Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения*

А.Р.Нурмухаметов
<oleshka@ispras.ru>
Ш.Ф. Курмангалеев
<kursh@ispras.ru>.
В.В. Каушан
<korpse@ispras.ru>
С.С. Гайсарян
<ssg@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25.

Аннотация. Уязвимости программного обеспечения представляют серьезную угрозу безопасности информационной системы. Любое программное обеспечение, написанное на языках C/C++, потенциально содержит в себе значительное количество уязвимостей, используя которые злоумышленник может с помощью специально подготовленных эксплойтов захватить контроль над системой. Для противодействия эксплуатации таких уязвимостей в данной работе предлагается использовать компиляторные преобразования: перестановка местами функций в модуле, добавление локальных переменных на стек функции, перемешивание локальных переменных на стеке. С помощью этих преобразований предлагается генерировать диверсифицированную популяцию исполняемых файлов компилируемого приложения. Такой подход, например, усложняет планирование ROP-атак на всю популяцию. Злоумышленник, получив в свое распоряжение один исполняемый файл, может сделать ROP-эксплойт, работающий только для этой версии приложения. Остальные исполняемые файлы популяции останутся устойчивыми к данной атаке.

Ключевые слова: уязвимость; компиляторные преобразования; ROP-атака; эксплойт.

1. Введение

Программное обеспечение, написанное на языках C/C++, потенциально содержит в себе значительное количество уязвимостей. Уязвимости могут

* Работа поддержана грантом РФФИ 14-01-00462 А

быть как результатом ошибок программирования и недостатков, допущенных при проектировании, так и результатом внесения специальных закладок в код открытых проектов. Оценить количество ошибок непросто из-за разнообразности исходного кода каждого проекта и неясности точного определения того, что является ошибкой, а что не является. Разные источники дают разные количественные данные, например, в работе [1] авторы говорят об одной ошибке на каждую тысячу строк кода. Не столько важно конкретное количество ошибок, сколько их наличие. Даже одна единственная ошибка, затерявшаяся от глаз разработчиков в миллионах строках исходного кода большого проекта, может быть с успехом использована злоумышленником для перехвата управления системы с последующим исполнением вредоносного кода.

Поиск и устранение ошибок, приводящих к уязвимостям, в исходном коде трудоемкая и дорогостоящая задача. Для ее решения существуют такие продукты, как Coverity Insight, Klocwork K9, Svace [2]. Ни один из них не в состоянии найти в большом проекте все уязвимости. В условиях наличия в программном обеспечении уязвимостей актуальной становится задача воспрепятствования их эксплуатации. Для ее решения существует ряд технологий: предотвращение выполнения данных (DEP) [3], рандомизация адресного пространства (ASLR) [4], протектор стека [5], безопасные функции [6], теневой стек [7, 8, 9], исполняемые файлы без инструкций возврата [10, 11]. Суть каждого метода подробнее будет описана в следующем разделе, кроме того, будут проанализированы их сильные и слабые стороны.

Ошибки программирования или специальные закладки, приводящие к уязвимостям программного обеспечения, особо опасны по той причине, что огромное количество компьютеров одновременно обладают идентичным программным обеспечением и могут быть массово атакованы одним эксплойтом. На данный момент сложилась ситуация программного однообразия. Идентичные исполняемые файлы самого популярного программного обеспечения работают на миллионах компьютеров. Это облегчает масштабное эксплуатирование, потому что одна и та же атака с успехом проходит на множество целей.

Задачу противодействия эксплуатации ошибок в программном обеспечении можно разделить на две подзадачи. Первая задача – противодействие современным средствам статического и динамического анализа, чтобы затруднить поиск ошибки и построение эксплойта. Для решения этой задачи требуется использовать запутывающие преобразования, которые также могут быть реализованы в запутывающем компиляторе [12]. Вторая задача – затруднение эксплуатации имеющихся уязвимостей.

В данной работе предлагается путем использования компиляторных преобразований (перестановка функций, добавление локальных переменных на стек, перемешивание переменных на стеке [13, 14]) генерировать

диверсифицированную популяцию исполняемых файлов компилируемого приложения. Злоумышленник, получив одну из версий исполняемого файла, может создать для него эксплойт, который не будет работать для других версий исполняемого файла в силу их различий. Подробно предлагаемый подход описывается в разделе 3.

Настоящая статья состоит из введения, пяти разделов и заключения. В разделе 2 описываются существующие компиляторные метод защиты. В разделе 3 подробно описывается предлагаемый подход. В разделе 4 описываются реализованные компиляторные преобразования и технические подробности, связанные с ними. Влияние на производительность наглядно представлено в разделе 5. В разделе 6 приводятся оценки защищенности предлагаемой методики защиты.

2. Обзор методов защиты

2.1 Предотвращение выполнения данных

Предотвращение выполнения данных (англ. Data Execution Prevention, DEP) — функция безопасности, которая не позволяет выполнять код из областей памяти, помеченных «только для данных». Присутствует во всех современных операционных системах и опирается на аппаратные возможности современных процессоров (NX-бит). Данная технология позволяет предотвратить атаки, которые сохраняют код для последующего исполнения в такой области. Примером таких атак являются атаки переполнения буфера. Классическая атака, использующая переполнение буфера, хранила вредоносный код в самом буфере на стеке, и управление передавалось на начало буфера. Если секция стека помечена флагом «только для данных», то вредоносный код не выполнится, произойдет обработка исключения. Таким образом, данная технология не устраняет саму уязвимость переполнения буфера, а всего лишь запрещает выполнение кода, например, на стеке. Буфер переполнить, возможно, но вредоносный код необходимо положить в другое место, выполнение кода в котором разрешено. Другой способ обхода такого способа защиты – ROP-атака, которая использует код, уже находящийся в области памяти с возможностью выполнения.

2.2 Рандомизация адресного пространства

Рандомизация адресного пространства (англ. Address Space Layout Randomization, ASLR) – технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур, а именно: образа исполняемого файла, подгружаемых библиотек, кучи и стека. Технология значительно усложняет эксплуатацию уязвимостей, приводящих к перехвату потока управления, так как довольно сложно угадать, где будет расположен

стек или куча, в которых можно поместить вредоносный код. Данная технология поддерживается во всех современных операционных системах.

2.3 Протектор стека

Вставка протектора стека (англ. Stack canary) – компиляторный метод защиты от уязвимостей переполнения буфера на стеке. Данный метод позволяет эффективно противостоять атакам переполнения буфера с незначительным падением производительности защищаемой программы. Он реализуется как патч к компилятору, изменяющий пролог и эпилог функции. В прологе функции на стек кладется специальное проверочное значение. Это значение располагается на стеке между локальными переменными и адресом возврата функции, исполняя роль стража адреса возврата. В эпилоге функции происходит проверка проверочного значения на изменение. Если произошло переполнение буфера, то проверочное значение изменится, и проверка приведёт к вызову обработчика исключительной ситуации.

2.4 Безопасные функции

Безопасные функции – это легковесная компиляторная защита от переполнения буфера для некоторых функций работы со строками и памятью. Для этих функций подсчитывается, если это возможно во время компиляции, размер буфера, в который будет производиться запись. Вызовы, для которых размер копируемого буфера известен во время компиляции, проверяются на переполнение непосредственно во время компиляции. Для других вызовов генерируется вызов специальной функции-оболочки, в которой во время исполнения производится проверка на переполнение. В случае если функция не прошла проверку на переполнение во время компиляции, то выбрасывается ошибка компиляции, если функция во время исполнения не пройдет проверку, то вызовется обработчик исключительной ситуации и исполнение прервётся. Использование безопасных функций позволяет контролировать выход за пределы буферов только в том случае, когда компилятор знает их размер. Данная защита не сможет воспрепятствовать переполнению буфера, размер которого неизвестен во время компиляции.

2.5 Теневой стек

В основе этого подхода лежит идея о создании для адресов возврата отдельного стека, который нельзя будет изменить, переполняя какой угодно буфер на стеке с данными. Существуют различные версии реализации теневого стека: аппаратного уровня (SmashGuard) [9], компиляторного уровня (StackShield) [8], уровня исполняемого файла (TRUSS) [7].

На аппаратном уровне семантика инструкций вызова и возврата изменяется таким образом, чтобы при вызове функции копия адреса возврата сохранялась в теневом стеке, а при возврате из функции сравнивалась с текущим адресом

возврата. Если адреса различаются, то происходит прерывание исполнения и запускается обработчик исключительной ситуации. Такой подход не требует никакой модификации программного обеспечения.

Аналогичная логика сохранения копии адреса возврата в теневом стеке используется в защите компиляторного уровня (StackShield). Во время компиляции вставляются инструкции, сохраняющие копию адреса возврата в теневой стек при вызове функции и сравнивающие его с текущим адресом возврата при возврате. Обработчик исключительного события вызывается в том случае, если адреса различаются. Для защиты приложения данным методом нужно иметь доступ к его исходным кодам.

TRUSS использует бинарную инструментацию для проверки адреса возврата до его использования. Каждая инструкция вызова функции инструментируется кодом, который сохраняет копию адреса возврата в теневой стек, а каждая инструкция возврата из функции инструментируется кодом сравнения адресов. В случае их различия вызывается обработчик исключительной ситуации. Преимуществом данного подхода является то, что таким образом можно защищать программы, для которых не предоставлено исходного кода.

2.6 Исполняемые файлы без инструкций возврата

Данный метод защиты направлен против ROP-атак и предлагает исключить абсолютно все инструкции возврата из бинарного кода программы. Машинные коды инструкций возврата могут содержаться не только в самих инструкциях возврата, но и в других инструкциях и их операндах. Для полного их удаления в работе [10] предлагается использовать компиляторные преобразования: перераспределение регистров для удаления машинных кодов возврата из машинных кодов регистровых операндов; оптимизации замены инструкций, содержащих внутри себя или внутри своих нерегистровых операндов машинные коды возврата, на эквивалентные инструкции; методику непрямого возврата для удаления из кода непосредственно инструкций возврата.

Методика непрямого возврата состоит в следующем: при каждом вызове функции вместо адреса возврата на стек записывается некоторый индекс. Этот индекс будет указывать на адрес возврата в некоторой таблице адресов возврата, которая специально выделена в отдельном месте в памяти. При каждом возврате управления из функции адрес возврата находится в таблице адресов возврата по индексу, сохраненному на стеке.

3. Описание предлагаемого подхода

Компилятор является ключевой частью процесса разработки программного обеспечения, в том числе потому, что он транслирует программу, написанную на языке высоко уровня, в низкоуровневую программу, непосредственно исполняемую компьютером. Одним из свойств любого компилятора является

детерминированность процесса трансляции. Это означает, что один и тот же файл исходного кода всегда транслируется в один и тот же исполняемый файл, конечно при условии того, что компиляция производится при одинаковом уровне оптимизации.

Компилятор, оборудованный набором диверсифицирующих преобразований, может генерировать большое количество функционально эквивалентных, но внутренне различных копий компилируемой программы. Поведение программы, специфицированное исходным кодом, у всех копий одинаковое. Разным является поведение в тех случаях, которые не специфицированы напрямую в исходном коде программы, стандартом языка или соглашениями о вызовах и общих интерфейсах. Таким образом, каждая версия программы ведет себя по-разному при попытке эксплуатировать ее неспецифицированное поведение (т.е. уязвимость).

Генерация диверсифицированной популяции исполняемых файлов для одного пакета исходного кода приводит к усложнению и удорожанию разработки эксплойтов. Разработчик, планирующий атаку, должен либо создать эксплойт для каждой копии, либо сделать его достаточно сложным для того, чтобы он мог успешно срабатывать сразу на нескольких копиях. В любом случае ему потребуется больше времени и ресурсов для планирования атаки и разработки эксплойта.

В современном мире большое количество программного обеспечения распространяется через магазины приложений различных платформ (Apple's App Store, Android Marketplace, Windows Market). В такой схеме распространения без труда можно осуществить автоматическую генерацию уникальной копии бинарного кода приложения по запросу пользователя. Злоумышленник, загрузив из магазина приложения одну из копии бинарного пакета, создаст для своей версии приложения эксплойт. При попытке атаки другие копии данного приложения не будут скомпрометированы этим эксплойтом (рис. 1).

Рис. 1. Схема атаки на диверсифицированную популяцию исполняемых файлов.

Для генерации разнообразных программ компилятор должен иметь внутри себя источник недетерминированности. Предлагается использовать линейный

конгруэнтный генератор случайных чисел, в котором последовательность псевдослучайных чисел генерируется рекуррентной формулой:

$$r_n = (A * r_c + C)\% M$$

где A, C, M – некоторые специальным образом подобранные константы, r_c – случайное число на текущей итерации генератора случайных чисел, r_n – случайное число на следующей итерации генератора случайных чисел.

Последовательность генерируемых псевдослучайных чисел однозначно задается по начальной затравке, которая задается флагом компилятора. Таким образом, реализуется воспроизводимая недетерминированность процесса компиляции, которая необходима, например, для отладки программ.

4. Реализованные преобразования

Для диверсификации генерируемых программ были реализованы следующие преобразования: перестановка функции, добавление и перемешивание переменных на стеке. Данные преобразования были реализованы в компиляторе GCC и разрабатываемом в ИСП РАН обфусцирующем компиляторе на базе LLVM [12]. В качестве основной реализации рассматривается GCC, поскольку обладает большей совместимостью (например, собирает ядро Linux). Использование GCC позволяет при необходимости генерировать диверсифицированные образы операционной системы целиком.

4.1 Перестановка функций

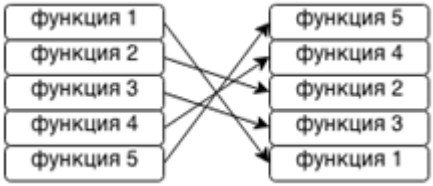

Рис. 2.Перестановка функций местами

Перестановка функций местами меняет структуру памяти процесса. Изменяется относительный порядок функций, меняются адреса точек входа. Данное преобразование особо полезно для противодействия ROP-атакам. При планировании таких атак важно знать местоположения гаджетов. Гаджеты – это небольшие участки машинного кода, оканчивающиеся инструкцией возврата. Правильно передавая управление с одного такого участка кода на другой, можно выполнить вредоносный код. Разработчик эксплойта, подобрав нужную ему последовательность гаджетов для доступной ему копии приложения, столкнется с той проблемой, что адреса гаджетов в другой копии приложения будут другими.

Скажем пару слов о реализации перестановки функций местами (рис. 2). Каждой функции в модуле сопоставляется случайное число, полученное от линейного конгруэнтного генератора случайных чисел. Функции переставляются в соответствии с порядком возрастания, присвоенных им случайных чисел. Количество уникальных перестановок функций местами, следовательно, и количество уникальных копий приложения, равно $n!$.

4.2 Добавление локальных переменных

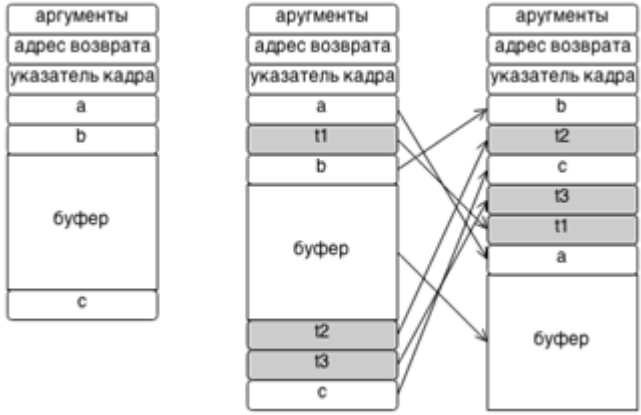

Рис. 3. Добавление локальных переменных и их перемешивание на стеке

Для добавления локальных переменных на стек функции был реализован компиляторный проход над промежуточным представлением GIMPLE. Этот проход добавляет в каждую функцию заданное флагом количество локальных переменных, хранящихся на стеке (рис. 3).

4.3 Перемешивание локальных переменных

Перемешивание переменных на кадре стеке функции (рис. 3) реализовано аналогично перестановке функций местами. Каждой локальной переменной, находящейся на стеке функции, сопоставляется случайное число, полученное от линейного конгруэнтного генератора случайных чисел. Локальные переменные перемешиваются в соответствии с порядком возрастания присвоенных им случайных чисел. Количество уникальных перестановок кадра стека функции равно:

$$(n_f + k)!$$

где n_f – количество локальных переменных на стеке функции, k - количество добавленных локальных переменных. Если рассмотреть количество уникальных перестановок стека глубиной m, то оно задается формулой:

$$\prod_{i=1}^{m}(n_i + k)!$$

где n_i – количество локальных переменных на стеке в функции на глубине вызова i.

5. Влияние на производительность

Тестирование производительности производилось на приложении SQLite. Результаты наглядно представлены в табл. 1. Всего было проведено шесть серий измерений. Первая из них с названием base проводилась при выключенных диверсифицирующих преобразованиях. Серия Peak1 – с включенной перестановкой функций. Серия Peak2 – с включенными перестановками функций и локальных переменных. Серии Peak3, Peak4, Peak5 – с обеими включенными перестановками и количеством добавленных переменных 1, 3, 5 соответственно.

Проведенные диверсифицирующие преобразования отрицательно сказываются на производительности и увеличивают размер кода. Чем больше вставляется локальных переменных на стек, тем больше падает производительность и тем больше становится размер исполняемого файла программы.

Табл. 1. Влияние на производительность реализованных преобразований

серия	Количество добавленных лок. перем.	Перестановка		Увеличение размера, %	Ухудшение производ., %
		лок. перем.	функций		
Peak5	5	+	+	5.16	15.75
Peak4	3	+	+	3.13	7.87
Peak3	1	+	+	1.29	3.94
Peak2	0	+	+	0	1.57
Peak1	0	-	+	0	0.79
Base	0	-	-	0	0

6. Оценка защищенности

Оценка защищенности производилась в режиме полностью контролируемого окружения. Режим полностью контролируемого окружения подразумевает под собой выключенный в системе ASLR и отключенный в исполняемом файле DEP. Для заданного образца защищённого приложения возможна полуавтоматическая генерация эксплойта, приводящего к перехвату управления. Для другой копии, скомпилированной с неизвестным значением затравки для генератора случайных чисел, возможно изменение эксплойта. Для этого нужно подобрать константу, характеризующую масштабы изменений между кадрами стека эксплуатируемых функций. Эта константа

линейно зависит от размера кадра стека незащищённого приложения и количества добавленных локальных переменных. Константу подбирается простым перебором, при этом на каждой итерации перебора требуется запуск целевого приложения. Перебор можно значительно упростить, заполняя свободное пространство эксплойта значением для адреса возврата. В этом случае требуется около десятка попыток. В случае, если на стеке находятся переменные, изменение которых может привести к преждевременному аварийному завершению из-за их использования, то перемешивание локальных переменных может привести как к усилению защиты, так и к её ослаблению.

В случае ROP-атаки возможно построение цепочки ROP-гаджетов по коду копии приложения, имеющегося в наличии. Однако построение ROP-эксплойта для другой копии с неизвестным значением затравки для генератора случайных чисел является довольно сложной задачей, потому что требуется знание адресов ROP-гаджетов для этой копии. Подбор этих адресов потребует неприемлемое количество времени.

Необходимо отметить возможность атаки на затравку для генератора случайных чисел. Предсказуемость этого значения может значительно упростить задачу перебора адресов ROP-гаджетов. Поэтому следует генерировать достаточно случайные значения для затравки генератора, чтобы их было сложно предсказать или подобрать.

Для обеспечения максимального уровня защиты рекомендуется использовать предлагаемый подход совместно с ASLR и DEP. Указанные технологии затрудняют прямую эксплуатацию уязвимостей, запрещая исполнять код на стеке и обеспечивая случайное перераспределение адресов загрузки исполняемых модулей. Тем не менее, сочетание ASLR и DEP зачастую недостаточно, поскольку в системе присутствуют файлы с выключенным ASLR, анализируя которые, можно создать набор ROP-гаджетов (или использовать для этого модуль атакуемого приложения). Применение диверсификации решает данную проблему, поскольку найденные адреса ROP-гаджетов, смещения функций, системных структур оказываются непригодны для построения эксплойта на нескомпроментированных системах.

7. Заключение

В представленной работе предложены и описаны диверсифицирующие преобразования, реализованные в рамках обфусцирующего компилятора на базе GCC. За счет этих преобразований в условиях современной модели распространения программного обеспечения через магазины приложений становится возможным предоставить каждому клиенту свою оригинальную версию бинарного приложения. Внутренняя структура приложений у различных клиентов разная, что повышает уровень защиты, а в случае некоторых типов атак (ROP-атак) противодействует эксплуатации уязвимостей. Реализованные преобразования отрицательно сказываются на

производительности приложений. По результатам тестов было зафиксировано ухудшение производительности на 15% и увеличение размера кода на 5% на тесте SQLite. С применением диверсифицирующих преобразований производилась сборка полного образа системы CentOS, включая ядро Linux и большое количество пакетов приложений. Это говорит о возможности генерации диверсифицированных образов операционной системы целиком.

В дальнейшие планы развития обфусцирующего компилятора входит как увеличение количества диверсифицирующих преобразований, так и оснащение его другими методами защиты от эксплуатации уязвимостей разными типами атак. Теневой стек выглядит одним из самых перспективных методов защиты от перехвата управления.

Список литературы

[1]. Dazhi Z., Detecting Program Vulnerabilities Using Trace-Based Security Testing, Ph. D. Dissertation, University of Texas at Arlington, Arlington, TX, USA, Advisor(s) Donggang L, AAI3474008, 2011.

[2]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института Системного Программирования РАН, том 21, 2011 г, стр. 23-38.

[3]. N. Stojanovski, M. Gusev, D. Gligoroski, S. Knapskog. Bypassing Data Execution Prevention on MicrosoftWindows XP SP2. Proceedings of the The Second International Conference on Availability, Reliability and Security, ARES '07, 2007, p. 1222-1226. doi:10.1109/ARES.2007.54

[4]. H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh. On the Effectiveness of Address-space Randomization. Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, 2004, p. 298-307. doi:10.1145/1030083.1030124

[5]. P. Wagle, C. Cowan. Stackguard: Simple stack smash protection for GCC. Proc. of the GCC Developers Summit, 2003, p. 243-255.

[6]. J. Jelinek. Object size checking to prevent (some) buffer overflows, 2004 https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

[7]. S. Sinnadurai, Q. Zhao, W. Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[8]. StackShield: A “stack smashing” technique protection tool for Linux. (http:/www.angelfire.com/sk/stackshield)

[9]. H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, A. Jalote, B. A. Kuperman. “SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address.” Technical Report TR-ECE 03-13, Purdue University, February 2004.

[10]. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, E. Kirda. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, 2010, p. 49-58. doi:10.1145/1920261.1920269

[11]. J. Li, Z. Wang, X. Jiang, M. Grace, S. Bahram. Defeating Return-oriented Rootkits with "Return-Less" Kernels. Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, 2010, p. 195-208. doi:10.1145/1755913.1755934

[12]. В.Иванников, Ш. Курмангалеев, А. Белеванцев, А. Нурмухаметов, В. Савченко, Р. Матевосян, А. Аветисян. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. руды Института Системного Программирования РАН, том 26, 2014 г, выпуск 1 стр. 327-342. doi: 10.15514/ISPRAS-2014-26(1)-12

[13]. M Stewart. Algorithmic Diversity for Software Security. (http://arxiv.org/abs/1312.3891)

[14]. M. Franz. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In Proceedings of the 2010 Workshop on New Security Paradigms, NSPW ’10, p. 7–16. doi:10.1145/1900546.1900550

Compiler protection techniques against software vulnerabilities exploitation*

A. Nurmukhametov
<oleshka@ispras.ru>
Sh. Kurmangaleev
<kursh@ispras.ru>
V. Kaushan
<korpse@ispras.ru>
S. Gaissaryan
<ssg@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. Software vulnerabilities are critical for security. All C/C++ programs contain significant amount of vulnerabilities. Some of them can be successfully exploitable by attacker to gain control of the execution flow. In this article we propose several compiler protection techniques against vulnerability exploitation: function reordering, insertion of additional dummy variables into stack, local variables permutation on the stack. These transformations were implemented in GCC. It successfully diversifies whole operational system including Linux kernel. We suggest to generate diversified population of binary application files with these transformations. Diversified applications can be easily distributed via the application stores. Every client downloads the unique copy of application. The proposed method complicates and increases the cost of ROP-attacks. After downloading of the binary copy attacker can create ROP-exploit for this copy but it would not be exploitable for another application copy. The diversified transformations decrease application performance about 15% and increase code size about 5%.

Keywords: vulnerability; compiler transformations; ROP-attacks; exploit.

References

[1]. Dazhi Z., Detecting Program Vulnerabilities Using Trace-Based Security Testing, Ph. D. Dissertation, University of Texas at Arlington, Arlington, TX, USA, Advisor(s) Donggang L, AAI3474008, 2011.

[2]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode program [The usage of static analysis for searching vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, 2011. p. 23-38.

[3]. N. Stojanovski, M. Gusev, D. Gligoroski, S. Knapskog. Bypassing Data Execution Prevention on MicrosoftWindows XP SP2. Proceedings of the The Second International

* The paper is supported by RFBR grant 14-01-00462 A

Conference on Availability, Reliability and Security, ARES '07, 2007, p. 1222-1226. doi:10.1109/ARES.2007.54

[4]. H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh. On the Effectiveness of Address-space Randomization. Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, 2004, p. 298-307. doi:10.1145/1030083.1030124

[5]. P. Wagle, C. Cowan. Stackguard: Simple stack smash protection for GCC. Proc. of the GCC Developers Summit, 2003, p. 243-255.

[6]. J. Jelinek. Object size checking to prevent (some) buffer overflows, 2004 https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

[7]. S. Sinnadurai, Q. Zhao, W. Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[8]. StackShield: A "stack smashing" technique protection tool for Linux. (http:/www.angelfire.com/sk/stackshield)

[9]. H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, A. Jalote, B. A. Kuperman. "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address." Technical Report TR-ECE 03-13, Purdue University, February 2004.

[10]. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, E. Kirda. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, 2010, p. 49-58. doi:10.1145/1920261.1920269

[11]. J. Li, Z. Wang, X. Jiang, M. Grace, S. Bahram. Defeating Return-oriented Rootkits with "Return-Less" Kernels. Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, 2010, p. 195-208. doi:10.1145/1755913.1755934

[12]. V.Ivannikov, SH. Kurmangaleev, A. Belevantsev, A. Nurmukhametov, V. Savchenko, R. Matevosyan, A. Avetisyan. Realizatsiya zaputyvayushhikh preobrazovanij v kompilyatornoj infrastrukture LLVM [Implementing Obfuscating Transformations in the LLVM Compiler Infrastructure]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, iss.1, 2014. p. 327-342. doi: 10.15514/ISPRAS-2014-26(1)-12

[13]. M Stewart. Algorithmic Diversity for Software Security. (http://arxiv.org/abs/1312.3891)

[14]. M. Franz. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In Proceedings of the 2010 Workshop on New Security Paradigms, NSPW '10, p. 7–16. doi:10.1145/1900546.1900550