УДК 006.72 **DOI** 10.15622/ia.21.2.3

И.А. Лубкин, В.В. Золотарев

КОМПЛЕКСНАЯ СИСТЕМА ЗАЩИТЫ ОТ УЯЗВИМОСТЕЙ, ОСНОВАННЫХ НА ВОЗВРАТНО-ОРИЕНТИРОВАННОМ ПРОГРАММИРОВАНИИ

Лубкин И.А., Золотарев В.В. **Комплексная система защиты от уязвимостей, основанных на возвратно-ориентированном программировании.**

Аннотация. Затруднительно или невозможно создать программное обеспечение, не содержащее ошибок. Ошибки могут приводить к тому, что переданные в программу данные вызывают нештатный порядок выполнения ее машинного кода. Разбиение на подпрограммы приводит к тому, что инструкции возврата из подпрограмм могут использоваться для проведения атаки. Существующие средства защиты в основном требуют наличия исходных текстов для предотвращения таких атак. Предлагаемая методика защиты направлена на комплексное решение проблемы. Во-первых, затрудняется получение атакующим контроля над исполнением программы, а во-вторых, снижается количество участков программ, которые могут быть использованы в ходе атаки. Для затруднения получения контроля над исполнением применяется вставка защитного кода в начало и конец подпрограмм. При вызове защищенной подпрограммы производится защита адреса возврата, а при завершении – восстановление – при условии отсутствия повреждения его атакующим. Для снижения количества пригодных для атак участков применяются синонимичные замены инструкций, содержащие опасные значения. Предложенные меры не изменяют алгоритм работы исходного приложения. Для проверки описанных решений была выполнена программная реализация и проведено ее тестирование с использованием синтетических тестов, тестов производительности и реальных программ. Тестирование показало правильность принятых решений, что обеспечивает устранение пригодных для атак участков и невозможность использования штатных инструкций возврата для проведения атак. Тестирование производительности показало 14 % падения скорости работы, что находится на уровне ближайших аналогов. Сравнение с аналогами показало, что количество реализуемых сценариев атаки для предложенного решения меньше, а применимость выше.

Ключевые слова: уязвимость, удаленное исполнение кода, защита кода, RoP, вставка кода.

1. Введение. История развития информационных технологий и их современное состояние позволяют сделать вывод о том, что разработка программного обеспечения (ПО) в общем случае не может исключать появление в нем ошибок. Отражением такого состояния дел являются работы по оценке нормального количества ошибок на определенное количество строк кода [1].

Ошибки ПО являются основой для реализации угроз. Объективно не может быть создана методика разработки ПО, не содержащего ошибок. Устранение ошибок, как разработчиками, так и специалистами по информационной безопасности также объективно не может быть выполнено полностью. Поэтому должны применяться меры, препят-

ствующие использованию ошибок в качестве уязвимостей. Для этого выполняются мероприятия по устранению условий, специфичных для определенных классов уязвимостей. В данной работе рассматривается класс *RoP*-уязвимостей. Уязвимости такого типа основаны на передаче в программу вызывающих сбой данных. Некорректная их обработка нарушает граф потока управления (далее – ГПУ) и приводит к реализации уязвимостей удаленного исполнения кода.

Предлагаемый подход к защите ставит своей целью устранение фрагментов программ – так называемых «гаджетов» (их виды и критерии описаны в [2]), используемых для реализации уязвимостей *RoP*-класса. Защита актуальна для программ, с которыми может взаимодействовать атакующий (например, сетевые сервисы, доступные через Интернет). Устранение реализуется за счет удаления гаджетов или их перевода в непригодное для атакующего состояние. При этом логика работы программы не нарушается.

Предлагаемый метод защиты не требует наличия исходных текстов программ. Это обеспечивает расширение области применимости по сравнению с теми существующими методами, которые могут быть использованы исключительно на этапе компиляции.

Структурно данная работа изложена следующим образом: во втором разделе приведён обзор наиболее близких аналогов предлагаемого подхода по выявлению и устранению уязвимостей. Из их анализа следует актуальность данной работы. В третьем разделе представлены основные сведения о рассматриваемом аппаратном и программном окружении защищаемых программ. Его необходимо учитывать для реализуемости системы защиты. Тут же описываются ограничения на применимость предлагаемого метода. Далее формулируются модель атаки и следующие из нее критерии определения гаджетов. В четвертом разделе предлагается алгоритм предотвращения использования штатных инструкций в составе гаджетов. В пятом разделе перечисляются виды структурных элементов ПО, которые могут быть интерпретированы атакующим как гаджеты. В подразделах приводятся конкретные меры защиты, и обосновывается отсутствие их влияния на целевой алгоритм ПО. В шестом разделе описывается программная реализация метода, и в седьмом – способы ее проверки на корректность и эффективность. В заключительном разделе будут сделаны выводы и сформулированы дальнейшие перспективы работы.

2. Обзор существующих подходов. Проанализируем существующие решения по защите от *RoP*-атак для формирования требований к разрабатываемой системе защиты. Они направлены на устранение условий проведения атаки. Согласно [2], таковыми являются:

– перехват управления атакующим в результате повреждения адреса возврата в стеке;

– передача управления на цепочку гаджетов, адрес размещения которых известен атакующему;

– наличие дополнительной уязвимости, которая позволяет читать фрагменты адресного пространства атакуемой программы (примитив чтения), в частности для определения адресов гаджетов.

В защищаемой системе могут присутствовать как программы с доступным эксплуатанту исходным кодом, так и без такового. Вследствие этого требование наличия исходного кода и проведения перекомпиляции должно быть включено в критерии сравнения, так как оно влияет на применимость средства защиты.

В таблице 1 приведены ближайшие обнаруженные аналоги, направленные на противодействие *RoP*-уязвимостям.

Таблица 1. Анализ средств противодействия *RoP*-уязвимостям

Наименование решения	Необходимость перекомпиляции	Защита от перехвата управления	Снижение числа гаджетов	Уязвимо при раскрытии информации	Примечание
Control-flow Enforcement Technology (CET) [3, 4]	Да	За счет теневого стека	Не требуется за счет контроля над ГПУ	Нет	Поддерживается с 2020 г. [5], но не всеми процессорами не всех изготовителей
PaXgrsecurity RAP [6] и схожий по принципу *StackGuard*	Да	За счет контроля целостности адреса возврата	Нет	Да [7]	—
ИСП Обфускатор [8, 9]	Да	За счет косвенной адресации	Нет	Да [10]	—

Продолжение Таблицы 1

Наименование решения	Необходимость перекомпиляции	Защита от перехвата управления	Снижение числа гаджетов	Уязвимо при раскрытии информации	Примечание
Selfrando (и схожие решения [11, 12])	Да	Нет	За счет случайного размещения при запуске	Да [13]	—
Shuffler [14]	Нет	За счет косвенной адресации	За счет периодического случайного перемещения	Да, до момента следующего перемещения	Уменьшение периода негативно влияет на производительность
RuntimeASLR [15]	Нет	Нет	Да, в дочерних процессах	Да	—
G-free [16] и работа [17]	Да	Нет	Да, кроме технологических участков	Нет	—
Scylla [18]	Да	Нет	Да, за счет шифрования участков кода и случайного перемещения	Да, в рамках расшифрованного участка	—

Из анализа приведенной подборки существующих решений противодействия *RoP*-атакам можно заключить, что первым проработанным направлением являются аппаратные системы защиты, но они требуют новейшего или специфичного оборудования и представлены не на всех платформах. Они неприменимы для эксплуатируемых систем, аппаратное обеспечение которых не подвергается модернизации (в том числе по экономическим причинам). Вторым направлением являются встраиваемые на этапе компиляции системы защиты, но они не могут быть применены при отсутствии исходных текстов ПО. Рассмотренные решения, которые применимы в описанной ситуации

(*Shuffler* и *RuntimeASLR*), уязвимы к получению атакующим информации о содержимом памяти, так как не ставят своей целью устранение гаджетов, а лишь пытаются сделать место их размещения неизвестным. Методика защиты, решающая перечисленные проблемы, не должна требовать наличия исходных текстов и должна обеспечивать отсутствие в памяти программы пригодных для использования гаджетов. Это послужило основанием для создания новой предлагаемой методики защиты. Она должна быть применима для архитектуры процессоров *AMD*64 (и ее аналога *Intel* 64), так как именно для них актуальны *RoP*-уязвимости, что подтверждается рассмотренными источниками.

3. Постановка задачи. Введем ограничения на область применимости предлагаемой методики. Введение ограничений осмысленно, так как абсолютные защиты нереализуемы. Ограничимся атаками типа *RoP*. Не рассматриваются по причине отсутствия или крайней редкости в рамках штатной эксплуатации программных средств:

– исполнимые участки, доступные для записи – и наоборот, у доступных для записи участков отсутствует атрибут исполнимости;

– программы, содержащие вручную написанные ассемблерные вставки (вследствие возможной их нестандартной структуры, что делает затраты на реализацию системы защиты неприемлемыми);

– ситуации отладки, когда возможна модификация контекста исполнения и данных в адресном пространстве программ внешней программой (в основном отладчиком);

– приложения с содержащимися в них программными закладками (в силу возможности внедрения такого вредоносного кода, который позволит обойти произвольную систему защиты из-за отсутствия механизмов разграничения доступа к регионам памяти в рамках одного адресного пространства).

Рассматриваются прикладные программы, написанные на компилируемых языках программирования (например, Си и Си++), так как они удовлетворяют перечисленным условиям. В общем виде они состоят из набора подпрограмм, которые в ходе выполнения вызывают другие подпрограммы. Структура подпрограмм и порядок обмена данными между ними (передача аргументов и получение результата выполнения) определяются бинарным интерфейсом приложений (далее – БИП). Типовая структура подпрограмм включает пролог, основную часть и эпилог. В прологе обычно сохраняются регистры, которые не должны быть повреждены при вызове (в типовом случае – *RBP*), устанавливается база фрейма стека для данной подпрограммы (в регистр *RBP* сохраняется значение регистра *RSP*) и выделяется память под локальные переменные (любые из перечисленных операций могут отсут-

ствовать). В основной части выполняется целевое содержимое подпрограммы (при этом баланс работы со стеком должен быть нулевым – сколько было помещено, столько должно быть извлечено). В эпилоге выполняются в противоположном порядке обратные операции, содержащиеся в прологе. В завершение вызывается инструкция возврата из подпрограммы.

При рассмотрении содержимого стека программы в произвольный момент работы он будет состоять из последовательно записанных фреймов. Пронумеруем их, начиная с нулевого, который соответствует подпрограмме, с которой началось исполнение. Введем обозначения для подпрограмм P_i и адресов возврата R_i, где i – некое неотрицательное число. Нижний индекс показывает положение фрейма стека относительно фрейма начальной подпрограммы. Например, если у текущей рассматриваемой подпрограммы фрейм i, то у вызывающей подпрограммы будет фрейм i-1, а у вызываемой – i+1. Описываемые структурные элементы приведены на рисунке 1.

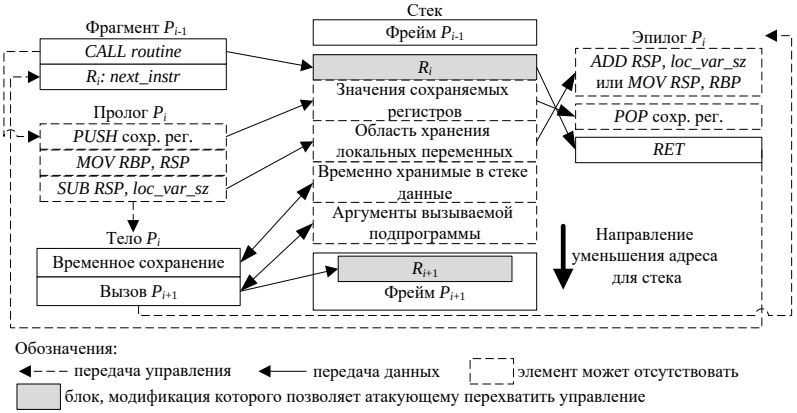

Рис. 1. Типовая структура подпрограмм

Для упрощения описания введем понятие опасной инструкции (далее – ОИ). Под ОИ будем понимать инструкции передачи управления по аргументу, который считывается из ОЗУ или регистров процессора. Примерами ОИ являются инструкции возврата *RET* или перехода по адресу, содержащегося в регистре *JMP* <регистр>. Инструкции перехода по фиксированному смещению не относятся к ОИ. Дополнительно введем понятие опасного значения (далее – ОЗ): это последовательность байтов в составе инструкции, которая не является ОИ, при

передаче управления на которые они будут интерпретированы процессором как ОИ.

На основании результатов анализа данных о структуре подпрограмм [3] и классификации гаджетов из [2] выделим критерии применимости участков программ в качестве гаджетов, которые могут использоваться злоумышленником и подлежат защите:

– последняя инструкция гаджета является ОИ или ОЗ. Она необходима для передачи управления на следующий гаджет. Адрес передачи управления должен быть предсказуем для атакующего;

– перед последней инструкцией гаджет должен содержать инструкции, модифицирующие содержимое регистров или памяти, эффект выполнения которых необходим для достижения целей атакующего;

– гаджет не должен содержать данных, которые интерпретируются процессором как некорректные инструкции или повреждают данные, необходимые атакующему.

Невыполнение любого из перечисленных критериев делает гаджет непригодным для использования в рамках атаки. Сформулируем модель атаки для определения целей атакующего. Атака осуществляется путем обмена данными с уязвимым приложением локально или удаленно. Модель включает в себя следующие действия атакующего:

– запись при выполнении тела P_i в стек массива данных такого размера, чтобы как минимум произошло повреждение R_i. В результате этого он будет заменен на необходимый атакующему адрес R_{gadget}, который передаст управление на первый гаджет в цепочке. Повреждение становится возможным при условии некорректной обработки поступающей информации атакуемым приложением. Дополнительно в стек по положительному смещению от места хранения адреса возврата записываются данные для дальнейших этапов эксплуатации уязвимости (адреса следующих гаджетов и их аргументы). Примем, что у атакующего в ходе выполнения P_i нет возможности модификации R_{i-1} … R_0 без повреждения R_i вследствие последовательной записи в стек. Из невыполнения данного условия следует наличие у атакующего примитива записи по произвольному адресу, что не позволяет противодействовать такой атаке из-за возможности произвольным образом задавать ГПУ (например, внося коррекции в таблицы виртуальных функций);

– по завершении выполнения эпилога уязвимой P_i выполняется инструкция *RET*, которая передает управление по адресу R_{gadget}, записанному атакующим вместо штатного адреса возврата R_i. Гаджеты должны заканчиваться инструкцией *RET*, которая считывает адрес

следующего гаджета и передает на него управление (саму цепочку формирует атакующий перед атакой);

– выполнение цепочки гаджетов, которое имеет конечной целью вызов функций ОС, которые необходимы атакующему для нарушения безопасности. Для этого атакующему необходимо записать аргументы функции в регистры, используемые для передачи аргументов (определяются БИП), и передать управление на её точку входа из последнего гаджета;

– делается предположение об отсутствии у атакующего одновременно и возможности контролировать значения произвольных регистров, и передавать управление по произвольному адресу. Невыполнение данного предположения делает *RoP*-атаку ненужной и позволяет, например, сделать область стека исполнимой и передать на нее управление посредством классической атаки на переполнение буфера;

– делается предположение о возможности у атакующего сформировать в стеке массив данных, содержащий адреса необходимых гаджетов, и передать управление на первый гаджет не только через повреждение R_i. Например, путем записи в единичный регистр, значение которого контролирует атакующий, адреса гаджета и последующего выполнения инструкции вида *JMP* <регистр> (*JoP*-атака). Другим примером является повреждение данных в куче (например, адреса таблицы виртуальных функций с последующим вызовом метода, для которого была выполнена подмена адреса);

– делается предположение о наличии в программе и доступности для атакующего уязвимости, реализующей примитив чтения. Подавая на вход некорректные данные, атакующий сможет получить фрагменты адресного пространства атакуемой программы. Предположим, что за раз считывается фрагмент, соизмеримый с размером фрейма подпрограммы. Примитив чтения позволяет до эксплуатации *RoP*-уязвимости сформировать дамп содержимого исполнимой памяти программы и определить адреса содержащихся в ней гаджетов;

– принимается (с учетом редкости уязвимостей ПО), что в рамках одного вызова подпрограммы не могут быть последовательно реализованы и уязвимость примитива чтения, и уязвимость повреждения R_i. Ситуация, при которой в рамках одной подпрограммы атакующий либо читает участок памяти ограниченного размера, либо в рамках нее же повреждает стек, принимается как актуальная.

Использование такой модели позволяют априорно сделать предлагаемые меры устойчивыми к большему числу сценариев атак. Безопасность решения не основывается на неизвестности атакующему сведений о программе или невозможности влиять на управление ина-

че, кроме как путем повреждения стека. Построенная на таких исходных посылках защита не станет уязвима из-за наличия у атакующего любого примитива чтения.

Можно сформулировать требования к предлагаемой системе защиты от *RoP*-уязвимостей:

– исходные коды защищаемой программы не требуются;

– преобразования защищаемого приложения не должны влиять на логику его работы (т.е. на результаты работы приложения);

– должна быть обеспечена невозможность использования штатных инструкций возврата из подпрограмм в качестве гаджетов. То есть после применения мер защиты при замене R_i на R_{gadget} в рамках выполнения тела P_i недопустим переход по адресу R_{gadget} при выполнении штатной инструкции возврата;

– должны быть устранены ОЗ. То есть инструкция, в составе данных которой присутствует ОЗ, должна быть заменена на одну или более инструкций, байтовое представление которых не содержит ОЗ.

Для реализации предлагаемой методики нужны средства резервирования в программе участков для вставки кода системы защиты без нарушения ее алгоритма и без использования исходных текстов. Пример такого средства и принцип его работы описан в [19, 20]. Предлагаемые меры защиты приведены далее.

4. Защита штатных инструкций возврата из подпрограмм. Из рассмотрения существующих подходов к решению данной проблемы можно выделить два направления: замена ОИ с косвенным указанием R_i и контроль целостности R_i. Недостатком первого подхода является утрата бинарной совместимости кода со стандартными библиотеками ОС, невозможность использования в качестве функций обратного вызова, невозможность реализации исключений. Второй подход не имеет указанных недостатков, но уязвим к специфичным для метода атакам. Для контроля целостности адреса возврата используется некое значение (условно ключ системы защиты), на основе которого и R_i в прологе P_i вычисляется значение, которое будет использовано для контроля целостности в эпилоге. Если атакующий компрометирует ключ (общий для всех подпрограмм), то любая из подпрограмм становится уязвима. Для компрометации достаточно прочитать любой фрейм стека посредством примитива чтения. Для устранения данного недостатка предлагается метод защиты, который является развитием второго подхода к защите.

Если для защиты ОИ из состава P_i при каждом запуске формируется непредсказуемое или трудно предсказуемое (для простоты далее называется случайным) значение K_i, то для возможности подмены

R_i необходимо получение содержимого фрейма стека, содержащего это значение. Техническим ограничением является скорость генерации K_i. Сравнение возможных источников приведено в таблице 2.

Таблица 2. Сравнение возможных источников K_i

Источник K_i	Характеристика источника	Количество случайных бит	Возможность компрометации	Примечание
Инструкция *RDRAND*	Декларируется как криптографический стойкий ГПСЧ	32	Аппаратные закладки или уязвимости	Доступна в процессорах *Intel* с 2013 года, а производства *AMD* – с 2015
Инструкция *RDTSC*	Недетерминированность из-за прерываний, промахов кэша, затрат на синхронизацию, возврата значений для разных ядер и т.п.	Минимум 4 (по результатам экспериментов авторов)	Нет из-за невозможности построения физической модели процессора	Доступна на всех процессорах с архитектурой *AMD*64
ГПСЧ, предоставляемые ОС или в составе эпилогов	Криптографически стойкий	Не менее 32	Нет из-за изоляции процессов в ОС	Имеет неприемлемую скорость генерации
ГПСЧ, функционирующий на выделенном ядре	Криптографически стойкий	Не менее 32	Локальные атаки	Ресурсы одного ядра становятся недоступны
Прочие источники случайности	Отсутствуют гарантии непредсказуемости в общем случае	Не определено	Нет оценки	Например, «мусор» в стеке

Помимо качества случайного значения с точки зрения эксплуатации, важна скорость работы. По результатам сравнения были оценены как перспективные варианты *RDRAND*, *RDTSC* и ГПСЧ на отдельном ядре. Дальнейшее их сравнение проводилось по результатам апробации. Для снижения накладных расходов защитное преобразова-

ние R_i также должно занимать минимальные ресурсы. При этом криптографическая стойкость такого преобразования не требуется. Вследствие этого может быть выбрана любая обратимая операция. В рамках рассмотренных существующих решений в подавляющем большинстве случаев используется *XOR*. Данный вариант был также выбран для предлагаемой системы защиты.

Согласно модели атаки, повреждение R_i может быть осуществлено в ходе выполнения тела P_i. Вследствие этого защитное преобразование должно быть выполнено до тела, в прологе, а контроль целостности адреса – после тела, в эпилоге. Вследствие того, что при выполнении атаки посредством повреждения R_i штатное продолжение функционирования программы будет невозможно в любом случае, то допустимо в защищенной программе аварийно завершать ее функционирование при выявлении попытки получения злоумышленником контроля над ГПУ. Вследствие этого, если при повреждении R_i управление будет передано кодом системы защиты на непредсказуемый адрес, то это будет приемлемым решением. Таким образом, для контроля целостности в эпилоге R_i заменяется на R_i *XOR* K_i, а в прологе на основании того же ключа за счет повторной операции преобразования восстанавливается корректный адрес возврата. Если атакующий заменит хранимое в стеке значение R_i *XOR* K_i на R_{gadget}, то в результате работы кода системы защиты управление будет передано на адрес R_{gadget} *XOR* K_i, что не позволит провести успешную *RoP*-атаку, так как значение K_i неизвестно атакующему.

Для корректной работы такой системы защиты сгенерированное в прологе значение K_i должно быть сохранено до эпилога для восстановления адреса. Место хранения должно выбираться так, чтобы оно не было доступно атакующему. Для этого могут быть использованы регистры общего назначения (далее – РОН) или регистры расширения. При этом программа должна быть проанализирована на наличие гаджетов и штатных инструкций, позволяющих атакующему влиять на место хранения. Подверженные влиянию регистры не должны использоваться для защиты. Отметим, что хранение в стеке неприемлемо, так как при возможности переполнения буфера атакующий может заменить хранящееся значение K_i на нулевое, а R_i на R_{gadget}. Тогда управление будет передано на нужный ему адрес.

Проблемой описанного подхода является то, что вследствие ограниченности числа регистров процессора сгенерированное в прологе P_{i-1} значение K_{i-1} должно где-то храниться на время вызова P_i (и для этого не могут использоваться те же регистры). Решением является сохранение K_{i-1} в стеке в рамках пролога P_i и возвращение в регистр-

хранилище в рамках эпилога. При этом хранение защищаемого значения в непреобразованном виде создает изъян безопасности (описан ранее), вследствие чего стоит хранить значение в виде «K_{i-1} *XOR* K_i» (решение проблемы первого в цепочке значения K_0 приведено далее). Для реализации данного варианта в ходе встраивания системы защиты должен проводиться анализ неиспользуемых регистров. При вызове внешних подпрограмм текущей подпрограммой ее K_i либо должно быть сохранено в стеке, либо для вызываемой внешней подпрограммы должно быть известно, что она не модифицирует хранилище K_i, либо для вызываемой библиотеки должна быть применена описываемая система защиты.

Анализ защищенности приведенного выше решения показал, что при наличии у атакующего примитива чтения в P_j может быть прочитан фрейм стека P_i, где $j > i$. Таким образом, атакующий получит значение R_i *XOR* K_i. Исходя из предсказуемости выполнения трасс программ, атакующий может определить значение R_i и вследствие этого получить значение K_i. В том же фрейме хранится и K_i в открытом виде. Если в P_i содержится уязвимость переполнения буфера, то атакующий, заменив защищенный адрес возврата на R_{gadget} *XOR* K_i, перехватит управление над выполнением программы.

Для защиты от реализации данного сценария необходимо выполнить в рамках пролога P_i дополнительное защитное преобразование над местом хранения защищенного R_{i-1} с использованием K_i. Также K_{i-1} в рамках пролога должен помещаться в стек в защищенном виде как K_{i-1} *XOR* K_i. Запишем в формальном виде защитные преобразования в прологе P_i, выполняемые в i и i-1 фрейме стека:

$$R_i \rightarrow R_i \; XOR \; K_i$$
$$R_{i-1} \; XOR \; K_{i-1} \rightarrow R_{i-1} \; XOR \; K_{i-1} \; XOR \; K_i.$$
$$K_{i-1} \rightarrow K_{i-1} \; XOR \; K_i$$

При выполнении пролога P_{i+1} в фрейме i будет выполнена модификация:

$$R_i \rightarrow R_i \; XOR \; K_i \; XOR \; K_{i+1}.$$

В эпилогах будут выполнены обратные преобразования. В частности, обратная операция размещается непосредственно перед ОИ, что делает невозможным использование штатных инструкций подпрограмм в качестве гаджетов. Попытка повреждения памяти, хранящей R_i

в защищенном виде, приведет к переходу на непредсказуемый для атакующего адрес и аварийное завершение программы.

При отсутствии повреждения защищенных адресов возвратов аргумент инструкции *RET* будет аналогичен оригинальной программе, и управление будет передано вызывающей подпрограмме. Таким образом, предлагаемая мера защиты не искажает ГПУ.

При выполнении P_0 предшествующего фрейма не существует. Для решения данной проблемы в рамках кода подготовки программы к запуску генерируется значение K_{init}, которое сохраняется в стеке и указывается как адрес возврата вызывающей подпрограммы для P_0. В ходе выполнения P_0 хранимое значение K_{init} будет заменено на K_{init} XOR K_0. Таким образом, значение K_{init} не хранится в незащищенном виде и может быть восстановлено атакующим только при знании K_0. Допустимо принять, что ГПУ стабилен и определен для атакующего, поэтому R_i известно атакующему (так как вызывающая подпрограмма определяется ГПУ), а, следовательно, он может вычислить «K_i *XOR* K_{i-1}» для любого i.

Отметим, что переходы по адресам, содержащимся в регистрах, также не могут быть устранены или заменены. Используемые не в качестве начального элемента *JoP*-атаки, они не являются самодостаточными без предшествующего помещения атакующим значений в регистры и передачи управления на соответствующие инструкции в рамках *RoP*-атаки. Следовательно, повышение устойчивости программы к *RoP*-атакам автоматически повышает устойчивость к *JoP*-атакам. При этом первичное получение контроля на ГПУ в рамках *JoP*-атаки возможно согласно модели атаки.

Рассмотрим возможные атаки, направленные на преодоление предложенных мер по защите возвратных инструкций.

1. Повреждение стека с перезаписью адреса. Для успешной реализации требует чтения значения R_i *XOR* K_i и переполнения стека в рамках одной подпрограммы с записью в стек значения R_{gadget} *XOR* K_i, где R_{gadget} – адрес первого гаджета в *RoP*-цепочке. При этом между утечкой значения защищенного адреса и применением эксплойта должно выполниться формирование полезной нагрузки эксплойта (то есть массива данных, который будет некорректно обработан и приведет к повреждению адреса возврата). Согласно модели атак, данный вариант принят как неактуальный.

2. Угадывание значения K_i. При использовании *RDRAND* или *RDTSC* при неизвестном предшествующем значении вероятность 2^{-32} (вследствие использования 32-битного случайного значения). При из-

вестном значении *RDTSC* для вызывающей подпрограммы можно пессимистично оценить как 2^{-4}.

3. Повреждение стека вызывающей подпрограммы с заменой R_i *XOR* K_i *XOR* K_{i+1} на R_{gadget} *XOR* K_i *XOR* K_{i+1}. Данный вид атаки эффективен, но требует, согласно модели атаки, знания K_i для корректного возврата из текущей подпрограммы для последующего перехвата управления в вызывающей подпрограмме.

4. Атака с чтением начальной области стека. Атакующий использует примитив чтения в *n*-й подпрограмме для получения значений K_{init} *XOR* K_0 и пар значений: $<K_i$ *XOR* K_{i-1}, K_i *XOR* $K_{i+1}>$, где $0<i<m$, $m<n$. Полученная последовательность не позволяет сформировать эксплойт для *i*-й подпрограммы, так как атакующему требуется значение K_i, которое невычислимо из полученных пар при использовании качественного источника случайных чисел.

5. Атака с чтением фрагментов полного стека. Использование примитива чтения в *n*-й подпрограмме для получения значений K_{init} *XOR* K_0 и пар значений: $<K_i$ *XOR* K_{i-1}, K_i *XOR* $K_{i+1}>$, где $0<i<n$, и значения K_n атака будет возможна, если все перечисленные значения могут быть получены без повторного вызова подпрограмм с *i*-й по *n*-ю, так как в противном случае будет выполнено пересоздание ключей на указанном участке. При пересоздании ключей в распоряжении атакующего будет фактическая последовательность: K_{init} *XOR* K_0 и пары значений $<K_i$ *XOR* K_{i-1}, K_i *XOR* $K_{i+1}>$ для $0<i<k$, пары значений $<K'_i$ *XOR* K'_{j-1}, K'_j *XOR* $K'_{j+1}>$ для $k<j<n$. Индекс *k* соответствует границе прочитанных фрагментов стека. Чтение стека производится атакующими фрагментами согласно модели атаки. Полученная последовательность не позволяет сформировать эксплойт для *i*-й подпрограммы, так как требует значения K_i, для вычисления которого необходимы значения K_{i+1}, … K_n.

6. При наличии уязвимой к переполнению буфера подпрограммы и примитива чтения в подпрограммах не в единой последовательности вызываемых подпрограмм атака перестает быть возможна (то есть нахождение в разных ветках дерева вызовов ГПУ).

5. Меры по устранению опасных значений. Проанализируем возможные места размещения ОЗ и эффективные меры по их устранению. В соответствии с разделом 3, такими местами являются:

1. Модификация последовательностей инструкций перед *RET* должна осуществляться таким образом, чтобы исключить чтение со сдвигом. Такое чтение в ходе атаки позволяет интерпретировать фрагменты существующих инструкций как необходимые атакующему другие инструкции. Защита обеспечивается за счет формирования перед

RET такой последовательности инструкций, которая не может быть интерпретирована атакующим как содержащая полезные для атаки операции.

2. Фрагменты инструкций, не входящие в множество ОИ, подлежат синонимической замене с сохранением алгоритма оригинальной программы.

3. Области размещения данных, по технологическим причинам являющиеся исполнимыми, должны быть лишены признака исполнимости.

Далее приводятся детали реализации для каждой из предложенных мер.

5.1. Противодействие исполнению со сдвигом. Первой задачей противодействия является оценка возможности использования атакующим кода системы защиты, находящегося в эпилоге, при исполнении его со сдвигом.

Второй задачей является поиск в коде оригинальной программы гаджетов, не входящих в состав эпилогов и получаемых за счет исполнения со сдвигом. Включающие гаджеты инструкции подлежат замене на синонимичные, которые исключают их использование атакующим.

Для решения обеих задач используется алгоритм *GALILEO*, описанный в [21] и позволяющий определить пригодность использования ОЗ и расположенных перед ним инструкций в качестве гаджета. Данный алгоритм обеспечивает исключение из рассмотрения таких ОЗ, перед которыми содержатся последовательности байт, которые не могут быть интерпретированы как инструкции или приводят к аварийному завершению программы. Код системы защиты подлежит анализу для определения того, увеличивает ли он число гаджетов относительно оригинального приложения.

5.2. Синонимическая замена опасных значений в составе инструкций. Для защиты от неверной интерпретации код программы должен быть проанализирован в соответствии с разделом 5.1 для определения инструкций, подлежащих защите. ОЗ включают инструкции возврата в пределах сегмента (0x*C*3, 0x*C*2) и инструкции возврата за пределы сегмента (двухбайтовые последовательности «0x48 0x*CA*» и «0x48 0x*CB*»). В результате анализа оригинальной программы формируется перечень инструкций, подлежащих защите путем синонимической замены.

Для каждой из них необходимо определить, в каком качестве в составе инструкции содержится ОЗ, для того чтобы сформировать эквивалентные замены. Из проверки исключаются ОИ из состава эпилогов, так как они защищаются вставкой кода применения к ним K_i.

Для контроля корректности работы алгоритма поиска могут использоваться существующие средства поиска гаджетов (например, [22]). Критерием корректности является то, что состав найденных по предлагаемой методике ОЗ должен быть не меньше, чем у существующих средств.

Под синонимической заменой инструкции, содержащей ОЗ, будем понимать такую замену, чтобы состояние используемых регистров и ячеек ОЗУ после исполнения оригинального и модифицированного участка не отличалось для любого исходного состояния используемых регистров и ячеек ОЗУ на начало рассматриваемого участка. Критерий использования сформулирован далее.

Рассмотрим виды составных частей инструкций, которые могут содержать ОЗ. Учет их специфики необходим для формирования порядка генерации синонимичных замен. Непосредственно байт, который интерпретируется как ОЗ, не может входить в состав префиксов (так как префикс не может совпадать с существующими операциями) и кодов операций первого уровня. Согласно [4, 19], ОЗ могут встречаться в следующих участках инструкций:

– аргумент данных операции, являющийся адресом (относительным или абсолютным);

– параметры операции, задающие режим ее выполнения, или операнды (*ModRM, SIB* компоненты инструкции);

– аргумент данных операции, не являющийся адресом;

– второй и третий байт многобайтного префикса (значения 0x*C*2 и 0x*C*3 являются допустимыми значениями, а 0x*CA* и 0x*CB* неприменимы, так как необходимый им префикс 0x48 не входит в область допустимых второго байта трехбайтного префикса и не может являться первым байтом ни двухбайтового, ни трехбайтового префикса);

– код операции второго или большего уровня, интерпретируемый как опасная операция.

Вариант с выявлением в полях, содержащих адрес необходимого атакующему значения, решается следующими способами:

1. При вхождении ОЗ в состав относительного адреса до (если относительный адрес является отрицательным числом) или после (если он положительный) в содержащую ОЗ подпрограмму выполняется вставка последовательности байт «0x90» такого размера, чтобы относительный адрес после коррекции с учётом вставки не содержал ОЗ;

2. При нахождении ОЗ в младшем байте абсолютного адреса выполняется перемещение целевого участка в конец содержащей его

секции (исходное место не используется). Новое размещение выбирается так, чтобы его адрес не содержал ОЗ;

3. При нахождении ОЗ не в младшем байте абсолютного адреса мер по его защите не требуется, так как вследствие работы механизма *ASLR* значение при начале работы программы будет изменено на случайное.

Процесс устранения опасных относительных адресов проводится итерационно, с контролем сформированных заменяющих адресов на возможное порождение новых уязвимых последовательностей. Возможна вставка региона, размер которого является диапазоном. Это может привести к порождению опасного участка. Тогда размер вставки изменяется и проверяется следующий размер. Сходимость процесса обеспечивается тем, что количество подлежащих перемещению участков монотонно убывает из-за того, что перед перемещением проверяется порождение опасных участков.

При нахождении опасного значения в префиксе или расширенном коде операции для его использования атакующим необходимо передать управление до начала опасной инструкции. Поэтому для устранения данного гаджета достаточно перед инструкцией вставить цепочку однобайтовых инструкций, которые делают некорректное исполнение невозможным (например, *NOP*). Для определения размера необходимой вставки используется подход, описанный в подразделе 5.1 для оценки возможности использования байта в инструкции, описанной ранее. Перебираются размеры от 1 до 14.

Для ситуаций, когда опасный байт входит в состав аргумента данных инструкции, или указания её операндов опасная инструкция должна быть заменена на две и более инструкции, совокупное выполнение которых дает эквивалентное преобразование состояний программы тому, которое осуществляла защищаемая инструкция, и не содержит опасных последовательностей. В ходе такой замены необходимо контролировать, чтобы выполнение результатов синонимической замены не оказывало влияние на логику целевой программы. Это требует использования только таких регистров или ячеек ОЗУ, которые не используются целевой программой. Если таких нет, то требуется сохранение с последующим восстановлением ресурсов, для которых присутствует конфликт использования. Критерием отсутствия использования является то, что значение регистра в дальнейшем в рамках анализируемой подпрограммы будет утрачено вследствие затирания другим значением без предшествующего этому чтения. Вследствие этого оно никак не может повлиять на работу программы. Место под

замещающую последовательность выделяется тем же алгоритмом, что используется для резервирования места в ходе защиты эпилогов.

В рамках опасного значения в составе операндов эквивалентная замена достигается путем замены операндов инструкции. Опасное значение для байта *ModRM* состоит из: режима, равного 3 (операндами являются регистры), поля, определяющего регистровый аргумент «*reg*», равного 2 или 3 (что соответствует регистрам *RDX/R*10 и *RBX/R*11), и поля «*r/m*», равного 0 или 1 (что соответствует регистрам *RAX/R*8 и *RCX/R*9). Для байта «*SIB*» аналогичные опасные значения должны содержаться в полях «*base*» и «*index*» с теми же значениями.

Для того чтобы сделать непригодным к использованию данный гаджет, достаточно заменить любой из операндов на регистр, который не входит в состав опасных. Если инструкция принимает только входной регистр или один из регистров является входным, а другой выходным, то эквивалентной заменой будет вставка перед защищаемой инструкцией команды *MOV*, которая копирует данные из опасного регистра-операнда в свободный регистр, который не входит в состав опасных. В самой инструкции опасный регистр заменяется на свободный, который к моменту выполнения инструкции будет содержать то же значение, что и в оригинальной программе.

Если единственным операндом инструкции является выходной регистр или регистр, который одновременно является и входным, и выходным, то выполняется вставка двух дополнительных инструкций: перед – запись в свободный регистр значения из опасного, после – запись в опасный регистр значения из свободного. Отметим, что инструкции перемещения значений между регистрами не влияют на регистр флагов, что не нарушает логики работы программы. В самой инструкции производится аналогичная замена опасного регистра на один из свободных и не входящих в перечень опасных. Алгоритм определения свободных регистров приведен далее.

При устранении ОЗ в составе непосредственного операнда данных принципиальным является то, имеет ли использующая опасное значение в качестве операнда инструкция альтернативную форму, в которой вместо непосредственного значения используется операнд в регистре. Необходимо отметить, что подавляющее большинство инструкций, которые могут использовать непосредственный операнд, имеют альтернативную форму с регистровым операндом.

Если альтернативная форма с регистровым операндом существует (например, инструкция *ADD*), то оригинальная инструкция заменяется на последовательность:

– если свободных регистров нет, то выполняется временное освобождение регистра путем записи его значения в стек (в конце последовательности значение должно быть восстановлено из стека);

– загрузка в свободный регистр преобразованного операнда. Для устранения опасного значения операнд хранится в преобразованном виде (например, инвертированном или циклически сдвинутом) так, чтобы преобразованный вид не содержал опасных значений;

– восстановление значения операнда путем выполнения обратной операции – той, которая использовалась для его преобразования;

– выполнение регистровой формы оригинальной инструкции с оригинальным операндом.

Если не существует альтернативной формы оригинальной инструкции (например, инструкция *ENTER*), то производится дополнение со стороны меньших адресов защитными инструкциями, которые исключают неправильную трактовку защищаемой инструкции. Дополнительно необходимо учитывать особенности:

– не для всех инструкций опасное значение входит в область допустимых значений. Например, приведенная инструкция *ENTER* принимает в качестве аргументов два непосредственных операнда. Опасное значение не может содержаться в младшем байте первого операнда, так как это нарушает выравнивание. Кроме того, опасное значение не может содержаться во втором операнде, так как его максимальное значение составляет 32. Маловероятно, что опасное значение будет содержаться в старшем байте первого операнда, так как фреймы стека размером в 32 кБайта представляют редкость. Вероятность появления снижает то, что опасные значения составляют 2 из 255 возможных значений старшего байта первого операнда, и то, что компиляторы в принципе не используют такие инструкции (что не сужает область применимости относительно указанной в третьем разделе);

– вторая и третья найденные инструкции, не имеющие альтернативной регистровой формы (*LWPINS* и *LWPVAL*), применимы только на этапе профилирования приложения и в генерируемом коде компилятора не встречаются (что не сужает область применимости относительно указанной в третьем разделе);

– иных инструкций, помимо перечисленных только с формой, использующей операнд, согласно документации [4, 19], не существует.

Поиск существующих решений, позволяющих определить используемые программой ресурсы процессора, показал, что единствен-

ной публично доступной реализацией является [21], принцип описан в [22]. Анализ данного решения показал, что оно не учитывает БИП, вследствие чего каждый вызов подпрограммы приводит к пометке всех регистров как используемых, что ведет к неверной оценке состава используемых регистров. Было принято решение разработать новый алгоритм определения свободных ресурсов, учитывающий БИП. Перед формулированием алгоритма рассмотрим теоретические сведения, необходимые для этого.

Для определения свободных ресурсов процессора необходим анализ полной последовательности инструкций подпрограммы, в которой находится подлежащая защите инструкция. Это связано с тем, что ряд регистров используется достаточно редко и при анализе неполного содержимого подпрограммы может быть сделан неверный вывод об отсутствии их использования. Анализу подлежат входные и выходные аргументы инструкций для определения регистров, которые могут быть задействованы в модифицированном участке и не повлияют на эквивалентность вычислений. Регистр может считаться свободным, если содержащееся в нем значение не может быть использовано в дальнейших вычислениях. Так как описанные выше меры оперируют регистрами целиком, то для того, чтобы регистр считался свободным, должны быть свободны все разряды регистра.

Пример: значение, содержащееся в регистре после считывания, в дальнейшем не считывается повторно, а вместо этого перезаписывается другим значением. Приведенный пример показывает, что необходимо рассматривать свободные ресурсы строго с точки зрения шагов выполнения программы (то есть свободность ресурса может быть выявлена только для диапазона конкретных инструкций). Учитывая БИП, максимальным участком анализа разумно выбрать подпрограмму. Подпрограмма представляется в виде набора линейно исполняющихся участков кода – базовых блоков (далее – ББ), переходы между которыми осуществляются согласно ГПУ.

Определение свободности ресурсов в рамках ББ (то есть когда последовательность выполнения детерминирована) является тривиальной задачей. При необходимости определения в границах больших, чем ББ (например, при необходимости вставки перед первой инструкцией ББ), необходимо учитывать все возможные переходы на все рассматриваемые ББ (если достоверно не определено, что возможен только один вариант перехода).

Если в ходе анализа используемых ресурсов встречаются инструкции, создающие сторонние эффекты, то возможны следующие варианты действий:

– при анализе инструкций вызова подпрограмм из состава защищаемого приложения должен быть учтен БИП как максимально ограничивающий состав свободных ресурсов. БИП определяет, из каких регистров считываются данные, в какие регистры записывается результат исполнения, какие сохраняют свое значение и какие будут иметь неопределенное значение в результате вызова подпрограмм. При недостатке ресурсов может быть проведен дополнительный анализ использования регистров в начале и в конце вызываемой подпрограммы. Если соответствующие ресурсы находятся в свободном состоянии, то должен быть сделан вывод о том, что они свободны, до и после вызова анализируемой вызываемой подпрограммы;

– если для системных вызовов и вызовов библиотечных функций не определен БИП, должно быть принято утверждение, что модифицируются все регистровые ресурсы. Если значения каких-либо ресурсов должны быть сохранены в интересах системы защиты, то до исполнения они должны быть сохранены (не обязательно непосредственно перед вызовом) и восстановлены на следующей инструкции, после создающей сторонние эффекты. Вследствие того, что в работе учитывается БИП, это дает априорную информацию об используемых ресурсах. Это позволяет сохранять только необходимые ресурсы, что снижает накладные расходы. Для ряда ситуаций сохранение значений непосредственно перед исполнением является недопустимым, так как нарушает эквивалентность состояний. Например, рассмотрим сохранение единичного восьмибайтового значения непосредственно перед инструкцией *CALL*, которая передает управление на подпрограмму со стандартным прологом, которая принимает аргументы через стек. Тогда адрес $RBP\text{-}18_{16}$ будет указывать вместо первого аргумента на сохраненное значение, что нарушит эквивалентность программ. Такая проблема ассоциирована со стеком, так как код программ содержит массово работу по относительным адресам при передаче аргументов и хранении локальных переменных. Безопасны для помещения значений в стек только участки подпрограмм двух видов: а) после выделения всех локальных переменных и до выделения первого блока аргументов для вызова вложенной подпрограммы, а также б) после восстановления стека, следующего за вызовом подпрограммы и помещением в стек аргументов для следующей подпрограммы.

Если модифицированный участок требует специфического набора ресурсов, и они на участке вставки не являются свободными, то они должны быть сформированы путем сохранения текущих целевых значений в прологе модифицированного участка и восстановления в эпилоге. Примером является инструкция *RDTSC*, которая записывает

результат своей работы в фиксированные регистры, что приводит к безвозвратной потере содержащихся в них целевых значений. Теоретически, возможным вариантом являлось бы сохранение значений всех регистров (например, последовательностью операций *PUSH* или инструкцией *PUSHAD* для 32-разрядного режима), но это существенно повышает накладные расходы. А с учетом массовой вставки кода для защиты инструкция возврата замедляет программу гарантированно неприемлемым образом.

Необходимо отметить, что в анализе ресурса памяти в предлагаемой системе защиты нет необходимости, вследствие чего данный вопрос не рассматривался.

Непосредственное определение свободных ресурсов, во-первых, сводится к определению регистров, запись в которые не приведет к нарушению эквивалентности состояний программы, а во-вторых, записанные в интересах системы защиты значения не будут повреждены в ходе вычислений целевой программы. На основе данных утверждений можно сформулировать критерий отнесения ресурсов к свободным:

– регистр считается свободным до инструкции, которая произведет запись в него;

– если после записи значение из регистра не считывается до выполнения повторной записи, то регистр должен рассматриваться как свободный с момента первой записи. При этом сама первая операция должна считаться избыточной в части модификации регистра;

– после помещения в регистр неопределенного значения (например, после возврата из подпрограммы в несохраняемых регистрах, согласно БИП) он считается свободным до записи в него значения.

Модификация свободных регистров не влияет на выполнение программы. Вследствие этого их использование в рамках кода системы защиты гарантированно не модифицирует алгоритм оригинальной программы.

Состояние одного регистра не влияет на состояние другого. То есть используемые регистры определяются машинным кодом и не связаны с результатами вычислений, так как архитектура команд не предусматривает непрямое указание регистров-операндов. Вследствие этого поиск свободных ресурсов допустимо проводить для каждого регистра независимо. Дальнейшее описание приведено для одиночного регистра, для получения полной информации процедура повторяется для всех необходимых регистров.

Пусть анализируемая подпрограмма состоит из набора инструкций f_j, $1 \le j \le n$, где n – количество инструкций в подпрограмме. Перед

определением состояния ресурсов выполним подготовительные операции. Согласно документации на процессор и БИП, определим для каждой f_j, является ли анализируемый регистр входным аргументом (читается), и запишем результат в массив *RI* размером n, индексы элементов которого соответствуют индексам инструкций. Выполним аналогичное определение с точки зрения того, является ли регистр выходным, и запишем результат в массив *RO*. Элементы массивов могут принимать значения: используется (*Y*), не используется (*N*), не определен (*U*). Результат определения записывается в массив *RF*, имеющий аналогичный размер, элементы которого принимают значения: свободен (*F*), занят (*U*).

БИП определяет порядок передачи аргументов, но не все используемые для этого регистры могут быть задействованы. Для их определения может быть либо проведен анализ вызываемой подпрограммы, либо проанализирована семантика инструкций подпрограммы. При втором подходе все инструкции между записывающей в регистр неопределенное значение и потенциально считывающей его в качестве аргумента вызываемой подпрограммы должны быть помечены как не использующие регистр. Для этого вводится массив *RR* размером n, значения которого принимают значения: инструкция предшествует чтению (*Y*) или нет (*N*).

Для проведения анализа необходимы сведения о ГПУ подпрограммы. Множество инструкций разбивается на ББ $b_k = (j_{bk}, j_{ek})$, где j_b и j_e – индексы первой и последней инструкции ББ, $1 \le k \le n_b$, n_b – число ББ. Для ББ определяются связи между ними (для каждого определяется перечень, ссылающихся на него ББ). Также определяется перечень эпилогов, которыми заканчивается выполнение анализируемой подпрограммы, $E = \{e_1, \dots, e_m\}$, где e – номера ББ, включающие эпилоги, а m – число эпилогов в подпрограмме. Для учета порядка обработанных блоков используется массив *PBB* размером n_b, элементы которого принимают значения: обработан (*P*), запланирован к обработке (*S*), не обработан (*U*, значение по умолчанию).

Разработанный алгоритм определения свободных ресурсов с учетом БИП приведен на рисунке 2.

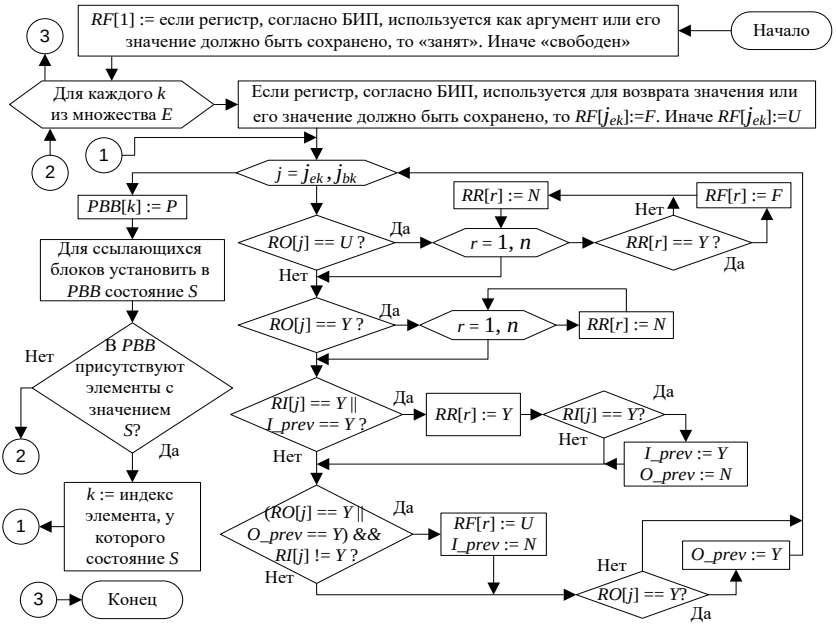

Рис. 2. Алгоритм определения свободных ресурсов

В результате выполнения для каждого регистра указанного алгоритма каждой инструкции анализируемой подпрограммы будет сопоставлен перечень свободных ресурсов. Для опасной инструкции он используется для формирования эквивалентной замены.

При описанном выше порядке вызываемые подпрограммы рассматриваются как «черный ящик»; выдвигаются предположения о том, что для передачи аргументов и возврата значений используется максимально возможный состав регистров в рамках БИП. Состав входных и выходных регистров для вызова подпрограммы может быть уточнен при необходимости (например, при недостатке свободных ресурсов) путем анализа содержимого вызываемой подпрограммы. Порядок такого анализа совпадает с описанным ранее, а сведения о считываемых и модифицируемых ресурсах вызываемой подпрограммы интерпретируются как эффект от выполнения инструкции вызова подпрограммы.

Отсутствие же использования ресурса в рамках всех инструкций программы свидетельствует о том, что любые операции с ним не повлияют на эквивалентность программ. Если имеется априорная информация об отсутствии использования ресурса в рамках кода программы и не входящих в состав программы библиотек (например,

вследствие ограничений на состав используемых компилятором инструкций) или гарантий отсутствия модификации, используемых системой защиты ресурсов системными библиотеками и системными вызовами, то она должны быть учтена для снижения времени анализа. Априорно не используемые ресурсы помечаются свободными.

5.3. Исключение данных из состава исполнимой памяти. Для реализации этой меры защиты неисполняемые данные должны быть исключены из перечня исполнимых областей. Тогда содержащиеся в них байты с опасными значениями не могут быть использованы атакующими как гаджеты. Для определения атрибута исполнимости применяется технология «*NX-bit*». Она позволяет разрешать исполнение инструкций только для конкретных участков программ (как минимум участка, содержащего машинный код). Перечень исполнимых участков содержится в заголовке образа программы.

Единицей указания атрибута исполнимости является страница памяти (в типовом случае размером 4096 байт). Для того чтобы в исполнимую память не были включены данные, применяется выравнивание. Перед участком кода и после него помещается массив байтов со значениями, которые не могут быть применены в составе гаджетов. Размер массива выбирается так, чтобы адреса начала и конца исполнимого участка памяти были кратны размеру страницы. В качестве значений байтов используется «0x*CC*», так как передача на них управления вызовет отладочное прерывание. Исходная секция, включающая данные до исполнимого кода, сам исполнимый код и данные после него разбиваются на три соответствующие секции, причем исполнимой (бит *NX* сброшен) является только вторая.

6. Программная реализация. Предлагаемые решения были апробированы на ОС *Debian* 9, относящейся к семейству *GNU*/*Linux*. Данная ОС использует БИП «*ABI Linux-AMD*64» [24]. Для реализации программной реализации использовалась среда *QtCreator*, компилятор *gcc* 6.3.0.

Общая архитектура программного средства (далее – ПС) приведена на рисунке 3. Заимствуемые в рамках данной работы компоненты обеспечивают анализ структуры и связей программы, а также восстановление корректной ссылочной структуры с учетом добавляемого кода системы защиты. Анализатор подпрограмм обеспечивает определение границ подпрограмм и их параметров, таких как: границы прологов и эпилогов подпрограмм, выделяемое в стеке место под сохраняемые в ходе работы подпрограммы регистров и локальные переменные. Алгоритм определения ресурсов приведен ранее. Логика функционирования средства поиска опасных участков описана ранее.

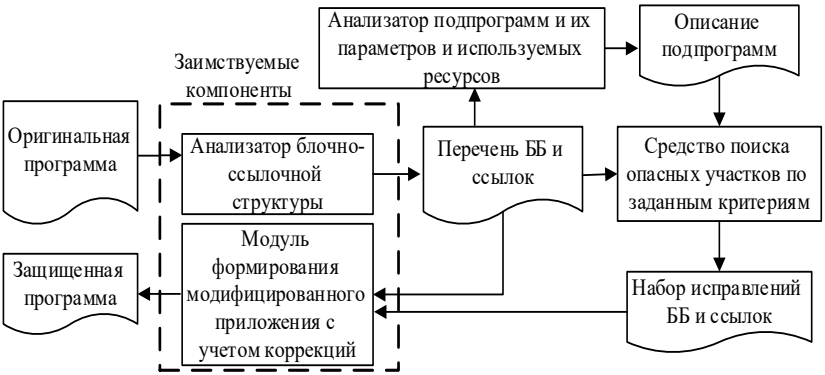

Рис. 3. Архитектура ПС

Для обеспечения автоматического тестирования ПС для управления был выбран интерфейс командной строки, а для вывода данных – текстовое представление. Посредством интерфейса командной строки задаются основные параметры, такие как имя оригинальной программы, имя выходного файла для записи защищенной программы, вид используемого источника K_i, детальность выводимой диагностической информации. Для отражения успешности применения системы защиты используется код завершения процесса (0 в случае успеха, иначе значение, указывающее на этап, вызвавший сбой). В консоль выводится диагностическая информация с настраиваемым уровнем детализации. Например, выводится информация о сегментах и секциях оригинальной программы, ББ, ссылках, найденных ОЗ, подпрограммах, а также об операциях по формированию защищенной программы. При сбое работы выводится диагностическая информация для устранения проблемы.

Для реализации защиты эпилогов используются вставки, представленные в таблице 3. Приведен пример при использовании инструкции *RDRAND*. Защите подлежат только подпрограммы, хотя бы один эпилог которых заканчивается инструкцией *RET* (не все подпрограммы заканчиваются такими инструкциями). Код приведенных вставок не содержит ОЗ.

Таблица 3. Вставки кода для защиты штатных инструкций возврата

Место вставки	Мнемоническое представление	Байтовое представление, шестнадц.	Примечание

В месте завершения пролога	rdrand r10
	movd r11 , xmm14
	xor [r11], r10
	push r11
	lea r11 , [rsp+**X**]
	movd xmm14, r11
	xor [r11], r10
	movd r11 , xmm15
	xor r11 , r10
	push r11
	movd xmm15, r10
		49 0F C7 F2
		66 4D 0F 7E F3
		4D 31 13
		41 53
		4C 8D 9C 24 **XX XX XX XX**
		66 4D 0F 6E F3
		4D 31 13
		66 4D 0F 7E FB
		4D 31 D3
		41 53
		66 4D 0F 6E FA
			X – смещение до адреса возврата; свободные РОН – *R*10 и *R*11; для хранения текущего ключа и адреса возврата выбраны не используемые в программе регистры расширения *XMM*15 и *XMM*14

Перед эпилогом	pop r9 pop r8 movd xmm14, r8	41 59 41 58 66 4D 0F 6E F0	Регистры *R*8, *R*9 свободны перед началом эпилога, согласно БИП

Перед инструкцией *RET*	movd r11, xmm15
	xor [r8], r11
	xor r9 , r11
	movd xmm15, r9
	xor [rsp], r11
		66 4D 0F 7E FB
		4D 31 18
		4D 31 D9
		66 4D 0F 6E F9
		4C 31 1C 24
			Регистр *R*11 также свободен к завершению подпрограммы

В начале подпрограммы *_start*	rdrand eax
	push rax
	movd xmm14, rsp
	movd xmm15, rax
		0F C7 F0
		50
		66 4C 0F 6E F4
		66 4C 0F 6E F8
			Подпрограмма *_start* не требует защиты, так как заканчивается инструкцией *HLT*

Анализ вставляемого перед инструкцией *RET*-кода показывает следующие возможности интерпретации при передаче управления за указанное количество байтов до самой инструкции *RET*:

– 1 байт – последовательность «*AND AL*, *C*3» (шестн. 24 *C*3), которая не может быть использована как гаджет (то есть байт *C*3 интерпретируется как часть инструкции «*AND*» и не может быть исполнен);

– 2 байта – «*SBB AL*, 24_{16}; *RET*» (шестн. 1*C* 24 *C*3) – потенциально может быть использована как гаджет, выполняет вычитание из младшего байта регистра *RAX* константы 24_{16};

– 3 байта – «*XOR* [*RSP*], *EBX*, *RET*» – такая инструкция не может быть использована как гаджет, так как не модифицирует регистров, необходимых для вычислений или управления программой, но при известном атакующему значении в регистре *EBX* может быть использована для передачи управления на другой гаджет (путем записи на вершину стека значения R_{gadget} *XOR EBX* при предшествующем повреждении стека);

– более 3 байт – перед *RET* неизменно будет инструкция «*XOR* [*RSP*], *R*11», которая преобразует значение адреса возврата на неизвестное атакующему значение (регистр *RSP* указывает на адрес возврата, так как защитный код вставлен перед инструкцией *RET*).

При применении описанного комплекса мер даже если атакующий получит возможность влиять на ГПУ (например, посредством атак на таблицы виртуальных функций), то в его распоряжении будет всего один гаджет, влияющий на РОН, не используемый для передачи аргументов в подпрограммы, и возможность передавать управление на другие такие же гаджеты, чего недостаточно для эксплуатации уязвимости, согласно модели атаки.

7. Экспериментальная проверка корректности и эффективности программного средства. Предлагаемый метод защиты влияет на стек приложения, а для *RoP*-атак даже минимальное изменение расположения данных в стеке приводит к их неработоспособности. Вследствие этого проверка со сравнением работоспособности эксплойта для оригинального и защищенного приложения не является показательной. Проверка работоспособности эксплойтов требует разработки отдельной методики и выходит за рамки данной статьи. Проверка корректности реализации проводилась по трем направлениям: неизменность логики работы программ, тестирование производительности и устранение пригодных для проведения атак гаджетов.

Для контроля неизменности логики работы применялись приложения (синтетические тесты, программы тестирования производительности и программы, снабженные модульными тестами), для которых исполняются все участки, для которых вносились изменения (это проверялось средствами контроля покрытия кода тестами *gcov*). Вследствие того, что выше приведено обоснование эквивалентности вносимых изменений (то есть отсутствия влияния на логику работы), то для оценки корректности не важна вариативность входных данных. Неизменность логики работы оценивалась по идентичности выводи-

мых программой данных при одинаковых входных (для оригинальной и защищенной программ) и отсутствию сбоев в работе. Результаты показали корректность и неизменность работы алгоритмов. Примеры эквивалентных замен для устранения гаджетов приведены в таблице 4. Для контроля устранения пригодных для использования гаджетов использовалось средство их поиска «*ROPgadget*» [25]. Анализ показал, что во всех обработанных программах отсутствуют иные гаджеты, кроме приведенных в разделе «Программная реализация». Примеры оригинальных и защищенных программ приведены в публичном репозитории github.com/LubkinIvan/zpk/tree/main/examples

Таблица 4. Примеры эквивалентных замен для устранения гаджетов

Содержимое памяти оригинальной программы	Опасный участок оригинальной программы	Опасный участок после эквивалентной замены	Содержимое памяти защищенной программы	Примечание
01 **C2**	ADD EDX EAX	MOV RCX, RAX ADD EDX, ECX	48 89 C1 01 **CA**	Опасное значение в *ModRM*

48 83 **C3** 01	ADD RBX, 1	MOV RCX, RBX
		ADD RCX, 1
		MOV RBX, RCX
			48 89 D9
			48 83 **C1** 01
			48 89 CB
				Опасное значение в *ModRM*

E8 **C3** FF FF FF	CALL -3D	NOP NOP CALL -3F	90 90 E8 **C1** FF FF FF	Опасное значение в операнде *Imm*32

Сравнение производительности выполнялось программным средством «*coremark*» [26]. Оно было выбрано, так как дополнительно контролирует целостность результатов расчетов. Замеры производительности показали падение относительно оригинальной программы: при использовании *RDRAND* – 91 %, *RDTSC* – 53 %, внешний источник ГПСЧ – 14 % (число запусков 100 для каждого). Характеристики тестовой системы: процессор *AMD A*6-6310 1,8 ГГц, 4 ядра, 4 ГБ ОЗУ, НЖМД 7200 об/мин.

Прямое сравнение предлагаемого средства с аналогами не является в полной мере корректным, так как они рассчитаны на несовпадающую область применения. Дополнительной проблемой является то, что для перечисленных в разделе 2 ближайших программных ана-

логов не опубликованы результаты их применения, вследствие чего возможно сравнение только на качественном уровне. Отметим также, что сравнение производительности носит справочный характер, так как если аналог имеет меньшие накладные расходы, но не обеспечивает защиту, то это достоинство не играет роли. Результаты сравнения приведены в таблице 5.

Таблица 5. Сравнение решений для противодействия *RoP*-уязвимостям

Средство защиты	Не анализируемые области	Устранение гаджетов	Уязвимость при наличии примитива чтения	Треб. исх. текст
PaXgrsecurity RAP	Исполнимые данные	Нет	Да	Да
Обфускатор машинного кода	Технологические участки и исполнимые данные	В анализируемых областях	Да	Да
Selfrando	Отсутствуют	Нет	Да	Да
Shuffler	Технологические участки и исполнимые данные	В части эпилогов	Между интервалами переразмещения	Нет
RuntimeASLR	Отсутствуют	Нет	Да	Нет
G-free	Технологические участки и исполнимые данные	Да	Нет	Да
Scylla	Технологические участки и исполнимые данные	Нет	В рамках расшифрованной области	Да
Разработанное решение	Отсутствуют	Да	Использование примитива чтения и повреждения стека в рамках вызова одной подпрограммы	Нет

8. Заключение. В данной статье были предложены оригинальная методика защиты от *RoP*-уязвимостей и её алгоритмическое обеспечение.

Предложенная методика поиска уязвимых участков и меры по их устранению обеспечивают выявление и эквивалентные замены участков, которые могут использоваться для атак. Научная новизна

заключается в обеспечении защиты в условиях наличия у злоумышленника почти полной информации об атакуемой программе (кроме ключей защиты K_i).

Разработанный алгоритм определения свободных ресурсов позволяет встраивать код защиты, не нарушая логику работы программ. Данный алгоритм учитывает БИП, что позволяет, в отличие от аналогов, получать более точные сведения о свободных регистрах. Методика защиты эпилогов от использования в составе гаджетов, с одной стороны, затрудняет получение атакующим контроля над ГПУ, а с другой стороны, не позволяет использовать их как гаджеты, если контроль над ГПУ был получен иным способом. Отличием от существующих средств является отсутствие общих для подпрограмм ключей защиты, компрометация которых делает уязвимой всю программу. Алгоритмическое обеспечение методики позволило реализовать ее в виде ПС.

Проведенная экспериментальная оценка программного средства показала, что данное средство обеспечивает устранение гаджетов, которые не основаны на инструкциях *RET* из состава эпилогов. Для эпилогов обеспечивается блокирование работы программы при попытке их использования атакующим. Для случаев, когда атакующий проводит атаки по сети, падение производительности находится на уровне менее защищенных аналогов.

Предлагаемое решение в ходе обеспечения защиты создает накладные расходы. Вследствие этого не имеет смысла его массовое применение. Наиболее целесообразным является применение для защиты сетевых сервисов, с которыми может взаимодействовать атакующий, и для приложений, обрабатывающих поступающие извне защищенной информационной системы данные. В описанных ситуациях успешная эксплуатация уязвимостей удаленного исполнения кода создает опасность нанесения неприемлемого ущерба, что перевешивает создаваемое системой защиты замедление работы.

Дальнейшее развитие видится в апробации для ОС *Windows* и разработке методики для оценки применимости реальных эксплойтов для защищенных приложений с учетом вносимых системой защиты в стек изменений.

Литература

1. Гласс, Р. Факты и заблуждения профессионального программирования // СПб.: Символ-Плюс. 2007. 240 с.
2. Вишняков А.В. Классификация ROP-гаджетов // Труды ИСП РАН. 2016. Т. 28. Вып. 6, с. 27–36. DOI: 10.15514/ISPRAS-2016-28(6)-2
3. Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity // Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and

Privacy (HASP '19). Association for Computing Machinery, New York, NY, USA. 2019. Article 8, 1–11. DOI: https://doi.org/10.1145/3337167.3337175
4. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 // https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf
5. Intel Launches World's Best Processor for Thin-and-Light Laptops: 11th Gen Intel Core // https://www.intel.com/content/www/us/en/newsroom/news/11th-gen-tiger-lake-evo.html
6. RAP: RIP ROP 2015 // https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf
7. Koo, Z.Z., Ayop, Zakiah, Abidin, Z.Z. Analysis of ROP attack on grsecurity / PaX linux kernel security variables // International Journal of Applied Engineering Research. 2017. no. 12. pp. 13179–13185.
8. Иванников В., Курмангалеев Ш., Белеванцев А., Нурмухаметов А., Савченко В., Матевосян Р., Аветисян А. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM // Труды ИСП РАН. 2014. Т. 26. Вып. 1. С. 327–342.
9. Нурмухаметов А.Р., Курмангалеев Ш.Ф., Каушан В.В., Гайсарян С.С. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения // Труды ИСП РАН. 2014. Т. 26. Вып. 3. С. 113–126. DOI: 10.15514/ISPRAS-2014-26(3)-6.
10. ИСП Обфускатор. Технология запутывания кода для защиты от эксплуатации уязвимостей // https://www.ispras.ru /technologies/isp_obfuscator/
11. 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. 2017. Т. 29. Вып. 6. С. 163–182. DOI: 10.15514/ISPRAS-2017-29(6)-9.
12. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE. 2016.
13. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits // PoPETs. 2016. no. 4. pp. 454–469.
14. 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.
15. Kangjie Lu, Stefan Nürnberger, Michael Backes, Wenke Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization // Proceedings of the 23rd Annual Network and Distributed System Security Symposium. 2016.
16. Onarlioglu K., Bilge L., Lanzi A., Balzarotti D., Kidra E. G-Free: Defeating return-oriented programming through gadget-less binaries // Proceedings of ACSAC: M. Franz and J. McDermott, Eds. ACM Press. 2010. pp. 49–58.
17. Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, Sina Bahram. Defeating return-oriented rootkits with «return-less» kernels. // Proceedings of EuroSys. 2010, edited by G. Muller. ACM Press. pp. 195–208.
18. Dean Sullivan, Orlando Arias, David Gens, Lucas Davi, Ahmad-Reza Sadeghi, Yier Jin. 2017. Execution Integrity with In-Place Encryption. arXiv preprint arXiv:1703.02698 (2017).
19. Lubkin I.A., Subbotin N.A. Technique of verified program module modification with algorithm preservation // IEEE Xplore Digital Library. 2017. 11th International IEEE scientific and technical conference "Dynamics of systems, mechanisms and machines" (Dynamics), 2017. pp. 1–5.

20. Lubkin I.A., Bazhenov I. O. Methodology of software code decomposition analysis // Dynamics of systems, mechanisms and machines. Omsk. 2018. pp. 1–5.
21. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007. ACM Conference on Computer and Communications Security (CCS), Proceedings of CCS, 2007. pp. 552–561.
22. Статья Permutation conditions. URL: https://z0mbie.dreamhosters.com/pcond.txt (дата обращения 01.09.2021).
23. Репозиторий с исходным кодом библиотеки eXtended Disassembler Engine (version 1.02). URL: https://github.com/nimrood/xde (дата обращения 01.09.2021).
24. AMD64 Architecture Processor Supplement Draft Version 0.99.7 // https://www.uclibc.org/docs/psABI-x86_64.pdf
25. Инструмент ROPgadget. Репозиторий с исходным кодом. URL: https://github.com/JonathanSalwan/ROPgadget (дата обращения 01.09.2021).
26. Coremark. Программа оценки производительности. URL: https://github.com/eembc/coremark (дата обращения 01.09.2021).

Лубкин Иван Александрович — старший преподаватель, кафедра безопасности информационных технологий, СибГУ им. М.Ф. Решетнёва. Область научных интересов: безопасность операционных систем, защита программного кода от несанкционированного использования, противодействие уязвимостям. Число научных публикаций — 27. lubkin@rambler.ru; проспект им. газеты Красноярский рабочий, 48Б, 660037, Красноярск, Россия; р.т.: +7 (391) 222-76-39; факс: +7(391)222-76-39.

Золотарев Вячеслав Владимирович — канд. техн. наук, доцент, заведующий кафедрой, кафедра безопасности информационных технологий, СибГУ им. М.Ф. Решетнёва. Область научных интересов: управление информационной безопасностью, противодействие уязвимостям. Число научных публикаций — 98. zolotarev@sibsau.ru; проспект им. газеты Красноярский рабочий, 48Б, 660037, Красноярск, Россия; р.т.: +7(391)222-76-39.

Поддержка исследований. Исследование выполнено при финансовой поддержке Минобрнауки России (грант ИБ). Соглашение № 21/2020.

DOI 10.15622/ ia.21.2.3

I. LUBKIN, V. ZOLOTAREV

COMPREHENSIVE DEFENSE SYSTEM AGAINST VULNERABILITIES BASED ON RETURN-ORIENTED PROGRAMMING

Lubkin I., Zolotarev V. **Comprehensive Defense System against Vulnerabilities Based on Return-Oriented Programming.**

Abstract. It is difficult or impossible to develop software without included errors. Errors can lead to an abnormal order of machine code execution during data transmission to a program. Program splitting into routines causes possible attacks by using return instructions from these routines. Most of existing security tools need to apply program source codes to protect against such attacks. The proposed defensive method is intended to a comprehensive solution to the problem. Firstly, it makes it difficult for an attacker to gain control over program execution, and secondly, the number of program routines, which can be used during the attack, decreases. Specific security code insertion is used at the beginning and end of the routines to make it complicated to gain control over the program execution. The return address is kept secure during a call of the protected routine, and the protected routine is restored after its execution if it was damaged by the attacker. To reduce the number of suitable routines for attacks, it was suggested to use synonymous substitutions of instructions that contain dangerous values. It should be mentioned that proposed defensive measures do not affect the original application`s algorithm. To confirm the effectiveness of the described defensive method, software implementation and its testing were accomplished. Acknowledging controls were conducted using synthetic tests, performance tests and real programs. Results of testing have demonstrated the reliability of the proposed measures. It ensures the elimination of program routines suitable for attacks and ensures the impossibility of using standard return instructions for conducting attacks. Performance tests have shown a 14 % drop in the operating speed, which approximately matches the level of the nearest analogues. The application of the proposed solution declines the number of possible attack scenarios, and its applicability level is higher in comparison with analogues.

Keywords: vulnerability, remote code execution, code protection, RoP, code insertion.

Lubkin Ivan — Senior lecturer, Department of information technologies security, Reshetnev Siberian State University of Science and Technology. Research interests: security of operating systems, protection of program code from unauthorized use, countering vulnerabilities. The number of publications — 27. lubkin@rambler.ru; 48Б, newspaper Krasnoyarskiy rabochiy Ave., 660037, Krasnoyarsk, Russia; office phone: +7 (391) 222-76-39; fax: +7(391)222-76-39.

Zolotarev Vyacheslav — Ph.D., Associate Professor, Head of department, Department of information technologies security, Reshetnev Siberian State University of Science and Technology. Research interests: information security management, vulnerabilities protection. The number of publications — 98. zolotarev@sibsau.ru; 48Б, newspaper Krasnoyarskiy rabochiy Ave., 660037, Krasnoyarsk, Russia; office phone: +7(391)222-76-39.

Acknowledgements. The reported study was funded by Russian Ministry of Science (information security). Contract № 21/2020.

References

1. Glass, R. [Facts and misconceptions of professional programming] Fakty i zabluzhdenija professional'nogo programmirovanija // Saint Petersburg: Symbol-Plus. 2007. 240 p. (InRuss.)
2. Vishnjakov, A.V. [Classification of ROP-gadgets] Klassifikacija ROP-gadzhetov // Proceedings of the ISP RAS. 2016. Vol. 28. Issue 6, pp. 27–36. DOI: 10.15514/ISPRAS-2016-28(6)-2 (InRuss.)
3. Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity // Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19). Association for Computing Machinery, New York, NY, USA. 2019. Article 8, 1–11. DOI:https://doi.org/10.1145/3337167.3337175
4. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 // https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf
5. Intel Launches World's Best Processor for Thin-and-Light Laptops: 11th Gen Intel Core // https://www.intel.com/content/www/us/en/newsroom/news/11th-gen-tiger-lake-evo.html
6. RAP:RIP ROP 2015 // https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf
7. Koo, Z.Z. & Ayop, Zakiah & Abidin, Z.Z. Analysis of ROP attack on grsecurity / PaX linux kernel security variables // International Journal of Applied Engineering Research. 2017. no. 12. pp. 13179–13185.
8. Ivannikov V., Kurmangaleev Sh., Belevancev A., Nurmuhametov A., Savchenko V., Matevosjan R., Avetisjan A. [Implementation of confusing transformations in the computer infrastructure LLVM] Realizacija zaputyvajushhih preobrazovanij v kompiljatornoj infrastrukture LLVM // Proceedings of the ISP RAS. 2014. Vol. 26. Issue 1. pp. 327–342. (InRuss.)
9. Nurmuhametov A.R., Kurmangaleev Sh.F., Kaushan V.V., Gajsarjan S.S. [] Primenenie kompiljatornyh preobrazovanij dlja protivodejstvija jekspluatacii ujazvimostej programmnogo obespechenija // Proceedings of the ISP RAS. 2014. Vol. 26. Issue 3. pp. 113–126. DOI: 10.15514/ISPRAS-2014-26(3)-6. (InRuss.)
10. [ISP Obfuskator. Code obfuscation technology to protect against exploiting vulnerabilities] ISP Obfuskator. Tehnologija zaputyvanija koda dlja zashhity ot jekspluatacii ujazvimostej // https://www.ispras.ru /technologies/isp_obfuscator/ (InRuss.)
11. Nurmuhametov A.R., Zhabotinskij E.A., Kurmangaleev Sh.F., Gajsarjan S.S., Vishnjakov A.V. [Fine-grained randomization of the program's address space at startup] Melkogranuljarnaja randomizacija adresnogo prostranstva programmy pri zapuske // Proceedings of the ISP RAS. 2017. Vol. 29. Issue. 6. pp. 163–182. DOI: 10.15514/ISPRAS-2017-29(6)-9. (InRuss.)
12. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE. 2016.
13. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits // PoPETs. no. 4. 2016. pp. 454–469.
14. 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.
15. Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization // Proceedings of the 23rd Annual Network and Distributed System Security Symposium. 2016.

16. Onarlioglu K., Bilge L., Lanzi A., Balzarotti D., Kidra E. G-Free: Defeating return-oriented programming through gadget-less binaries // Proceedings of ACSAC: M. Franz and J. McDermott, Eds. ACM Press, 2010. pp. 49–58.
17. Jinku LI, Zhi WANG, Xuxian JIANG, Mike GRACE, and Sina BAHRAM. Defeating return-oriented rootkits with «return-less» kernels // Proceedings of EuroSys: Edited by G. Muller. ACM Press, 2010. pp. 195–208.
18. Dean Sullivan, Orlando Arias, David Gens, Lucas Davi, Ahmad-Reza Sadeghi, Yier Jin. Execution Integrity with In-Place Encryption // arXiv preprint arXiv:1703.02698. 2017.
19. Lubkin I.A., Subbotin N.A. Technique of verified program module modification with algorithm preservation // 11th International IEEE scientific and technical conference "Dynamics of systems, mechanisms and machines" (Dynamics): IEEE Xplore Digital Library, 2017. pp. 1–5.
20. Lubkin I.A., Bazhenov I.O. Methodology of software code decomposition analysis // Dynamics of systems, mechanisms and machines. Omsk, 2018. pp. 1–5.
21. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007 ACM Conference on Computer and Communications Security (CCS) // Proceedings of CCS. 2007. pp. 552–561.
22. Article Permutation conditions. URL: https://z0mbie.dreamhosters.com/pcond.txt (accessed 01.09.2021)
23. Sources of library eXtended Disassembler Engine (version 1.02). URL: https://github.com/nimrood/xde (accessed 01.09.2021)
24. AMD64 Architecture Processor Supplement Draft Version 0.99.7 // https://www.uclibc.org/docs/psABI-x86_64.pdf
25. Instrument ROPgadget. Source code repository. URL: https://github.com/JonathanSalwan/ROPgadget (accessed 01.09.2021).
26. Coremark. Benchmark software. URL: https://github.com/eembc/coremark (accessed 01.09.2021)