Мелкогранулярная рандомизация адресного пространства программы при запуске[1]

[1] *А.Р. Нурмухаметов <oleshka@ispras.ru>*
[1] *Е.А. Жаботинский <ezhabotinskiy@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[1,2,3,4] *С.С. Гайсарян <ssg@ispras.ru>*
[1] *А.В. Вишняков <vishnya@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *2119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК*
[3] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. Уязвимости программного обеспечения являются серьезной угрозой безопасности. Важной задачей является развитие методов противодействия их эксплуатации. Она приобретает особую актуальность с развитием ROP-атак. Имеющиеся средства защиты обладают некоторыми недостатками, которые могут быть использованы атакующими. В данной работе предлагается метод защиты от атак такого типа, который называется мелкогранулярной рандомизацией адресного пространства программы при запуске. Приводятся результаты по разработке и реализации данного метода на базе операционной системы CentOS 7. Рандомизацию на уровне перестановки функций осуществляет динамический загрузчик с помощью дополнительной информации, сохраненной с этапа статического связывания. Описываются детали реализации и приводятся результаты тестирования производительности, изменения времени запуска и размера файла. Отдельное внимание уделяется оценке эффективности противодействия эксплуатации с помощью ROP атак. Строятся две численных метрики: процент выживших гаджетов и оценка работоспособности примеров ROP цепочек. Приводимая в статье реализация применима в масштабах всей операционной системы и не имеет проблем совместимости с точки зрения работоспособности программ. По результатам проведенных работ была продемонстрирована работоспособность данного подхода на

[1] Работа поддержана грантом РФФИ 17-01-00600 А

реальных примерах, обнаружены преимущества и недостатки и намечены пути дальнейшего развития.

Ключевые слова: рандомизация адресного пространства; диверсификация; ASLR; ROP.

DOI: 10.15514/ISPRAS-2017-29(6)-9

Для цитирования: Нурмухаметов А.Р., Жаботинский Е.А., Курмангалеев Ш.Ф., Гайсарян С.С., Вишняков А.В Мелкогранулярная рандомизация адресного пространства программы при запуске. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 163-182. DOI: 10.15514/ISPRAS-2017-29(6)-9

1. Введение

Современное программного обеспечения попадает к конечным пользователям с некоторым количеством ошибок. Инструменты статического анализа и разнообразное тестирование не позволяют устранить из ПО все ошибки. Среди ошибок имеются такие, которые потенциально позволяют при некоторых условиях и входных данных контролировать поведение программы или раскрывать ее данные. Существует статистика (CVE [1]) о публично известных уязвимостях в ПО. Она показывает на рис. 1, что количество обнаруживаемых уязвимостей как минимум не уменьшается. Невозможно в условиях ограниченности ресурсов найти и устранить их все. Успешная эксплуатация уязвимостей может привести к утечке конфиденциальных данных и сбоям в работе информационных систем. Из этого следует, что актуальной проблемой является предотвращение эксплуатации уязвимостей.

Классификация CVE содержит большое количество видов уязвимостей. Для их эксплуатации придуманы разнообразные методы [2–7]. Техника эксплуатации также сильно зависит от механизмов защиты, применяемых в операционных системах и на аппаратном уровне (ASLR, DEP) [8–11].

Наиболее опасным механизмом эксплуатации является ROP [2] и ему подобные [4]. Идея таких методов заключается в том, что вредоносный код составляется из гаджетов — небольших фрагментов кода самой программы, каждый из которых заканчивается инструкцией передачи управления. В случае ROP — это инструкция возврата из функции. При этом адреса гаджетов размещаются подряд на стеке, чередуясь с аргументами, которые эти гаджеты снимают со стека. Из гаджетов составляются цепочки, которые позволяют выполнять произвольный код [2].

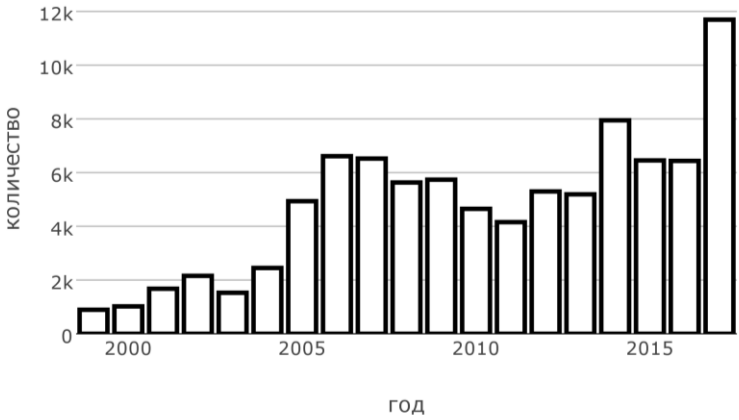

Рис. 1. Количество уязвимостей, занесенных в базу CVE с 1999 по 2017 год
Fig. 1. The number of vulnerabilities in the CVE database from 1999 to 2017 year

Таким образом, если атакующий использует уязвимость с возможностью передачи управления по произвольному адресу, то он может выполнять произвольный код в контексте атакуемой программы даже при наличии DEP (набор программных и аппаратных технологий, запрещающий приложению исполнять код из областей памяти, содержащих данные). Однако, чтобы выполнить полезный для атакующего код, нужно знать местоположение этого кода в памяти атакуемого процесса. Для построения ROP цепочки нужно знать адреса используемых гаджетов. Это означает, что для защиты от эксплуатации ошибок можно использовать рандомизацию адресного пространства процесса. Различные подходы к рандомизации адресного пространства предлагались в статьях [12–20]. Некоторые из них применяются на практике (ASLR). Однако перечисленные подходы обладают недостатками или не дают достаточной степени защиты.

Текст данной работы состоит из 6 частей. Первая глава представляет собой введение, в котором обосновывается актуальность данной работы. Во второй главе приводится обзор существующих решений. В третьей главе приводится описание предлагаемого метода и детали его реализации. Четвертая глава посвящена оценке влияния предлагаемого метода на производительность и размер программ. В пятой главе обсуждается эффективность противодействия эксплуатации уязвимостей и приводятся экспериментальные результаты. В шестой главе подводятся итоги и намечаются планы дальнейшего развития.

2. Обзор существующих решений

Из-за повсеместного присутствия программно-аппаратных защитных механизмов, предотвращающих выполнение данных (DEP), современные атаки ограничены использованием существующего кода для составления цепочек. Частично эту проблему решает технология рандомизации адресного пространства процесса (ASLR), которая позволяет задавать произвольный базовый адрес стека, кучи, сегментов кода и библиотек. Эта технология имеет два ключевых недостатка: изменяется только базовый адрес сегментов, внутренняя структура библиотеки или исполняемого файла сохраняется и относительные смещения между структурными элементами остаются постоянными; для поддержки такого механизма необходимо собирать позиционно-независимые исполняемые файлы, что отрицательно сказывается на производительности [21]. Сохранение относительных смещений внутри библиотек и исполняемых файлов приводит к тому, что атакующий может восстановить секцию кода программы или библиотеки по одному утекшему адресу. Для построения ROP цепочки необходимо знать адреса некоторого количества гаджетов. Для предотвращения вычисления всех этих адресов по одному раскрытому адресу можно сделать непредсказуемыми относительные смещения в коде.

Мелкогранулярная рандомизация во время компиляции. Существует подход по генерации диверсифицированной популяции исполняемых файлов во время компиляции и связывания [13]. Данный подход создает ряд инфраструктурных трудностей по распространению индивидуальных рандомизированных копий программы и создает непреодолимые сложности для сертификации, поскольку сертифицируется конкретный исполняемый файл. Кроме того, он обладает недостатками, связанными с возможностью утечки конкретного исполняемого файла, что компрометирует ту систему, откуда он был взят. Более того, от запуска к запуску карта памяти остается неизменной, что позволяет атакующему восстановить размещение кода в памяти в случае перезапускаемого сервиса. В свою очередь, рандомизация адресного пространства при запуске ограничивает период времени для раскрытия карты памяти процесса, а неудачные попытки, приводящие к аварийному завершению программы, делают все полученные данные бесполезными.

Мелкогранулярная рандомизация во время запуска. В рамках данной работы реализована рандомизация с гранулярностью до функций, то есть код каждой функции при запуске программы размещается в памяти по случайному адресу, после чего исправляются все упоминания адреса этой функции в других функциях. Аналогичное решение было предложено в работе Selfrando [15]. Однако их подход направлен на защиту отдельных приложений, в частности Tor Browser. Код, выполняющий рандомизацию, включается в сам исполняемый файл. Добавление необходимой для

рандомизации информации производится при помощи скриптовых оберток компилятора и компоновщика.

Сбор информации о функциях и релокациях извне компоновщика не позволяет поддерживать использование TLS. Включение кода для рандомизации в исполняемый файл увеличивает его размер и время загрузки, а кроме того мешает работе других средств защиты, так как делает исполняемый файл трудно отличимым от различных самомодифицирующихся программ, которые зачастую являются вредоносными. Подход Selfrando рассчитан на упрощение сборки отдельных приложений с поддержкой рандомизации, но плохо применим в масштабах всей системы.

Другой подход применен в инструменте XIFER [16]: библиотека, выполняющая рандомизацию, загружается при помощи LD_PRELOAD. Для рандомизации не требуется специальной подготовки исполняемого файла или библиотеки, так как вся необходимая информация получается посредством дизассемблирования. Более того, поддерживается гранулярность рандомизации на уровне базовых блоков и мельче. Это позволяет достичь большей энтропии, что делает защиту надежнее.

Однако такой подход замедляет запуск программ. Скорость обработки кода оценивается в 687 КБ/с. Это означает, что запуск приложения bash, имеющего в CentOS 7 секцию кода с размером 555 КБ, займет около одной секунды без учета используемых им динамических библиотек. Помимо этого, сильное уменьшение гранулярности рандомизации приведет к слишком частым промахам в кэше при исполнении рандомизированного кода, что уменьшит скорость работы приложения. Кроме того, дизассемблирование кода создает вероятность ошибок, особенно на больших программах, а разбиение функций на части может серьезно нарушить работу отдельных механизмов (eh_frame).

Постраничная рандомизация. При компиляции для программы генерируется позиционно-независимый код, который разбивается на фрагменты, дополняемые до целого количества страниц виртуальной памяти. Передача управления между этими фрагментами осуществляется через дополнительную таблицу (аналог таблицы глобальных смещений GOT). При запуске программы каждый фрагмент целиком загружается в случайные страницы виртуальной памяти, а адреса страниц записываются в эту таблицу.

Таким образом, хотя полученная энтропия и ниже, чем при мелкогранулярной рандомизации, но она все равно намного выше, чем при использовании обычного ASLR. При этом код не модифицируется при загрузке в память, а значит, запуск программ происходит немного быстрее. Одна физическая страница с кодом может разделяться между процессами, в то время как при мелкогранулярной рандомизации каждый процесс вынужден хранить свою рандомизированную копию библиотеки, что увеличивает использование памяти.

Первой попыткой такой реализации был oxymoron [17]. Он реализует постраничную рандомизацию для x86-64. Исполняемый код разбивается на страницы, в конце при необходимости добавляется межстраничная передача управления на следующую страницу. Адрес таблицы для межстраничной передачи управления хранится в сегментном регистре, значение которого архитектура не позволяет читать. Поэтому атакующий не может просто узнать адрес этой таблицы и считать ее для построения вредоносного кода. Каждая библиотека имеет свою таблицу, а межбиблиотечные вызовы осуществляются через функции-заглушки, которые загружают адрес таблицы конкретной библиотеки в сегментный регистр.

Другая реализация представлена в инструменте pagerando для ARM [18]. Функции переупорядочиваются для исключения лишних межстраничных переходов в конце страницы без чрезмерного количества неиспользуемой памяти в конце каждой страницы. Адрес таблицы хранится в регистре общего назначения и при межбиблиотечных вызовах сохраняется на стеке. Никаких мер для предотвращения утечки этого адреса не предпринимается. Влияние реализации постраничной рандомизации на производительность оценивается в 1-5 %.

Рандомизация во время работы программы. Бывают ситуации, при которых серверный процесс для обработки каждого запроса дублирует себя при помощи вызова fork. В таком случае атакующий может угадывать карту адресного пространства, не заботясь о стабильности работы атакуемого процесса. В случае завершения сервер создает новый процесс с той же самой картой адресного пространства, после чего можно продолжать перебор. Авторы предлагают отслеживать все указатели в памяти процесса при помощи инструментации машинного кода и динамического анализа помеченных данных, а затем с использованием этой информации заново выполнять обычный ASLR в дочернем процессе после выполнения fork. Анализ помеченных данных замедляет работу в 10-20 раз.

В нескольких работах для защиты предлагается использовать более частую перерандомизацию адресного пространства процесса. В работе [19] предлагается проводить рандомизацию перед системными вызовами, получающими информацию извне и следующими после вызовов, выводящих информацию. Таким образом, собранные данные о состоянии процесса устаревают к моменту, когда атакующий может воздействовать на поведение процесса. На SPEC2006 такой метод замедляет работу в среднем всего на 2 %, но это относительно компиляции с ключом -Og, с которым этот набор тестов работает на треть медленнее, чем с -O2. Кроме того, требуется модификация ядра ОС, а исполняемый файл должен быть дополнительно аннотирован для отслеживания указателей, что возможно только для программ, написанных на чистом Си и с некоторыми ограничениями на работу с указателями.

В работе [20] предлагается проводить рандомизацию через фиксированные интервалы времени и параллельно с работой основной программы из отдельного потока. Рандомизация производится на уровне функций с помощью таблицы символов, которую компоновщик оставляет в исполняемом файле, и дизассемблированного при ее помощи кода. Вместо отслеживания указателей происходит изменение их семантики. Указатель хранит индекс адреса в глобальной таблице. Адреса возврата на стеке шифруются уникальным для каждой функции ключом, который меняется при перерандомизации. Вызовы функций реализуются через относительные переходы, то есть программа статически связывается в памяти при запуске. Это устраняет GOT как возможный источник утечки адресов, но предотвращает использование dlopen и исключений Си++. Данный метод замедляет и запуск программ (из-за дизассемблирования и связывания), и выполнение (из-за шифрования адресов возврата). На SPEC2006 с единственной рандомизацией при запуске замедление составляет в среднем 8 %. При перерандомизации каждые 200 мс замедление составляет 13.5 % в среднем. Такое сравнительно небольшое замедление достигается благодаря тому, что рандомизация выполняется параллельно.

3. Предлагаемый метод и его реализация

В рамках данной работы предлагается реализации мелкогранулярной, с гранулярностью не крупнее функций, рандомизации адресного пространства программ при запуске. Для реализации этого подхода при сборке программ исполняемые и библиотечные файлы дополняются информацией о границах функций и релокациях (упоминаниях адресов кода или данных в программе, например, адреса одной функции в коде другой). При запуске программы системный динамический загрузчик использует эту информацию для случайного размещения отдельных функций в памяти. Данный метод требует доступа к исходному коду и процессу сборки. Рандомизация выполняется только при загрузке программы. Адресное пространство не изменяется, например, при вызове fork. Кроме того, предлагаемая рандомизация не затрагивает адресное пространство ядра.

Для реализации мелкогранулярной рандомизации на этапе запуска программы в динамический загрузчик и в инструментарий для сборки программ были внесены изменения. Рандомизация реализована для архитектуры x86-64 и операционной системы CentOS 7, использующей ELF как основной формат исполняемых и библиотечных файлов. Для минимизации потенциальных проблем совместимости в этот формат не было внесено никаких изменений. Необходимая для рандомизации информация хранится в дополнительной секции, которая игнорируется при использовании стандартного динамического загрузчика.

3.1 Хранение информации для рандомизации

Для выполнения мелкогранулярной рандомизации на уровне функций необходимо знать границы этих функций и релокации. Для хранения этой информации в ELF файле размещается дополнительная секция. В этой секции хранятся информация о релокациях, виртуальные адреса функций без рандомизации, их длина и выравнивание.

Кроме того, из соображений эффективности некоторые адреса в структуре самого ELF файла и в определенных частях программы также записаны в таблицу релокаций. К таким релокациям относятся адреса в секции eh_frame и адреса динамических символов.

Помимо дополнительной секции в сегмент NOTE добавляется запись, содержащая виртуальный адрес загрузки этой секции в память. Этот сегмент предназначен для хранения произвольной дополнительной информации, все загрузчики и инструменты просто игнорируют неизвестные им записи. Таким образом, ELF файл, собранный с поддержкой рандомизации, может быть загружен любым стандартным динамическим загрузчиком, а вся дополнительная информация будет просто проигнорирована.

3.2 Модификации динамического загрузчика

В Linux на x86-64 при запуске исполняемого ELF файла ядро операционной системы загружает в память все загружаемые сегменты из этого файла и из упомянутого в нем динамического загрузчика. Выполнение начинается с точки входа динамического загрузчика, который работает в контексте самого процесса. Он загружает все требуемые динамические библиотеки, подготавливает программу к запуску и передает управление на точку входа самой программы.

Для реализации рандомизации при запуске программы необходимо внесение изменений в динамический загрузчик. Динамический загрузчик является частью библиотеки glibc. В динамический загрузчик была добавлена функциональность, которая находит в ELF файле, загруженном в память, дополнительную секцию. С помощью найденной дополнительной секции выполняется случайное переупорядочивание функций загружаемого файла. После этого выполняется проход по списку релокаций и их исправление. Если дополнительная секция отсутствует, то рандомизация не совершается. Таким образом, модифицированный загрузчик может загружать программы и библиотеки, собранные без поддержки рандомизации.

Описанная функциональность модифицирует код программы. Код, как правило, загружается в память, для которой запрещается запись, поэтому для нее временно разрешается изменение. Если в системе присутствует система ограничения доступа (SELinux, PAX), то может потребоваться ее дополнительная настройка для разрешения такого поведения.

Дополнительная секция и выделенная во время рандомизации память освобождаются перед передачей управления программе. Таким образом, после завершения рандомизации в памяти процесса не остается никакой дополнительной информации, утечка которой могла бы раскрыть размещение функций.

3.3 Модификация инструментария сборки

Для создания дополнительной секции были внесены изменения в статический компоновщик. Информация о границах функций получалась от компилятора с помощью указания ключа командной строки -ffunction-sections. Статический компоновщик собирает информацию обо всех релокациях при их разрешении в процессе связывания и сохраняет ее в дополнительной секции. Кроме того, Некоторые типы релокаций, например, из TLS и eh_frame потребовали нетривиальной обработки, которую сложно провести вне компоновщика, что предотвратило их поддержку в схожей работе [15].

Для удобства использования статическому компоновщику был добавлен ключ командной строки, включающий поддержку рандомизации. При указании этого ключа при промежуточной сборке нескольких объектных файлов в один предотвращается слияние секций, содержащих код, и дополнительная секция не создается. При статической сборке этот ключ игнорируется. Рандомизация статически собранных файлов не поддерживается.

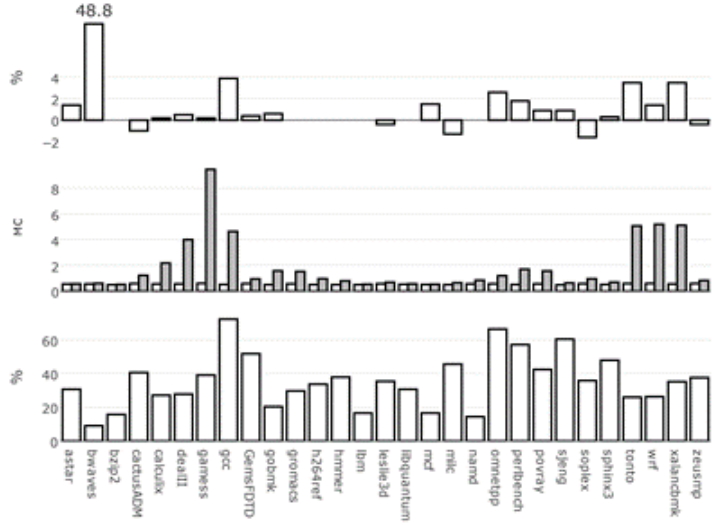

Рис. 2. Результаты тестов SPEC2006 на изменение производительности, увеличение времени запуска и увеличении размера (сверху вниз)
Fig. 2. Test results of SPEC2006 for performance, runtime startup and size increasing (from top to bottom)

4. Измерение производительности

Тестирование проводилось на системе CentOS 7. Процессор Intel Core i7-4790 (3.6 GHz), 16 ГБ оперативной памяти. Проверка корректности работы была проведена на промышленном наборе тестов SPEC2006 и минимальной сборке операционной системы CentOS 7. SPEC2006 отрабатывает корректно, почти все проверенные пакеты CentOS 7 проходят все тесты, аномалий в работе программ не наблюдается. Отдельные приложения могут модифицировать свой собственный исполняемый файл, что приводит к ошибкам при использовании рандомизации, но это необходимо исправлять на уровне самих приложений, а не инструментария, реализующего рандомизацию. Использовалась стандартная система сборки и выполнения этого набора тестов. Для одной и той же конфигурации SPEC2006 запускался со стандартными компоновщиком и загрузчиком без рандомизации и с модифицированными с включенной рандомизацией.

Время выполнения тестов. Изменение времени работы тестов приведено на рис. 2 на графике измерение производительности. Среднее геометрическое замедление составляет примерно 1.5 %. Большинство тестов показало незначительное изменение времени выполнения, за исключением bwaves. Для него замедление составило более 40 %, это объясняется тем, что для этого приложения критически важна локальность распределения кода. Некоторые из тестов по такой же причине даже показали незначительные улучшения производительности по сравнению с нерандомизированной версией программы.

Время загрузки программ. Время загрузки измерялось путем многократного запуска программ с прекращением исполнения перед передачей управления на точку входа. На рис. 2 приведены результаты измерений. Несмотря на достаточно большое относительное замедление процесса загрузки программ, в отдельных случаях эта величина достигает 10 раз, время загрузки остается пренебрежимо малым по сравнению с типичным временем работы нетривиальных программ. Самый медленный запуск программы из набора тестов при замедлении 15.3 раза занимает всего 9.5 миллисекунд.

Размер исполняемого файла. На рис. 2 приводится график изменения размера исполняемого файла при рандомизации. В среднем размер исполняемого файла увеличивается на 50 %, максимальная величина – 73 %. Следует отметить, что хранимая в исполняемом файле дополнительная информация занимает место только на диске. После завершения рандомизации, занимаемая ей память, освобождается. С учетом размера современных дисков и типичного суммарного размера исполняемых и библиотечных файлов в системе (3 ГБ на тестовой системе) – это не является проблемой.

5. Противодействие эксплуатации уязвимостей

Актуальным вопросом для данной работы является исследование эффективности реализованного метода по способности противодействовать эксплуатации уязвимостей. Для ее оценки существует два принципиальных подхода. Первый, который используется в большинстве статей аналогичной тематики, заключается в теоретико-логическом обосновании эффективности. Второй метод заключается в экспериментальной проверке реализованных методов защиты с приведением результатов статистических измерений. Наиболее полное исследование второго типа опубликовано в работе [22]. В данной работе прибегнем к второму методу оценки эффективности реализованного метода.

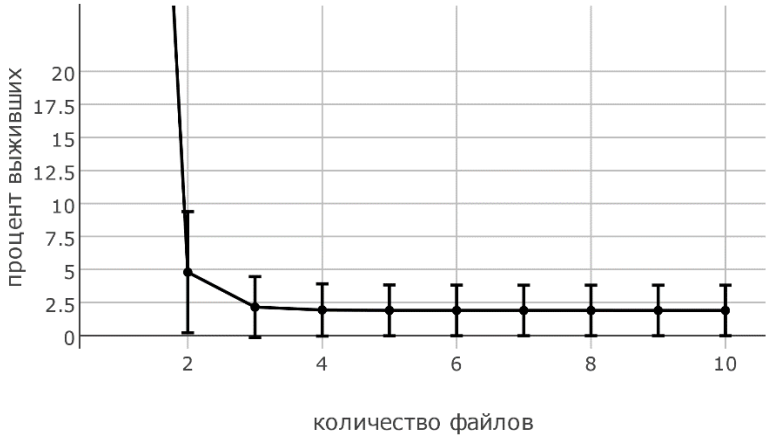

Рис. 3. Среднее относительное количество выживших гаджетов в зависимости от размера популяции
Fig. 3. The average percentage of survived gadgets depending on the size of population

Предполагаемый сценарий атаки на приложение заключается в следующем: атакующий имеет в своем распоряжении исполняемый файл приложения; из гаджетов этого файла строится ROP цепочка; данную цепочку пытаются выполнить на других экземплярах приложения.

Экспериментальная оценка производилась с помощью нескольких серий измерений. Исследуемым набором приложений были исполняемые файлы из стандартной минимальной установки CentOS 7, располагаемые в директориях /usr/bin и /usr/sbin. Для корректности полученных результатов брались только файлы, собранные без поддержки позиционно-независимого кода. Тестовый набор состоит из 487 файлов.

Для исследования необходимо было сохранять состояние адресного пространства приложения после момента его перемешивания. Это было

сделано с помощью сохранения дампа памяти процесса. Дамп памяти сохраняется в ELF формате, где каждому сегменту создается своя отдельная секция. Однако по умолчанию сохранять секции с кодом в дамп памяти не требуется, поскольку эта секция остается неизменной и всегда доступна в файле на диске. Для записи всех сегментов рабочей памяти процесса были внесены изменения в алгоритм сохранения дампов памяти отладчика gdb. С его помощью для каждого файла из тестового набора было получено по 10 дампов памяти. Собранные дампы памяти образовали вместе с оригинальными файлами тестовую популяцию над которой производились все эксперименты с помощью классификатора гаджетов [23].

Поиск и классификация гаджетов. Поиск гаджетов осуществляется при помощи инструмента с открытым исходным кодом ROPgadget [24]. Инструмент находит инструкции передачи управления в исполняемых секциях программы и дизассемблирует несколько байт, предшествующих найденным инструкциям. Все успешно дизассемблированные блоки инструкций добавляются в список потенциальных гаджетов.

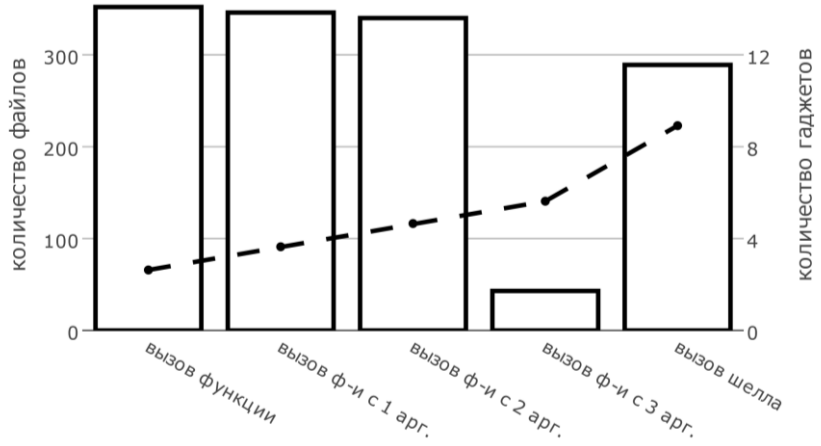

Рис. 4. Количество успешно созданных цепочек и их средний размер для различных модельных примеров ROP цепочек
Fig. 4. The number of successfully created chains and their average size for different model examples of ROP chains

Полученные кандидаты в гаджеты классифицируются согласно семантическим типам, описанным в статье [23]. Инструкции гаджета транслируются в промежуточное представление, которое в дальнейшем интерпретируется. Во время интерпретации отслеживаются обращения к регистрам и памяти на чтение и запись. Начальные считанные значения генерируются случайным образом. В результате интерпретации получаются начальные и конечные значения регистров и памяти, которые ограничивают

список возможных семантических типов, которым удовлетворяет гаджет. После этого производится еще несколько запусков процесса интерпретации с различными входными данными. В результате остаются только те типы, которым удовлетворял гаджет на всех запусках.

Классифицированные гаджеты сохраняются в базу данных вместе с дополнительной информацией о типе гаджета, об его адресе, о параметрах гаджетов и побочных эффектах. С помощью полученных баз данных возможно узнать, существует ли в данном файле на заданном адресе гаджет, какие параметры и побочные эффекты у гаджета на заданном адресе и так далее.

Оценка количества выживших гаджетов. Введем определение термину выживший гаджет. Пусть имеется некоторая популяция разных версий дампов памяти одной и той же программы. Тогда будем называть для нее выжившим гаджетом такой гаджет исходного исполняемого файла, который находится по одному и тому же адресу в каждом экземпляре популяции. Выжившие гаджеты важны для исследования по причине того, что составленная из таких гаджетов ROP цепочка работоспособна на каждом экземпляре в популяции.

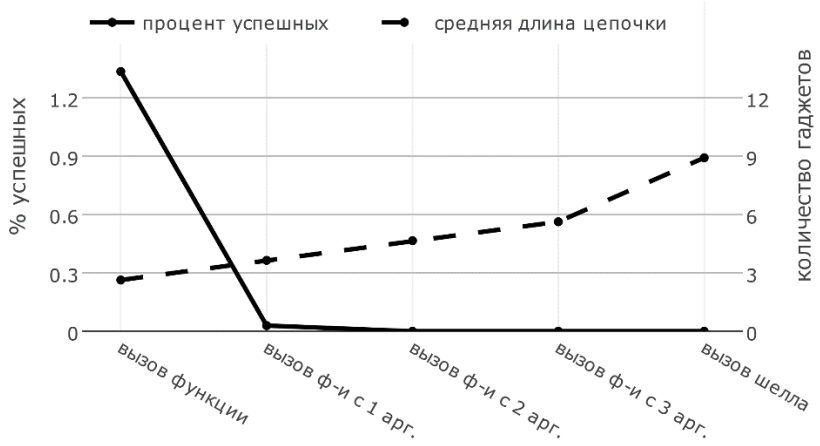

Рис. 5. Усредненная относительная работоспособность оригинальной ROP цепочки для других файлов популяции

Fig. 5. Average success rate of the original ROP chain for other population files

Измерение количества выживших гаджетов производилось путем обращения к базам данных гаджетов, полученных с помощью классификатора гаджетов. Зависимость количества выживших гаджетов от размера популяции представлена на рис. 3. На нем представлена кривая, отражающая среднее арифметическое значение доли выживших гаджетов по всем программам из тестового набора, кроме того у каждой точки отложено среднеквадратическое отклонение от среднего значения. Характер формы представленных кривых

напоминает экспоненциально убывающую последовательность с некоторым константным смещением по оси абсцисс вверх (примерно 2 %). Это остаточное количество выживших гаджетов, наблюдаемое независимо от размера популяции, объясняется следующим замечанием: в исполняемом сегменте кроме кода функций, местоположение которых меняется, находятся также вспомогательные секции: таблица связывания процедур PLT, INIT, FINI и другие. Они остаются неизменными, и гаджеты внутри них всегда являются выжившими.

Оценка работоспособности ROP цепочек. Важно оценить работоспособность ROP цепочек, построенных по оригинальному исполняемому файлу, для других экземпляров популяции. Размер ROP цепочек может варьироваться от нескольких гаджетов до десятков и более. Возьмем несколько примеров цепочек, возрастающего размера и сложности: вызов функции без аргументов, вызов функции с 1 аргументом, вызов функции с 2 аргументами, вызов функции с 3 аргументами и вызов оболочки командной строки.

Перечисленные примеры цепочек составляются по базам данных гаджетов. На рис. 4 приведены результаты построения ROP цепочек для тестового набора исполняемых файлов. Столбцы значений отвечают количеству файлов из тестового набора, для которых построение конкретного примера оказалось возможным. Пунктирная кривая показывает количество гаджетов в ROP цепочке для каждого примера. Затем проверяется работоспособность построенных цепочек для экземпляров в популяции соответствующего исходного файла. Процентное отношение работоспособных файлов к размеру популяции отображено на рис. 5. Из данного графика видно, что процент успешности резко падает с увеличением длины цепочки и для нетривиальных цепочек стремится к нулю. Стоит отметить, что относительно небольшое значение процента успешности для вызова функции без аргументов объясняется тем, что для реализации такой цепочки зачастую достаточно гаджетов из неизменяемых секций (PLT, INIT, FINI).

6. Заключение

В данной работе представлена реализация мелкогранулярной рандомизации адресного пространства программ, с гранулярностью на уровне функций, при их запуске. Функции исполняемых файлов и библиотек размещаются при их загрузке в случайном порядке. Это увеличивает энтропию рандомизации адресного пространства по сравнению с ASLR, что усложняет построение и проведение ROP атак на защищенные таким образом программы. Была экспериментально оценена эффективность противодействия эксплуатации методом ROP с помощью двух метрик: процент выживших гаджетов и оценка работоспособности примеров ROP цепочек. Приводимая в статье реализация показала свою пригодность для применения в масштабах всей системы. Кроме

того, она лишена проблем совместимости: исполняемый файл, собранный с поддержкой рандомизации, может быть загружен стандартным динамическим загрузчиком, а рандомизирующий динамический загрузчик может загружать обычные исполняемые файлы формата ELF без дополнительной секции. В ходе тестирования среднее замедление времени работы тестового набора SPEC2006 составило 1.5 %. Время загрузки программ остается пренебрежимо малым по сравнению с временем их работы.

У разработанной реализации на данный момент имеются незначительные недостатки, которые можно исправить в будущем. Самым существенным недостатком является несоответствие отладочной информации, что затрудняет отладку рандомизированного кода. В дальнейшем необходимо генерировать актуализированную отладочную информацию для исполняемых файлов в режиме отладки. Реализованная рандомизация проводится на уровне функций. Поддержка более мелкой гранулярности позволит увеличить энтропию, а значит, усилить защиту. Также имеет смысл реализовать рандомизацию размещения коротких функций с учетом связей между ними. Близкое размещение функций, часто вызывающих друг друга, может повысить производительность отдельных программ. Кроме того, результаты тестирования эффективности защиты показывают, что в дополнительной защите нуждаются также секции исполняемого файла (PLT, INIT, FINI).

Список литературы

[1]. CVE Details website: vulnerabilities by date. По состояние на 10.04.2017 г. http://www.cvedetails.com/browse-by-date.php

[2]. R. Roemer, E. Bbuchanan, H. Shacham, S. Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., vol. 15, no. 1, 2012, pp. 2:1–2:34.

[3]. A. Sadeghi, S. Niksefat, M. Rostamipour, Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, no. 434, 2017, pp. 1-18

[4]. T. Bletsch, X. Jiang, V. Freeh, W. Liang, Zh. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30-40.

[5]. H. Hu, Sh. Shinde, S. Adrian, Z.L. Chua, P. Saxena, Zh. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. IEEE Symposium on Security and Privacy (SP), 2016, pp. 969-986.

[6]. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp 552-561.

[7]. A. Bittau, A. Belay, A. Mashtizadeh et al. Hacking blind. Proceedings of the 2014 IEEE Symposium on Security and Privacy, 2014, pp. 227–242.

[8]. M. Abadi, M. Budiu, ´U. Erlingsson, J. Ligatti. Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur, vol. 13, no. 1, 2009, pp. 4:1–4:40.

[9]. A.J. Mashtizadeh, A. Bittau, D. Boneh, D. Mazi`eres, Ccfi: Cryptographically enforced control flow integrity. Proceedings of the Sixth ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 941–951.

[10]. N. Christoulakis, G. Christou, E. Athanasopoulos, S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 38–49.

[11]. N. Carlini, A. Barresi, M. Payer et al. Control-flow bending: On the effectiveness of control-flow integrity. Proceedings of the 24th USENIX Conference on Security Symposium, 2015, pp. 161–176.

[12]. K. Lu, S. N¨urnberger, M. Backes, W. Lee. How to make ASLR win the clone wars: Runtime re-randomization. 23nd Annual Network and Distributed System Security Symposium, 2016.

[13]. А.Р. Нурмухаметов, Ш.Ф. Курмангалеев, В.В. Каушан, С.С. Гайсарян. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения. Труды ИСП РАН, том 26, вып. 3, 2014, стр. 113-126. DOI: 10.15514/ISPRAS-2014-26(3)-6

[14]. A. Gupta, S. Kerr, M. Kirkpatrick, E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. Network and System Security, 7 th International Conference, 2013.

[15]. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits. PoPETs, no. 4, 2016, pp. 454–469.

[16]. L. Davi, A. Dmitrienko, S. N¨urnberger, A. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM, 8th ACM Symposium on Information, Computer and Communications Security, 2013.

[17]. M. Backes, S. Nurberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 433–447.

[18]. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE, 2016.

[19]. D. Bigelow, T. Hobson, R. Rudd et al. Timely rerandomization for mitigating memory disclosures, Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 268–279.

[20]. D. Williams-King, G. Gobieski, K. Williams-King et al. Shuffler: Fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 367–382.

[21]. M. Payer. Too much PIE is bad for performance. Technical report.

[22]. J. Coffman, C. Wellons, C.C. Wellons. ROP Gadget Prevalence and Survival under Compiler-based Binary Diversification Schemes. Proceedings of the 2016 ACM Workshop on Software PROtection, 2016, pp. 15-26.

[23]. А.В. Вишняков. Классификация ROP гаджетов. Труды ИСП РАН, том 28, вып. 6, 2016 г., стр. 27-36. DOI: 10.15514/ISPRAS-2016-28(6)-2

[24]. ROPgadget. По состоянию на 16.10.2017 г. https://github.com/JonathanSalwan/ROPgadget

Fine-grained address space layout randomization on program load

[1] *A.R. Nurmukhametov <oleshka@ispras.ru>*
[1] *E.A. Zhabotinskiy <ezhabotinskiy@ispras.ru>*
[1] *Sh.F. Kurmangaleev <kursh@ispras.ru>*
[1,2,3,4] *S.S. Gaissaryan <ssg@ispras.ru>*
[1] *A.V. Vishnyakov <vishnya@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics (HSE) 11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. Program vulnerabilities are a serious security threat. It is important to develop defenses preventing their exploitation, especially with a rapid increase of ROP attacks. State of the art defenses have some drawbacks that can be used by attackers. In this paper we propose fine-grained address space layout randomization on program load that is able to protect from such kind of attacks. During the static linking stage executable and library files are supplemented with information about function boundaries and relocations. A system dynamic linker/loader uses this information to perform functions permutation. The proposed method was implemented for 64-bit programs on CentOS 7 operating system. The implemented method has shown good resistance to ROP attacks based on two metrics: the number of survived gadgets and the exploitability estimation of ROP chain examples. The implementation presented in this article is applicable across the entire operating system and has shown 1.5 % time overhead. The working capacity of proposed approach was demonstrated on real programs. The further research can cover forking randomization and finer granularity than on the function level. It also makes sense to implement the randomization of short functions placement, taking into account the relationships between them. The close arrangement of functions that often call each other can improve the performance of individual programs.

Keywords: address space layout randomization; diversification, ASLR; ROP.

DOI: 10.15514/ISPRAS-2017-29(6)-9

For citation: Nurmukhametov A.R., Zhabotinskiy E.A., Kurmangaleev Sh. F., Gaissaryan S.S., Vishnyakov A.V. Fine-grained address space layout randomization on program load. Trudy ISP RAN/Proc. ISP RAS, 2017, vol. 29, issue 6, pp. 163-182 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-9

References

[1]. CVE Details website: vulnerabilities by date. Accessed 10.04.2017. http://www.cvedetails.com/browse-by-date.php

[2]. R. Roemer, E. Bbuchanan, H. Shacham, S. Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., vol. 15, no. 1, 2012, pp. 2:1–2:34.

[3]. A. Sadeghi, S. Niksefat, M. Rostamipour, Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, no. 434, 2017, pp. 1-18

[4]. T. Bletsch, X. Jiang, V. Freeh, W. Liang, Zh. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30-40.

[5]. H. Hu, Sh. Shinde, S. Adrian, Z.L. Chua, P. Saxena, Zh. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. IEEE Symposium on Security and Privacy (SP), 2016, pp. 969-986.

[6]. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp 552-561.

[7]. A. Bittau, A. Belay, A. Mashtizadeh et al. Hacking blind. Proceedings of the 2014 IEEE Symposium on Security and Privacy, 2014, pp. 227–242.

[8]. M. Abadi, M. Budiu, ´U. Erlingsson, J. Ligatti. Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur, vol. 13, no. 1, 2009, pp. 4:1–4:40.

[9]. A.J. Mashtizadeh, A. Bittau, D. Boneh, D. Mazi`eres, Ccfi: Cryptographically enforced control flow integrity. Proceedings of the Sixth ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 941–951.

[10]. N. Christoulakis, G. Christou, E. Athanasopoulos, S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 38–49.

[11]. N. Carlini, A. Barresi, M. Payer et al. Control-flow bending: On the effectiveness of control-flow integrity. Proceedings of the 24th USENIX Conference on Security Symposium, 2015, pp. 161–176.

[12]. K. Lu, S. N¨urnberger, M. Backes, W. Lee. How to make ASLR win the clone wars: Runtime re-randomization. 23nd Annual Network and Distributed System Security Symposium, 2016.

[13]. A. Nurmukhametov, Sh. Kurmangaleev, V. Kaushan, S. Gaissaryan. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, vol. 41, no. 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052

[14]. A. Gupta, S. Kerr, M. Kirkpatrick, E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. Network and System Security, 7 th International Conference, 2013.

[15]. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits. PoPETs, no. 4, 2016, pp. 454–469.

[16]. L. Davi, A. Dmitrienko, S. N¨urnberger, A. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM, 8th ACM Symposium on Information, Computer and Communications Security, 2013.

[17]. M. Backes, S. Nurberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 433–447.
[18]. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE, 2016.
[19]. D. Bigelow, T. Hobson, R. Rudd et al. Timely rerandomization for mitigating memory disclosures, Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 268–279.
[20]. D. Williams-King, G. Gobieski, K. Williams-King et al. Shuffler: Fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 367–382.
[21]. M. Payer. Too much PIE is bad for performance. Technical report.
[22]. J. Coffman, C. Wellons, C.C. Wellons. ROP Gadget Prevalence and Survival under Compiler-based Binary Diversification Schemes. Proceedings of the 2016 ACM Workshop on Software PROtection, 2016, pp. 15-26.
[23]. Vishnyakov A.V. Classification of ROP gadgets. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 27-36 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-2
[24]. ROPgadget. https://github.com/JonathanSalwan/ROPgadget. Accessed 16.10.2017