с генератора

Поля с необходимыми данными, считанные в генераторе, передаются в патчер средствами google-protobuf.

В файле context.h в структуру patch_info_s мы добавили поле отвечающую за архитектуру. Наличие этого поля объясняется дальнейшим введением ветвления в зависимости от архитектуры.

6.3.2.2 Принцип введения ветвления

Архитектурно-зависимые части patcher находятся в разных частях кода. Поэтому невыгодно каждый раз ставить ветвление в соответствующей части кода. Лучше сразу задать правильную логику, прописав условие один раз. Было принято решение создать структуру с колбеками. Это структура описана в файле context.h - struct arch_cb. В этой структуре хранятся указатели на архитектурно зависимые функции. Таким образом, один раз при инициализации контекста, инициализируется эта структура, и вызову соответствующих функций предшествует обращение к структуре. При инициализации контекста процесса, производится проверка архитектуры, два варианта: x86_64 и x86. Проверка производится по полю распакованному из generator и заполненному в структуре patch_info_s. И далее инициализируется используемый указатель на структуру arch_cb, содержащийся в структуре, отвечающей за контекст. Он инициализируется специально подготовленными структурами x86_64_cb или x86.

6.3.2.3 Предварительные предположения об архитектурно-зависимых функциях

Для минимизации поиска архитектурно-зависимых функций экспериментальным путем было предварительно проведено исследование проекта на предмет наличия модулей и функций, потенциально зависимых от архитектуры.

Одной из потенциально зависящих от архитектуры функций была обертка над ассемблерной инструкцией jmpq. В этой обертке был использован исключительно opcode инструкции и в ходе чтения мануалов было выяснено, что данная инструкция одинаково работает на обеих архитектурах: x86, x86_64. После общего обзора проекта на предмет наличия требуемых модулей был начат их поиск экспериментально.

6.3.2.4 Поиск архитектурно-зависимых функций и их обработка

Поиск архитектурно зависимых функций производился экспериментальным путем. Запускались 32-битные исполняемые целевые программы и анализировались выводимые ошибки. После нахождения этих функций анализировался их принцип работы и логика. Далее они переписывались под x86, и сохранялись в файле архитектурно зависимых функций.

Путем раскрутки стека ошибок архитектурно-зависимые функции были найдены.

В перечень архитектурно-зависимых модулей вошли функции, работающие с адресами. Они являются архитектурно-зависимыми по причине того, что на архитектуре x86-64 адреса имеют разрядность 64, на x86 - адреса имеют разрядность 32. Для правки этих функций указатели, являющиеся адресами были реализованы 32-битными. При правке подобных функций учитывался тот факт, что системный вызов ptrace на архитектуре x86-64 работает 8-байтовыми кусками, и по выровненным на 8 байт адресам. Т.е. с функциями подобного рода, содержащими этот системный вызов работали примерно как с диском - читали весь сектор, меняли нужный байт, писали сектор обратно. Это необходимо при работе с целевыми 32-битными программами, которые подвергались патчингу. После функции были вынесены в специальные файлы, хранящие в себе все архитектурно-зависимые части.

Также в перечень архитектурно-зависимых модулей вошли функции содержащие функцию `int compel_syscall( ... )`, эта функция работает под разные архитектуры в зависимости от выставленных флагов. Обертки над этой функцией были также вынесены в специальный модуль.

7. Заключение

Результатом проделанной работы является, внедрение в исходный проект поддержки живого патчинга 32-битных приложений, собранных на архитектуре x86.

Во-первых, данное достижение является практическим доказательством того, что проект virtuozzo/nsb основанный на инструментировании может быть портирован для работы с целевыми программами, собранными на архитектуре не x86-64. В нашем случае мы показали это на примере программ, собранных на архитектуре x86.

Во-вторых, в проект был добавлен специальный интерфейс, позволяющий переключаться между архитектурами, в зависимости от типа архитектуры, для которой собрано запускаемое приложение. Добавление подобной функциональности позволяет с легкостью портировать этот проект на другие архитектуры в будущем.

В-третьих, в ходе исследования было выяснено, что данный проект не может применяться на архитектуре x86, но может применяться к 32-битным приложениям. Это связано с тем, что сам проект не может быть собран на архитектуре x86, в связи с отсутствием в libcompel поддержки этой архитектуры.

Таким образом, исходные возможности проекта virtuozzo/nsb были расширены и дополнены возможностью использования проекта для динамической модификации кода запущенных 32-битных приложений.

Список использованных источников

1. *Linux man pages online - http://man7.org/linux/man-pages/*
2. *Руководство пользователя Linux - http://www.opennet.ru/docs/RUS/lug/*
3. *Patch(computing) - https://en.wikipedia.org/wiki/Patch_(computing)*
4. Dynamic Software Updating - *https://en.wikipedia.org/wiki/Dynamic_software_updating*
5. *Ksplice - https://en.wikipedia.org/wiki/Ksplice*
6. *Kpatch - https://en.wikipedia.org/wiki/Kpatch*
7. *KGraft - https://en.wikipedia.org/wiki/KGraft*
8. *“Операционная система UNIX” А. Робачевский*
9. *Заразить во благо: как мы исполняем паразитный код - https://habr.com/company/virtuozzo/blog/343548/*