Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук

На правах рукописи

Федотов Андрей Николаевич

Разработка метода оценки эксплуатируемости программных дефектов

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат физико-математических наук, доцент
Падарян Вартан Андроникович

Москва — 2017

Оглавление

Стр.

Стр.

Введение

На сегодняшний день вопрос безопасности программного обеспечения представляет как практический, так и исследовательский интерес. Обеспечение безопасности программ непосредственно связано с поиском ошибок и уязвимостей. Ведущие научные институты, а также коммерческие компании занимаются исследованиями и разработками в области обнаружения ошибок и уязвимостей.

Уязвимости, позволяющие нарушителю выполнить произвольный код, представляют собой огромную опасность. Поиск такого типа уязвимостей – сложная, комплексная задача, которая требует непосредственного участия аналитика. Сложности заключаются не только в поиске состояния программы, при котором уязвимость проявляется, но и в оценке того, на какие аспекты безопасности она способна повлиять.

Достоверным способом проявления уязвимости является запуск программы на наборе входных данных (эксплойт), при котором уязвимость приводит к заданным нарушениям, таким, например, как выполнение заданного нарушителем кода или вызов библиотечной функции с контролируемыми нарушителем параметрами. Программные дефекты, лежащие в основе таких уязвимостей называются эксплуатируемыми дефектами. В настоящие время развиваются методы, позволяющие генерировать некоторые виды эксплойтов автоматически. Их разработкой активно занимаются зарубежные исследователи, в первую очередь - коллективы университетов Карнеги-Меллон, Беркли, Висконсина, Санта-Барбары, работы которых базируются на технологиях автоматической генерации тестовых наборов (фаззинг) и символьной интерпретации. Символьная интерпретация описывает процесс преобразования входных данных в программе в виде формул над символьными переменными и константами. Входные данные для программы выступают в качестве символьных переменных, которые способны принимать произвольные значения.

Современные средства фаззинга успешно применяются на практике и позволяют обнаружить множество аварийных завершений, порядка десятков тысяч для больших анализируемых программ. Набор входных данных, во время обработки которого происходит аварийное завершение программы, используется в качестве отправной точки при построении эксплойта. Символьная ин-

терпретация чрезвычайно требовательна к вычислительным ресурсам и не может применяться ко всему потоку аварийных завершений без предварительной фильтрации, но известные работы не уделяют этой важной проблеме должного внимания. Качественная фильтрация аварийных завершений позволит эффективнее проводить генерацию эксплойтов и в целом быстрее обнаружить эксплуатируемые дефекты.

Дальнейший анализ аварийного завершения изучает, на какие именно ячейки памяти и регистры процессора влияют входные данные в момент аварийного заверешения, и какие при этом ограничения накладываются на вход. В силу многообразия процессорных архитектур динамическому анализу на уровне бинарного кода требуется независимость от архитектуры процессора, на которым выполняется исследуемый код. Проблемой оказывается сложность формального описания условий срабатывания уязвимости. Формальное описание условий срабатывания уязвимости задается в рамках символьной интерпретации: условия, обеспечивающие эксплуатацию уязвимости, формируются в виде уравнений и неравенств над символьными переменными. В настоящий момент, известны способы формального описания условий срабатывания уязвимостей, которые могут приводить к выполнению произвольного кода из-за переполнения буфера на стеке или уязвимости форматной строки. Существующие решения автоматической генерации эксплойтов для таких типов уязвимостей позволяют без вмешательства аналитика получить работоспособный эксплойт. Тем не менее, эти решения обладают набором недостатков, в них присутствует ряд технических ограничений и упрощенная модель среды выполнения программ. Сильней всего сказывается игнорирование работы защитных механизмов, которые полностью или частично делают неработоспособными сгенерированные эксплойты в современных системах. Вследствие этого становится невозможно выделять из общей массы те аварийные завершения, которые по-настоящему несут опасность.

Таким образом, разработка метода, позволяющего проводить автоматическую генерацию эксплойтов по информации об аварийном завершении программы, с учётом особенностей современных операционных систем и процессорных архитектур, а также механизмов противодействия эксплуатации уязвимостей – является актуальной задачей. Применение автоматической генерации эксплойтов для оценки эксплуатируемости позволит разработчику своевременно обна-

руживать программные дефекты, которые необходимо исправлять в первую очередь.

Целью данной работы является исследование и разработка метода оценки эксплуатируемости программных дефектов по потоку аварийных завершений программы. Метод должен учитывать работу современных механизмов защит от эксплуатации уязвимостей, а также применяться к программам работающим под управлением ОС Linux и семейства ОС Windows.

Основные задачи:

1. Исследовать механизмы защиты от эксплуатации программных дефектов, реализованные в современных ОС общего назначения и размещаемые в исполняемом коде программ компиляторами.
2. Разработать архитектурно независимый метод автоматической генерации эксплойтов, позволяющих выполнять заданный аналитиком код. Входными данными для метода выступает информация об аварийном завершении программы. Метод должен учитывать влияние механизмов защиты на возможность эскалации последствий от срабатывания дефекта.
3. Разработать метод фильтрации аварийных завершений.
4. Разработать метод оценки эксплуатируемости программных дефектов, основанный на методе автоматической генерации эксплойтов и методе предварительной фильтрации аварийных завершений.
5. Разработать программные инструменты, реализующие метод автоматической генерации эксплойтов, метод предварительной фильтрации аварийных завершений и метод оценки эксплуатируемости программных дефектов. Оценить область применимости и эффективность предложенных методов.

Научная новизна:

1. Разработанный метод автоматической генерации эксплойтов позволяет проводить формирование эксплойтов с учётом механизма защиты от выполнения данных (DEP) и рандомизации адресного пространства процесса (ASLR). Метод применим к программам, работающим под управлением ОС Linux и семейства ОС Windows.
2. Разработанный метод предварительной фильтрации базируется на предложенной в работе классификации аварийных завершений и позволяет вырабатывать начальную оценку эксплуатируемости, учитывая

влияние встроенных компилятором механизмов защиты от переполнений буферов.

Практическая значимость работы состоит в том, что разработанная система оценки эксплуатируемости программных дефектов может встраиваться в системы непрерывной интеграции, что повышает безопасность программного обеспечения в промышленной разработке. Эффективность системы подтверждена на примере оценки эксплуатируемости дефектов, полученных в результате фаззинга. Система используется в различных научно-исследовательских и промышленных организациях.

Методология и методы исследования. Результаты диссертационной работы получены на базе использования методов динамического анализа бинарного кода и методов символьной интерпретации. Математическую основу исследования составляют теория алгоритмов, теория множеств и математическая логика.

Положения, выносимые на защиту:

1. Метод автоматической генерации эксплойтов по информации об аварийном завершении программы на основе символьной интерпретации трассы машинных команд с применением промежуточного представления Pivot. Метод учитывает работу механизмов защиты от эксплуатации уязвимостей DEP и ASLR, а также применим к программам, работающим под управлением ОС Linux и семейства ОС Windows.
2. Метод оценки эксплуатируемости программных дефектов, использующий автоматическую генерацию эксплойтов и предварительную фильтрацию аварийных завершений.
3. Программный инструмент, реализующий метод оценки эксплуатируемости программных дефектов.

Апробация работы. Основные результаты работы обсуждались на конференциях:

1. IV международный форум по практической безопасности «Positive Hack Days». Москва, 21-22 мая 2014.
2. 24-ая научно-техническая конференция «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 29 июня - 02 июля 2015.
3. Открытая конференция ИСП РАН. Москва, 1-2 декабря 2016.

Публикации. По теме диссертации опубликовано 5 научных работ, в том числе 4 научные статьи [1–4] в рецензируемых журналах, входящих в перечень рекомендованных ВАК РФ. Работа [3] индексирована в Scopus. В работах [1;3;5] представлен разработанный автором метод автоматической генерации эксплойтов. В статье [4] автором описаны улучшения метода автоматической генерации эксплойтов, позволяющие учитывать механизмы защиты от эксплуатации уязвимостей. В работе [2] представлен разработанный автором метод оценки эксплуатируемости программных дефектов.

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Объём и структура диссертации. Диссертация состоит из введения, четырёх глав, заключения и одного приложения. Полный объём диссертации составляет 98 страниц, включая 17 рисунков и 5 таблиц. Список литературы содержит 67 наименований.

В **первой главе** приводится обзор работ, которые имеют отношение к теме диссертации, а также рассматриваются современные защитные механизмы, препятствующие эксплуатации уязвимостей.

Вторая глава посвящена описанию предлагаемого подхода к оценке эксплуатируемости программных дефектов. Также представлены описания метода предварительной фильтрации аварийных завершений и метода автоматической генерации эксплойтов по аварийным завершениям.

В **третьей главе** рассматривается программная реализация методов, предложенных во второй главе.

В **четвёртой главе** приводятся результаты применения разработанных методов и инструментов.

В **заключении** содержатся выводы и направления дальнейшего развития разработанных методов и их программных реализаций.

Глава 1. Обзор работ

1.1 Механизмы противодействия эксплуатации уязвимостей

Современные механизмы защиты, препятствующие эксплуатации уязвимостей, можно разделить на две категории: защитные механизмы операционной системы и защитные механизмы, предоставляемые компилятором. К первой категории относятся рандомизация размещения адресного пространства (ASLR) и предотвращение выполнения данных (DEP). Во вторую категорию входят механизм размещения «канареек» и технология Fortify source, совмещающая легковесные проверки времени компиляции и автоматическую замену потенциально уязвимых функций на защищённые аналоги, а также защита обработчиков исключений в Windows (SafeSEH).

1.1.1 Рандомизация размещения адресного пространства (ASLR)

Рандомизация размещения адресного пространства, предполагает, что программа будет загружаться на различные адреса. В современных дистрибутивах операционных систем этот механизм успешно применяется. В операционных системах симейства Windows рандомизация поддерживается, начиная с Windows Vista. В системах на базе Linux поддержка ASLR имеется с 2005 года (с версии ядра 2.6.12).

Для затруднения эксплуатации ранадомизации подвергается размещение стека, кучи, динамически загружаемые библиотеки, а также адрес загрузки исполняемого файла. Размещение на произвольных адресах требует от разработчиков компилировать исполняемые файлы и в позиционно-независимый код. Следует отметить, что большинство реализаций ASLR вырабатывают случайную карту памяти для программы один раз на все время работы системы до следующей загрузки. Это означает, что перезапускаемые сервисы, падение которых не приводит к падению всей системы, будут достаточно долго работать на одной и той же карте памяти. В таком случае, утечка информации об адре-

сах работающей программы позволит настроить эксплойт таким образом, что он сможет преодолеть ASLR. Ситуация с утечкой адресов может возникнуть, при наличии уязвимости форматной строки [6].

Основываясь на анализе работы ALSR в 32-ух разрядной ОС на базе Linux, который приведён в статье [7], следует, что рандомизации подвергаются не все биты входящие в адрес. Например, в адресе вершины стека рандомизируются 28 младших битов. При рандомизации динамически загружаемых библиотек рандомизируются в адресе биты с 12 по 27. Таким образом возникает возможность обхода ASLR, используя частичную перезапись адреса, на который будет передано управление в будущем.

Одним из распространённых способов обхода ASLR является использование специальных инструкций «трамплинов» вида *call/jump $reg* [8]. Вместо того, чтобы передавать управление сразу на код полезной нагрузки, передача управления происходит на «трамплин», который уже в свою очередь передаёт управление на код полезной нагрузки. Для использования трамплинов необходимо выполнения двух условий:

– модуль, в котором находится «трамплин», не рандомизируется;
– значение регистра указывает в контролируемую область памяти.

В современных дистрибутивах Linux и Windows встречаются ситуации, когда некоторые модули не рандомизируются [4], В Linux довольно часто не рандомизируются образы исполняемых файлов, а в Windows некоторые библиотеки.

1.1.2 Предотвращение выполнения данных (DEP)

Предотвращение выполнения данных – механизм защиты, встроенный в различные операционные системы Windows, Linux и т.д., который запрещает программе выполнять код из области памяти, помеченной как «данные». Таким образом, стек и куча становятся недоступными для выполнения. При попытке выполнить код из этих регионов памяти, возникает исключение. В основе этой защиты лежит использование *NX бита (AMD)* или его аналога в процессорах от Intel *XD-бит* [9]. В свою очередь, чтобы использовать *NX бит* необходима поддержка процессором режима PAE. Благодаря *NX биту* появляется возможность помечать страницы памяти как не исполняемые ()*NX бит* равен 1), или

исполняемые (*NX бит* равен 0). Для работы защиты исполнения данных, помимо аппаратной поддержки, требуется поддержка со стороны операционной системы. В Linux поддержка появилась с версии ядра 2.6.8 (Август 2004). В Windows поддержка появилась, начиная с Windows XP Service Pack 2 (2004) и Windows Server 2003 Service Pack 1 (2005).

Одной из атак, использующейся для обхода этого защитного механизма является атака возврата в библиотеку [10]. Эта атака, как правило, применяется при эксплуатации уязвимости переполнения буфера на стеке. Адрес возврата из функции перезаписывается адресом библиотечной функции, и на стеке располагаются параметры для вызываемой функции. В Linux-подобных ОС наиболее распространённым является вызов функции *system* из библиотеки *libc*.

В современных дистрибутивах DEP и ASLR работают совместно, поэтому атака возврат в библиотеку становится затруднительной, ввиду того, адрес функции рандомизируется. В этом случае применяется подход под названием возвратно-ориентированное программирование (ROP) [11]. Код полезной нагрузки формируется в виде набора гаджетов. Гаджеты представляют собой набор инструкций из исполняемой области памяти, который заканчивается инструкцией передачи управления. Для осуществления этой атаки необходимо, чтобы гаджеты находились в нерандомизируемой и исполняемой области памяти. В настоящее время активно развиваются работы, связанные с автоматическим формированием кода полезной нагрузки в виде ROP-цепочек [12].

1.1.3 «Канарейка»

«Канарейка» – это специальное значение, которое размещено на стеке и разделяет собой пространство автоматических локальных переменных и служебные данные – адрес возврата и сохраненные регистры. Размещение «канарейки» выполняется в прологе функции, в эпилоге ее значение сравнивается с эталоном. Их различие интерпретируется как срабатывание ошибки переполнения буфера с угрозой порчи адреса возврата и других служебных данных. Программа в таком случае аварийно завершается, не доходя до более серьезных нарушений безопасности.

Значение «канарейки» либо заранее определено (состоит из терминальных символов), либо генерируется случайным образом. Кроме добавления «канарейки» происходит перестановка локальных переменных. Все указатели располагаются перед массивами (ниже по стеку), что препятствует их перезаписи. Стоит отметить, что перестановок полей в структурах не происходит. Применение «канарейки» сразу же показало эффективность данного метода защиты.

В тоже время, «канарейка» противодействует только одному способу эксплуатации переполнения буфера на стеке с перезаписью адреса возврата из функции [13]. Другие эксплуатируемые ситуации, такие как: уязвимость форматной строки [14; 15], ситуация, при которой атакующий контролирует адрес записи в память и записываемое значение [16], переполнение буфера на куче [17] и т.д..

1.1.4 Fortify source

Компилятор gcc, начиная с версии 4.0, поддерживает макрос FORTIFY_SOURCE, активирующий легковесные проверки на переполнение буфера в библиотечных функциях копирования: memcpy, strcpy, sprintf, gets и др. Некоторые проверки осуществляются во время компиляции, выдавая результат в виде предупреждений, остальные проверки происходят во время выполнения и в случае срабатывания аварийно завершают программу. Для проверок во время выполнения, функции заменяются безопасными аналогами, которые и производят необходимые проверки. Замена функций копирования происходят только тогда, когда известны размеры буфера-приёмника на стадии компиляции. Функции, работающие с форматной строкой, заменяются аналогами, которые проверяют использование формата %n, отслеживают использование не литеральных форматов и др. Наличие таких проверок нацелено на предотвращение целенаправленной перезаписи адреса возврата, не вызывающей порчи «канарейки».

1.1.5 Защита времени загрузки

Существует статическое и динамическое связывание библиотек и исполняемого файла.При динамическом связывании код библиотек не прикрепляется к исполняемому файлу в время связывания. Вызов процедур в ОС Linux осуществляется через глобальную таблицу смещений (GOT), в которой хранятся адреса вызываемых функций. При определении адреса процедуры в библиотеках довольно часто используется «ленивое связывание». При таком способе адреса конкретной процедуры не вычисляются до тех пор, пока не производится её вызов [18; 19].

Существуют способы эксплуатации, при которых атакующий перезаписывает запись в глобальной таблице смещений. Таким образом, при вызове функций через перезаписанный GOT-слот, произойдёт передача управления на код полезной нагрузки [20]. В качестве защиты от такой атаки используется непосредственное разрешение всех адресов вызываемых функций с последующим запретом секции .got на запись. Для применения непосредственного связывания в компиляторе *gcc* используется опция *-znow*. Кроме того, непосредственное связывание можно применять при каждом запуске процесса в системе, установив значение переменной окружения LD_BIND_NOW отличное от нуля.

1.1.6 Защита обработчиков исключений в Windows (SafeSEH)

Обработчиком исключений является фрагмент кода программы, предназначенный для реакции программы на возникшее исключение. Помимо пользовательских обработчиков исключений операционные системы Windows предоставляют обработчик исключения по умолчанию. В Windows используется структурированная обработка исключений. На стеке размещается структура, состоящая из двух полей. Первое поле содержит адрес следующей структуры обработчика исключений. Второе поле указывает на код обработчика исключений. Структуры для пользовательских обработчиков размещаются в кадре стека функции, в котором находится код обработчика исключений. Таким образом, структуры представляют собой односвязный список. Атакующий может

перезаписать второе поле в структуре обработчика исключений и при возникновении исключения произойдёт передача на код полезной нагрузки. Начиная с Windows XP SP1, перед выполнением кода обработчика исключения обнуляются все регистры. Такая ситуация приводит к тому, что использование «трамплинов» становиться не возможным. Также существует другой способ эксплуатации, когда перезаписывается первое поле структуры. В прологе обработчика исключений по смещению +8 (x86) от указателя стека размещается адрес следующей структуры. Передав управление на последовательность инструкций *pop pop ret*, атакующий затем перенаправит управление на код полезной нагрузки [21].

В большинстве случаев перезапись структур обработчиков исключений происходит при переполнении буфера на стеке. Вместе с этим, как правило, осуществляется перезапись адреса возврата из функции. Эксплуатация переполнения буфера на стеке через перезапись адреса возврата проще в осуществлении. Эксплуатация этой уязвимости через обработчик исключения даёт возможность обойти «какнарейку». Это достигается, благодаря тому, что передача управления на код полезной нагрузки происходит до сравнения «какнарейки» с эталонным значением, путём вызова исключения.

В качестве защиты от эксплуатации структур обработчиков исключений компанией Microsoft был разработан метод защиты SafeSEH, который применяется с помощью опции компилятора /SAFESEH [22]. Во время компиляции все адреса обработчиков исключений сохраняются в отдельной таблице в исполняемом файле. Перед тем как передать управление на обработчик при возникновении исключения, адрес обработчика сравнивается с адресами из таблицы. Если адрес совпадает с одним из адресов в таблице, управление будет передано на обработчик, в противном случае передачи управления не происходит. Такой способ позволяет надёжно защититься от атак на структуры обработчиков исключений. Для полноценной защиты процесса необходимо, чтобы исполняемый файл и все используемые библиотеки были собраны с опцией /SAFESEH. Если хотя бы одна библиотека или исполняемый файл собран без этой опции, то существует возможность применения атаки на структуры обработчика исключений.

1.1.7 Анализ современных дистрибутивов ОС Linux на предмет защищенности исполняемых файлов

Анализ на предмет защищенности проводился для исполняемых файлов из директории */usr/bin/* в распространенных дистрибутивах ОС Linux. Для анализа использовалась утилита *hardening-check* из пакета *hardening-includes* [23]. Данная утилита позволяет проверить, собран ли исполняемый файл в позиционно-независимом коде (ПНК), используются ли безопасные функции Fortify Source и «канарейка», защищена ли секция .got от записи, происходит ли связывание непосредственно в момент загрузки. В табл. 1 приведены результаты анализа некоторых современных дистрибутивов Linux, позволяющие сделать несколько выводов. Анализ дистрибутивов Linux показал, что в большинстве современных дистрибутивах присутствуют такие защитные механизмы, как: ASLR, DEP, «канарейка» и FORTIFY_SOURCE. Защитный компиляторный механизм FORTIFY_SOURCE фактически исправляет некоторые ошибки, совершаемые разработчиком, тем самым устраняя потенциальные возможности эксплуатации уязвимостей, основанных на этих дефектах.«Канарейка» позволяет предотвратить эксплуатацию уязвимости переполнения буфера на стеке с перезаписью адреса возврата из функции. Другие способы эксплуатации остаются доступны для атакующего.

Подавляющие большинство (порядка 85%) исполняемых файлов собирается не в виде позиционно-независимого кода, что позволяет их использовать для поиска гаджетов, и, таким образом, применять методы обхода DEP и ASLR. При этом следует признать, что не изучалось, какие именно программы остались позиционно зависимыми и насколько критична их компрометация с точки зрения безопасности всей системы. С другой стороны доля программ с позиционно зависимым кодом не меняется в течение последних двух лет, их одинаково много как в 32-х разрядных, так и в 64-х разрядных системах.

Во многих программах используется ленивое связывание, что оставляет возможность перезаписи .got-слотов с целью передачи управления на код полезной нагрузки.

Таким образом, предлагаемые в работе методы эксплуатации уязвимостей сосредоточены на попытках преодоления таких защитных механизмов, как DEP и ASLR.

Таблица 1 — Результаты анализа содержимого /usr/bin некоторых дистрибутивов Linux.

Название	Дата выпуска	Кол-во исп. файлов	Без ПНК	Без «канарейки»	Без Fortify Source	Ленивое связывание
Debian 6.0.10 (32 бита)	19.07.14	4705	4613 / 98%	4617 / 98%	3899 / 83%	4639 / 99%
Debian 8.3.0 (32 бита)	23.01.16	387	311 / 80%	81 / 23%	70 / 20%	311 / 80%
Arch (32 бита)	25.05.16	2846	2671 / 94%	383 / 13%	493 / 17%	2717 / 95%
Ubuntu 14.10 (32 бита)	23.10.14	1162	1016 / 87%	242 / 21%	96 / 8%	1036 / 89%
Ubuntu 14.04.1 (64 бита)	24.07.14	851	712 / 84%	67 / 8%	48 / 6%	709 / 83%
Ubuntu 16.04.1 (64 бита)	21.07.16	1053	891 / 85%	211 / 20%	81 / 8%	881 / 84%

1.2 Средства оценки эксплуатируемости программных дефектов

В разделе рассматриваются работы, в которых решаются задачи, схожие с задачей оценки эксплуатируемости программных дефектов. Существующие инструменты можно отнести к двум группам. К первой группе относятся средства анализа аварийных завершений программ: !exploitable [24], gdb exploitable plugin [25], CrashFilter [26]. Во вторую группу входят инструменты, обеспечивающие автоматическую генерацию эксплойтов: AEG [27; 28], MAYHEM [29], CRAX [30] и REX [31].

1.2.1 Средства анализа аварийных завершений

Инструмент !exploitable

Во время анализа аварийных завершений инструмент !exploitable [24] исследует состояние программы (значения регистров и памяти, стек вызовов и т.д.) в момент срабатывания исключения. К этому состоянию применяется набор правил, каждое из них описывает определённый класс аварийных завершений. Класс аварийных завершений, в свою очередь, представляет собой описание ситуации, из-за которой возникло аварийное завершение. Например, чтение из памяти по нулевому указателю, деление на ноль и т.д.. Для проведения анализа в инструменте используется отладчик Windbg [32], из чего следует, что анализируются программы, работающие под ОС Windows. На рис. 1.1 представлено описание правила для определения нарушения доступа во время инструкции чтения из памяти. Поля EXCEPTION_ADDRESS_RANGE,

```
// Rule: AVs at the instruction pointer are exploitable if not near null in user mode
BEGIN_ANALYZE_DATA
    PROCESSOR_MODE              USER
    EXCEPTION_ADDRESS_RANGE     NOT_NEAR_NULL
    EXCEPTION_TYPE              STATUS_ACCESS_VIOLATION
    EXCEPTION_SUBTYPE           ACCESS_VIOLATION_TYPE_READ
    EXCEPTION_LEVEL             DONT_CARE
    ANALYZE_FUNCTION            IsFaultingAddressInstructionPointer
    RESULT_CLASSIFICATION       EXPLOITABLE
    RESULT_DESCRIPTION          L"Read Access Violation at the Instruction Pointer"
    RESULT_SHORTDESCRIPTION     L"ReadAVonIP"
    RESULT_EXPLANATION          L"Access violations at the instruction pointer are exp
    RESULT_URL                  NULL
    RESULT_IS_FINAL             true
END_ANALYZE_DATA
```

Рисунок 1.1 — Описание правила в инструменте !exploitable.

EXCEPTION_TYPE, EXCEPTION_SUBTYPE формируют описание класса аварийного завершения. Поле RESULT_CLASSIFICATION определяет принадлежность класса аварийного завершения к группе. В инструменте поддерживается 4 группы аварийных завершений:

– эксплуатируемые аварийные завершения. Аварийные завершения, эксплуатация которых наиболее вереоятна.

- возможно эксплуатируемые аварийные завершения. Эксплуатация таких аварийных завершений потенциально возможна.
- вероятно неэксплуатируемые аварийные завершения. Для этих аварийных завершений провести экслпуатацию с большой вероятностью невозможно.
- аварийные завершения, экслпуатируемость которых не была установлена в ходе анализа.

Поля RESULT_DESCRIPTION, RESULT_SHORTDESCRIPTION, RESULT_EXPLANATION формируют текстовое описание класса аварийного завершения. Стоит отметить, что инструмент позволяет анализировать не только пользовательские программы, но и программы уровня ядра. При анализе используется промежуточное представление, которое позволяет поддержать анализ нескольких процессорных архитектур:x86, x86_64, arm. Также в инструменте используется анализ помеченных [33]. Анализ помеченных данных реализован в очень упрощённом виде. Проводится прямой анализ от инструкции, в результате которой произошло исключение. Критерием для анализа являются входные операнды инструкции. Анализ продолжается до достижения конца базового блока. Если в результате анализа помеченное значение используется для передачи управления, то аварийное завершение считается эксплуатируемым. В связи с тем, что размер базового блока, как правило, небольшой, результаты такого анализа зачастую не дают положительного результата . В таблице 2 представлено количество классов аварийных завершений, разбитых по группам, которые поддерживает инструмент. Стоит отметить, что большая часть аварийных завершений приходится на экслпуатируемые и возможно эксплуатируемые завершения (26 классов). Исходные коды инструмента находятся в свободном доступе [34].

Gdb exploitable plugin

В основе инструмента лежит тоже подход, что и в !exploitable. В отличие от !exploitable, gdb exploitable plugin [25] использует отладчик gdb [35] и анализирует программы, работающие под управлением ОС Linux. В таблице 3 представлено количество классов аварийных завершений, разбитых по группам, которые

Таблица 2 — Количество классов аварийных завершений по группам в инструменте !exploitable.

Группы классов аварийных завершений	Количество
Эксплуатируемые аварийные завершения	17
Возможно эксплуатируемые аварийные завершения	9
Возможно неэксплуатируемые аварийные завершения	6
Аварийные завершения, эксплуатируемость которых не была установлена	13
Всего классов аварийных завершений:	45

поддерживает gdb exploitable plugin. Группы аварийных завершений совпадают

Таблица 3 — Количество классов аварийных завершений по группам в инструменте gdb exploitable plugin.

Группы классов аварийных завершений	Количество
Эксплуатируемые аварийные завершения	10
Возможно эксплуатируемые аварийные завершения	5
Возможно неэксплуатируемые аварийные завершения	3
Аварийные завершения, эксплуатируемость которых не была установлена	4
Всего классов аварийных завершений:	22

с группами в инструменте !exploitable. Инструмент поддерживает меньшее количество классов аварийных завершений, чем !exploitable. Большинство классов аналогичны классам !exploitable. Остальные классы описывают исключительные ситуации, специфичные для Linux-систем. Анализ производится для пользовательских программ. У инструмента отсутствует внутреннее представление, но поддерживаются архитектуры: x86, x86_64, arm. На рис. 1.2 представлено описание правила для класса аварийного завершения, соответствующие исключению при выполнении инструкции возврата x86/x86_64. В описании правила фигурирует принадлежность класса к определённой группе, имя функции, проверяющей выполнение правила, а также текстовое описание класса аварийного

```
('EXPLOITABLE', [
    dict(match_function="isReturnAv",
    desc="Access violation during return instruction",
    short_desc="ReturnAv",
    explanation="The target crashed on a return instruction, which likely "
    "indicates stack corruption."),
```

Рисунок 1.2 — Описание правила в инструменте gdb exploitable plugin.

завершения. Инструмент написан на языке python и находится в свободном доступе.

CrashFilter

CrashFilter [26] позволяет проводить оценку эксплуатируемости аварийных завершений для исполняемых файлов, работающих на архитектуре ARM. Подход к анализу схож с подходом, применяемым в инструменте !exploitable. Основным преимуществом, по сравнению с !exploitable, является улучшенный анализ помеченных данных. Также как и в !exploitable, анализ помеченных данных начинается от инструкции, выполнение, которой вызвало исключение. Ключевая особенность в том, что анализ проводится статически, а не динамически, и не ограничивается одним базовым блоком. Аварийное завершение, считается эксплуатируемым, если помеченные данные используются в качестве адреса для передачи управления. Также, если при выполнении инструкции записи в память, значение и адрес, по которому происходит запись, оказываются помеченными, то аварийное завершение, считается эксплуатируемым. Для реализации анализа помеченных данных используется промежуточное представление REIL [36]. Инструмент реализован в виде модуля-расширения для платформы анализа бинарных файлов [37]. Исходные коды инструмента находятся в открытом доступе [38].

1.2.2 Средства автоматической генерации эксплойтов

AEG

Одна из первых систем, продемонстрировавшая возможность автоматической генерации эксплойтов для реальных программ, была система AEG [27; 28]. Реализованный в системе подход к автоматической генерации эксплойтов базируется на динамическом символьном выполнении [39] и подходах к описанию предикатов безопасности [40]. Анализу подвергается не только исполняемый файл, но и исходные тексты программы. С помощью анализа исходных текстов происходит поиск ошибки в программе, например, переполнения буфера на стеке. На рис. 1.3 представлена схема работы инструмента AEG. Анализ ограничивается исполняемыми файлами для архитектуры x86. Исходя из при-

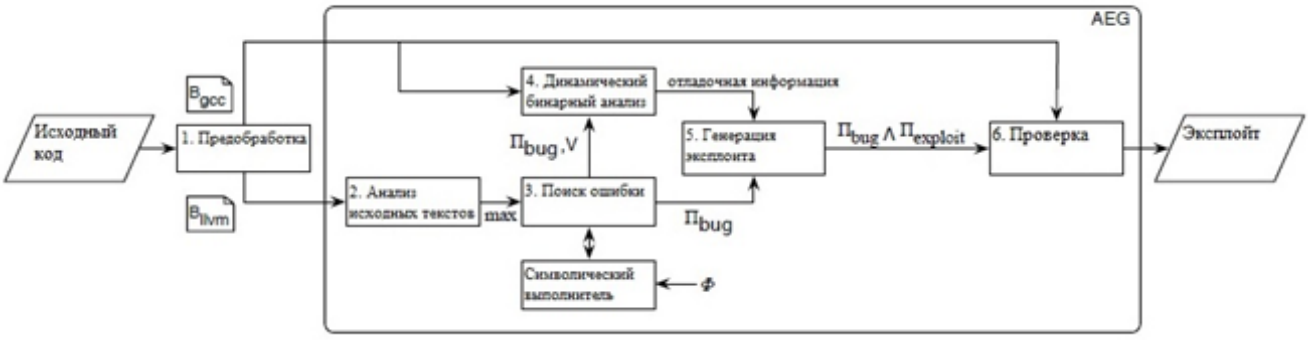

Рисунок 1.3 — Схема работы инструмента AEG.

ведённой выше схемы, процесс формирования эксплойтов можно поделить на несколько стадий:

- предобработка;
- анализ исходного кода;
- поиск дефекта;
- динамическое символьное выполнение;
- генерация эксплойта;
- проверка работоспособности.

На стадии *предобработки* происходит получение бит-кода LLVM [41] и компиляция исполняемого файла.

При *анализе исходных текстов* определяется размер символьных данных для последующих стадий. Для этого используется следующая эвристика: к максимальному размеру буфера, выделенного в программе добавляется 10% от его размера. Под рассмотрение попадают буферы, размеры которых известны на этой стадии.

На стадии *поиска ошибки* происходит анализ бит-кода LLVM с применением разработанных критериев для обнаружения дефекта. Помимо этого строится предикат пути до места возникновения ошибки. Данные, полученные в результате решения предиката пути будут входными данными для программы на следующей стадии.

Динамическое символьное выполнение используется для сбора необходимой информации для генерации эксплойта: адреса буферов в памяти, адрес на стеке, в котором размещён адрес возврата из функции.

Генерация эксплойтов поддерживается для двух типов уязвимостей: уязвимость переполнения буфера на стеке и уязвимость форматной строки. На этой стадии формируется предикат безопасности, который объединяется с предикатом пути, и после чего происходит решение полученной системы. В случае успеха, полученная подстановка является эксплойтом. На стадии *проверки* происходит запуск программы с эксплойтом в качестве входных данных.

В качестве результатов авторы приводят успешную генерацию эксплойтов для 16 уязвимостей, две из которых являлись уязвимостями нулевого дня. Система находится в закрытом доступе. Из публикаций, касательно этой системы, невозможно однозначно сделать вывод о том, что полученные эксплойты работоспособны в современных операционных системах при включённых защитах DEP и ASLR. В публикации по теме ROP-компиляции [12] этого же коллектива, есть упоминания об инструменте AEG. Исходя из этого, можно предположить, что в системы AEG применяются методы, позволяющие генерировать эксплойты, которые способны обходить DEP и ASLR.

Mayhem

Система Mayhem [29] является развитием системы AEG. Mayhem осуществляет поиск дефектов с последующей генерацией эксплойта. Как и в AEG

генерация экслойтов происходит в случае, когда счётчик инструкций становиться символьным, а также уязвимости форматной строки. Основной отличительной особенностью инструмента от AEG, является то, что Mayhem не требует для анализа исходных текстов программы, весь анализ происходит исключительно по бинарным файлам для архитектуры x86. В проекте Mayhem представлен метод гибридного символьного выполнения. В начале анализа используется так называемый «онлайн» подход к символьному выполнению. В этом подходе реальное выполнение программы происходит одновременно с символьным выполнением. Система символьного выполнения S2E [42] придерживается такого способа анализа. Из-за экспоненциального роста числа путей выполнения значительно возрастают накладные расходы, связанные хранением в памяти символьных состояний. Альтернативой «онлайн» подходу является «оффлайн» анализ. В случае «оффлайн» анализа символьное выполнение происходит про трассе, полученной в результате реального выполнения. Минусом этого подхода является необходимость строить предикат пути с начала выполнения программы для каждого нового пути выполнения. Как только Mayhem обнаруживает нехватку памяти, система переходит «оффлайн» режим выполнения. В случае, если ресурсы освободятся, то система может вернуться к «онлайн» режиму.

Также одним из достижений Mayhem является метод работы с символьными адресами памяти. В случае попадания символьного значения в адресное выражение, необходимо понять, к какому адресу памяти происходит доступ. Очевидным и наиболее частым решением является конкретизация значения адреса. Для записи по символьному адресу в Mayhem используется конкретизация. При чтении по символьному адресу система пытается предоставить все возможные варианты допустимых значений адреса.

В качестве результатов авторы приводят успешную генерацию эксплойтов для 29 уязвимостей. Система находится в закрытом доступе.

CRAX

Система CRAX [30] позволяет автоматически генерировать эксплойты для аварийных завершений. В основе CRAX лежит система S2E [42], позволяющая проводить полносистемное символьное выполнение. Анализу подвергаются ис-

полняемые файлы для архитектуры x86. В работе приводятся различные случаи, при которых осуществляется генерация эксплойтов:

- символьное значение оказалось в счётчике инструкций (EIP).
- символьные значения оказались в адресе памяти и в записываемом значении во время выполнения инструкции записи (CWE-123).
- символьное значение в форматной строке во время вызова функции работы с форматной строкой.

Также применяется некоторые способы обхода защитных механизмов. В случае, если символьное значение оказалось в счётчике инструкций после выполнения инструкции RET, то для обхода DEP используется атака типа возврат в библиотеку. Для обхода ASLR применяются трамплины. При генерации эксплойта используется классический подход. С момента получения входных данных и до точки аварийного завершения происходит построение предиката пути, а на инструкции аварийного завершения происходит построение предиката безопасности. В случае эксплуатации уязвимости форматной строки предикат безопасности строиться до вызова функции работы с форматной строкой, во время которого в форматной строке содержаться символьные переменные.

В качестве результатов авторы приводят успешную генерацию эксплойтов для 33 уязвимостей, большинство из которых совпадают со списком уязвимостей, представленным в Mayhem.

Открытая версия системы [43] обладает значительными ограничениями, по сравнению с заявленными результатами в публикации. В открытой версии поддерживается генерация эксплойтов только для ситуаций, при которых символьное значение оказалось в счётчике инструкций, защитные механизмы при этом никак не учитываются. Результатом работы инструмента является единственный файл с эксплойтом, Из этого следует, что система не поддерживает получение входных данных из нескольких файлов или несколько различных источников входных данных.

REX

Система REX [31] входит в состав платформы анализа бинарного кода angr и позволяет генерировать эксплойты для аварийных завершений. Анализу

подвергаются исполняемые файлы для архитектуры x86, работающие под операционными системами семейства Linux и специально разработанной ОС (cgc) для DARPA Cyber Grand Challenge на базе Linux. Система находится в открытом доступе [44].

Исходя из анализа исходных текстов системы можно сделать следующие выводы. В самом начале происходит анализ аварийного завершения и определяется возможность его эксплуатации. В качестве эксплуатируемых аварийных завершений считаются аварийные завершения, в результате которых счётчик команд содержит символьное значение и завершения, при которых символьные значения оказались в адресе памяти и в записываемом значении во время выполнения инструкции записи (CWE-123). Причём эксплуатация CWE-123 происходит для программ, работающих под управлением ОС cgc. Эксплуатация ситуаций с символьным значением в счётчике команд может проходить с учётом работы защитных механизмов DEP и ASLR. Код полезной нагрузки в этом случае представляет собой ROP-цепочку.

1.3 Выводы

В современных операционных системах уделяют достаточно большое внимание обеспечению безопасности.В результате анализа современных дистрибутивов операционных систем семейства Linux было установлено, что в большинстве дистрибутивов присутствуют такие защитные механизмы, как DEP, ASLR, «канарейка» и Fortify source. Защитный компиляторный механизм Fortify source фактически исправляет некоторые ошибки, совершаемые разработчиком, тем самым устраняя потенциальную возможность для эксплуатации уязвимостей, основанных на этих дефектах. «Канарейка» позволяет предотвратить эксплуатацию уязвимости переполнения буфера на стеке с перезаписью адреса возврата из функции. Другие способы эксплуатации остаются доступны для атакующего.Таким образом, предлагаемый в данной работе метод автоматической генерации эксплойтов сосредоточен на преодолении таких защитных механизмов, как: DEP и ASLR.

Из проведённого обзора средств оценки эксплуатируемости программных дефектов вытекают следующие выводы. Большинство рассмотренных инстру-

ментов могут быть использованы для применены для решения поставленных задач. Вместе с тем, известные подходы обладают следующими существенными ограничениями.

Подход на основе анализа аварийных завершений позволяет с высокой точностью определить неэксплуатируемые аварийные завершения. Ещё одним достоинством стоит отметить относительную простоту анализа, и, как следствие, высокую скорость работы. Существенным ограничением подхода является отсутствие эксплойта для аварийных завершений, которые были классифицированы как эксплуатируемые. В связи с этим, невозможно однозначно утверждать об эксплуатируемости дефекта.

Методы автоматической генерации эксплойтов позволяют с высокой вероятностью получить работоспособный эксплойт. Автоматическая генерация эксплойтов основывается на использовании динамического символьного выполнения, поэтому ей свойственна такая проблема как рост числа символьных уравнений, что, как следствие, ведёт к увеличению времени решения системы, а значит растёт и время генерации эксплойта. Кроме того, существующие инструменты имеют некоторые ограничения. Все системы анализируют исполняемые файлы программ для архитектуры x86. Тем самым отсутствует возможность проведения анализа для других распространённых процессорных архитектур таких как x86_64, MIPS, PPC, ARM. Инструменты AEG и MAYHEM работают только с пользовательскими программами. Также эти инструменты находятся в закрытом доступе, а в публикациях нет информации о том, как учитывается работа современных защитных механизмов при генерации эксплойта. Открытая Система CRAX, основанная на полносистемном символьном выполнении, представленном в системе S2E, позволяет генерировать эксплойты только для ситуаций, когда символьное значение оказалось в счётчике инструкций, при этом работа защитных механизмов никак не учитывается. Система REX даёт возможность генерировать эксплойты для ситуации, когда символьное значение оказалось в счётчике инструкций для программ, работающих под управлением операционной системы Linux с учётом работы DEP и ASLR.

Таким образом, существующие системы не удовлетворяют выдвинутым требованиям и требуют значительной доработки.

В настоящей работе далее описан подход к решению задачи оценки эксплуатируемости программных дефектов, разработанные в рамках которого методы

в большинстве случаев позволяют успешно преодолевать выявленные ограничения.

Глава 2. Оценка эксплуатируемости программных дефектов

Схема метода оценки эксплуатируемости программных дефектов представлена на рис. 2.1. Метод состоит из трёх последовательных этапов. Начальными параметрами для первого этапа являются входные данные, приводящие к аварийному завершению программы, которые, например, могут быть получены в результате фаззинга, динамического символьного выполнения или путём анализа систем отслеживания ошибок.

На первом этапе (раздел 2.1) выполняется предварительная фильтрация аварийных завершений. Основная цель предварительной фильтрации – отсеять аварийные завершения, эксплуатация которых наименее вероятна. Уже на этапе анализа аварийного завершения можно охарактеризовать некоторые аварийные завершения, как неэксплуатируемые. Такие аварийные завершения являются, например следствием, разыменования нулевого указателя или деления на ноль. Дальнейших анализ этих аварийных завершений не имеет смысла, и позволяет сократить общие время анализа.

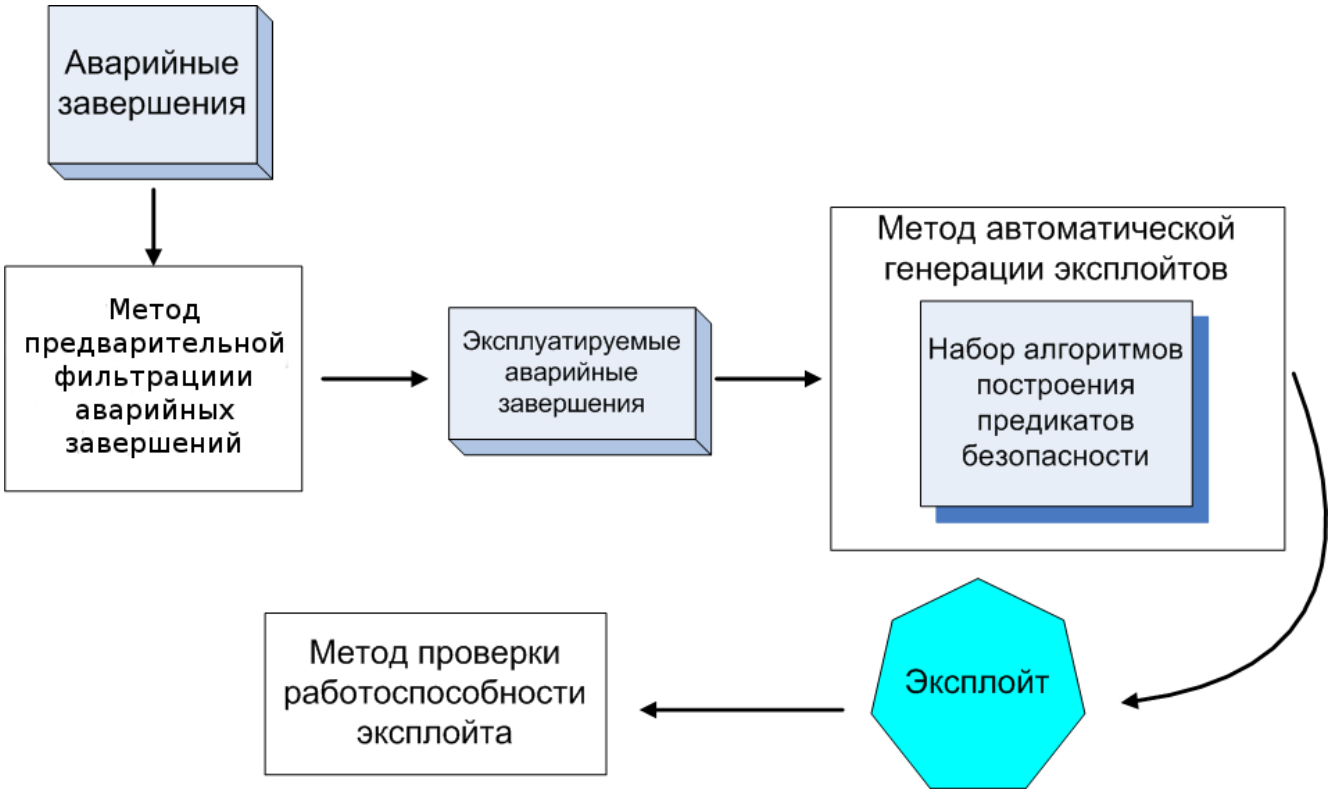

Рисунок 2.1 — Схема метода оценки эксплуатируемости программных дефектов

В случае, если аварийное завершение классифицируется как эксплуатируемое, начинается второй этап – автоматическая генерация эксплойта (раздел 2.2). Предлагаемый подход к автоматической генерации эксплойта основан на динамическом символьном выполнении трасс, полученных с помощью полносистемной эмуляции. Трасса снимается со сценария работы программы, при котором происходит аварийное завершение. По трассе выполнения от точек получения входных данных и до точки аварийного завершения программы происходит построение предиката пути (раздел 2.3). Предикат пути представляет собой набор символьных уравнений и неравенств, описывающий путь выполнения в программе, приведший к аварийному завершению. Неотъемлемым шагом при генерации эксплойта является построение предиката безопасности (раздел 2.4). Предикат безопасности описывает состояние программы, при котором достигается выполнение заданного кода полезной нагрузки. При генерации эксплойта учитывается набор защитных механизмов, который может работать на целевой системе.Если генерация эксплойта оказывается успешной, то стартует третий этап – проверка работоспособности полученного эксплойта. На этом этапе программа запускается в эмуляторе и в качестве входных данных ей подаётся сгенерированный эксплойт. Затем происходит мониторинг работы программы на предмет ожидаемого поведения. Это ожидаемое поведение является выполнением полезной нагрузки эксплойта. Примером такого поведения может выступать запись определённого значения в последовательный порт. Если во время мониторинга удалось обнаружить ожидаемое поведение программы, то эксплойт считается работоспособным, а дефект считается эксплуатируемым.

2.1 Метод предварительной фильтрации аварийных завершений.

В основе метода предварительной фильтрации аварийных завершений лежит анализ аварийных завершений. Аварийные завершения программ возникают в следствии срабатывания дефекта в программе, который при этом порождает исключение. Исключения можно разделить на два типа:

– программные исключения;

– аппаратные исключения.

В некоторых языках программирования (C++, Java и т.д.) существует возможность установки собственных обработчиков исключений. Они предназначены для описания реакции на различного рода ошибки времени выполнения и на другие нештатные ситуации, возникающие при работе программы. Как правило, в языке присутствует специальный оператор для генерации исключения (throw или raise). В этом случае исключение представляет собой объект данных. Таким образом, для работы с исключением необходимо создать объект исключения и затем вызвать исключительную ситуацию с этим объектом в качестве параметра. После чего произойдёт вызов соответствующего обработчика для этого типа исключения. Успешно обработав исключение программа может продолжить свою работу.

Аппаратные исключения непосредственно процессором. Такие исключения могут возникнуть, например, при попытке обращения к памяти, доступ к которой запрещён. Примером такой ситуации в программе может выступать обращение по неинициализированному (нулевому) указателю, или по указателю, значение которого было испорчено в результате переполнения буфера. Именно аппаратные исключения представляют наибольшую опасность. В определённых условиях атакующий может получить контроль над потоком выполнения программы и выполнить свой код. Аппаратные исключения обрабатываются ядром операционной системы. В ОС на базе Linux для уведомления процессов о возникших исключениях используются сигналы. При получении сигнала вызывается соответствующий обработчик. У процесса может быть установлен свой обработчик сигнала, в противном случае вызывается системный обработчик, который для большинства сигналов завершает выполнение процесса. Помимо номера сигнала обработчик может получать и дополнительную информацию, которая будет полезна при обработке сигнала. Эта информация описывается структурой *siginfo_t*, которая представлена на рис. 2.2. Первое поле структуры – номер сигнала. Сигнал под номером 8 (SIGFPE) возникает при ошибке во время арифметической операции. Сигнал под номером 11 (SIGSEGV) возникает при ошибке доступа к памяти. Также полезную информацию несёт собой поле *si_addr*. В случае сигнала SIGFPE это поле указывает на адрес инструкции, на которой возникло исключение. В случае сигнала SIGSEGV в поле *si_addr* содержится значение адреса, из-за обращения к которому возникло исключение. Ещё одним интересным сигналом, на который стоит обратить внимание при анализе аварийных завершений, является сигнал под номером 6 (SIGABRT). Этот сиг-

```
typedef struct {
    int si_signo;
    int si_code;
    union sigval si_value;
    int si_errno;
    pid_t si_pid;
    uid_t si_uid;
    void *si_addr;
    int si_status;
    int si_band;
} siginfo_t;
```

Рисунок 2.2 — Структура для описания информации о сигнале

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD  *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Рисунок 2.3 — Структура для описания исключения в Windows

нал процесс посылает сам себе для завершения работы. Он часто используется библиотеками (*ibc*) при обнаружении ошибочных ситуаций. Например, при срабатывании канарейки, безопасной функции работы со строками или ошибками при работе с менеджером памяти.

Как и Linux операционная система Windows занимается обработкой аппаратных исключений. Структура, описывающая информацию об исключении в Windows приведена на рис. 2.3.

Первое поле описывает код исключения. Исключение EXCEPTION_ACCESS_VIOLATION возникает при ошибке доступа к памяти, а исключение с типом EXCEPTION_INT_DIVIDE_BY_ZERO появляется при целочисленном делении на ноль. Поле *ExceptionAddress* указывает на адрес инструкции, на которой возникло исключение. В массиве *ExceptionInformation* храниться дополнительная информация об исключении, например, для исключения с кодом

EXCEPTION_ACCESS_VIOLATION храниться адрес памяти, к которой была попытка доступа, а также флаг с характером доступа (чтение\запись).

Легко видеть, что структуры для описания исключений в Linux и Windows достаточно схожи и содержат необходимую информацию. Для того, чтобы оценить возможность эксплуатации аварийного завершения этой информации недостаточно и требуется провести дополнительный анализ.

2.1.1 Классы аварийных завершений

Сперва опишем ситуации, при которых эксплуатация аварийного завершения потенциально возможна.

Первой ситуацией может выступать нарушение доступа при попытке выполнить текущую инструкцию. Это означает, что произошла передача управления по адресу, доступ к которому запрещён. Атакующий потенциально может указать адрес кода, где располагается полезная нагрузка, что в последствии приведёт к выполнению этого кода. Проверить наличие описанной ситуации достаточно просто: необходимо в момент возникновения исключения проверить доступ на чтение памяти, адрес которой указан в счётчике инструкций. Если память не доступна, то это указывает на возникновение приведённой выше ситуации.

Второй случай – исключение при попытке выполнить инструкцию вызова или безусловной передачи управления. В архитектуре x86\x86_64 операндом инструкции вызова или безусловной передачи управления может выступать ячейка памяти. Пример такой инструкции: *CALL DWORD PTR[EAX]*. Исключение возникает, если память по адресу, значение которого лежит в регистре *EAX* не доступно для чтения. Атакующий может подменить значение регистра *EAX* на другой адрес. Этот адрес будет указывать на код полезной нагрузки. Таким образом, в результате выполнения этой инструкции произойдёт передача управление на код атакующего. Для проверки этой ситуации необходимо выполнить следующие действия:

1. убедиться, что инструкция во время выполнения которой возникло исключение при попытке доступа к памяти – это инструкция вызова или без условной передачи управления.

2. проверить, что операндом является ячейка памяти, а также эта ячейка памяти адресуется с помощью регистров.
3. значение адреса ячейки памяти совпадает с адресом, из-за доступа к которому произошло исключение.

Третья ситуация возникает во время исключения при выполнении инструкции возврата *(RET)*. Исключение происходит из-за того, что доступ к ячейке памяти, на которую указывает указатель стека запрещён. Атакующий потенциально может подменить указатель стека другим значением. Это значение будет указывать на ячейку памяти, в которой храниться адрес, указывающий на код полезной нагрузки. После выполнения инструкции возврата осуществиться передача управления на код атакующего. Для того, чтобы убедиться в наличии именно этой ситуации достаточно проверить, что исключение при попытке доступа к памяти произошло в момент выполнения инструкции возврата.

Четвёртой ситуацией является исключение при выполнении инструкции записи в память. Эта ситуация описана в перечне слабостей программного обеспечения (*Common weakness enumeration*) под номером 123 и имеет высокую вероятность успешной эксплуатации. Контролируя адрес записи и записываемое значение, атакующий может перезаписать значения важных ячеек памяти, значением адреса, по которому располагается код полезной нагрузки. Примеры таких ячеек:

– ячейка на стеке, содержащая адрес возврата из функции.
– GOT-слот для вызова библиотечной функции.

Для того, чтобы проверить на наличие данной ситуации требуется:

1. удостовериться, что инструкция во время выполнения которой возникло исключение при попытке доступа к памяти – это инструкция записи в память.
2. проверить, что ячейка памяти адресуется через регистры, а также записываемое значение имеет размер равный либо 4 байта (x86) либо 8 байт (x86_64).
3. адрес записи совпадает с адресом, из-за доступа к которому произошло исключение.

Стоит отметить, что для проверки того, что атакующий контролирует записываемое значение, требуется проводить более сложный анализ, такой как анализ помеченных данных. Для того чтобы, максимально упростить и ускорить ста-

дию предварительной фильтрации использование анализа помеченных данных не предусматривается. Технология анализа помеченных данных будет применяться на стадии генерации эксплойта. Проверка контроля адреса записи обеспечивается третьим пунктом требований для второго и четвёртого случая. В дополнение к этой проверке используется следующая эвристика. Если значение адреса доступа достаточно мало (меньше порогового значения 65536), то считается, что произошла попытка доступа к неинициализированной переменной (обращение к нулевому указателю), и атакующий не контролирует это значение. В противном же случае считается, что значение адреса находится по контролем атакующего.

Описанные выше четыре ситуации, формируют группу эксплуатируемых классов аварийных завершений.Все остальные классы относятся к неэксплуатируемым классам аварийных завершений.

Исключения при попытке выполнить текущую инструкцию, а также при выполнении инструкции вызова или возврата указывают на возможность выполнить произвольный код. При выполнении произвольного кода атакующим возникает высокая вероятность нарушения конфиденциальности, целостности и доступности обрабатываемой информации. Исходя из системы оценок уязвимостей CVSS, такие ситуации имеют высокий или критический уровень угрозы.

Ниже приводится список классов аварийных завершений, которые относятся к группе эксплуатируемых аварийных завершений.

1. Аварийные завершения, возникшие в результате исключения при попытке выполнить текущую инструкцию. Адрес инструкции больше порогового значения.
2. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию вызова или безусловной передачи управления. Адрес из-за доступа к которому произошло исключение, больше порогового значения.
3. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию возврата. Адрес из-за доступа к которому произошло исключение, больше порогового значения.
4. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию записи в память. Адрес из-за доступа к которому произошло исключение, больше порогового значения.

Список неэксплуатируемых аварийных завершений содержит следующие классы.

1. Аварийные завершения, возникшие в результате исключения при попытке выполнить текущую инструкцию. Адрес инструкции меньше порогового значения.
2. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию вызова или безусловной передачи управления. Адрес из-за доступа к которому произошло исключение, меньше порогового значения.
3. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию возврата. Адрес из-за доступа к которому произошло исключение, меньше порогового значения.
4. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию записи в память. Адрес из-за доступа к которому произошло исключение, меньше порогового значения.
5. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию чтения из памяти. Адрес из-за доступа к которому произошло исключение, больше порогового значения.
6. Аварийные завершения, возникшие в результате исключения при попытке выполнить инструкцию чтения из памяти. Адрес из-за доступа к которому произошло исключение, меньше порогового значения.
7. Аварийные завершения, возникшие в результате срабатывания защиты в безопасной функции.
8. Аварийные завершение, возникшие в результате срабатывания «канарейки».
9. Аварийные завершение, возникшие в результате срабатывания защиты менеджера памяти.
10. Аварийные завершение, возникшие в результате деления на ноль.
11. Аварийные завершение, возникшие в результате нарушения доступа к памяти.
12. Аварийные завершение, возникшие в результате получения сигнала SIGABRT.
13. Аварийные завершение, возникшие в результате получения сигнала SIGFPE.
14. Нормальное завершение программы.

Неэксплуатируемые аварийные завершения под номерами 1-4 совпадают эксплуатируемые классами, за исключением того, что адрес доступа к памяти меньше порогового значения. Для класса аварийных завершений под номерами 5 необходимо проводить дополнительный анализ, аналогично подходу из работы [26]. Этот анализ позволит утверждать, что контролируемое значение адреса может попасть в инструкцию передачи управления. Класс под номером 6 является неэксплуатируемым из-за не выполнения требований по значению адреса доступа. Классы 7, 8, 9 описывают виды защитных механизмов и проверок, которые предотвращают эксплуатацию. Класс 10 не позволяет произвести эксплуатацию с выполнением произвольного кода. Аварийные завершения, вызванные исключением нарушения доступа, которые не удалось классифицировать как эксплуатируемые или отнести к неэксплуатируемым классам под номерами 1-6, относятся к неэксплуатируемому классу под номером 11. Для программ, работающих под управлением ОС Linux аварийные завершения, которые не удалось отнести к классам 7-9, относятся классу номер 12. Все исключения связанные с арифметическими операциями, если это не целочисленное деление на ноль, и при этом программа работает под управлением ОС Linux относятся к классу под номером 13. Специальный класс неэксплуатируемых аварийных завершений под номером 14 описывает ситуацию, когда программа корректно завершилась. Стоит отметить, что представленный набор классов аварийных завершений насчитывает 18 классов и является расширяемым.

2.1.2 Анализ аварийных завершений

Реализация правил для определения рассмотренных выше классов аварийных завершений предполагает использование динамического анализа бинарного кода. Это вытекает из начальных условий - имеется набор входных данных, на которых программа аварийно завершается. В качестве одного из основных требований к анализу, является скорость работы, которая должна быть сравнима с запуском программы без анализа. Таким образом, подходящими кандидатами являются следующие подходы:

– отладка;

– динамическая бинарная инструментация.

На базе отладчика gdb [35] реализован инструмент анализа аварийных завершений gdb exploitable plugin [25]. Этот инструмент используется для анализа программ, работающих под управлением ОС Linux. Для программ работающих под управлением ОС Windows существует инструмент !exploitable [24] реализованный на основе отладчика Windbg [32]. В итоге, получилось, что для разных платформ – свой инструмент со схожими функциями.

Динамическая бинарная инструментация даёт больший простор для автоматизации анализа, чем отладка. Помимо анализа состояния программы в момент аварийного завершения, инструментация позволяет собирать информацию во время выполнения программы. Например, информацию о вызовах интересных для анализа функций (stack_chk_fail, malloc_printerr, _chk_fail и т.д.), а также информацию о пройденных базовых блоках во время выполнения. Помимо этого, динамическая бинарная инструментация обладает необходимыми возможностями для проведения анализа: перехват сигналов и исключений, дизассемблирование инструкций. Кроме того, существуют кроссплатформенные системы, реализующие данную технологию. Одной из такой систем является DynamoRIO [45].

Исходя из приведённых выше рассуждений, реализация правил для определения классов аварийных завершений будет использовать в своей основе динамическую бинарную инструментацию.

2.1.3 Взаимосвязь между дефектами и классами аварийных завершений

Как уже было отмечено ранее, аварийные завершения программ являются следствием проявления дефектов кода. На рис. 2.4 представлено соответствие между дефектами на уровне языка программирования и эксплуатируемыми классами аварийных завершений программы.

Исключение во время записи в память может быть следствием двойного освобождения памяти [46] или переполнения буфера на куче [47] в некоторых менеджерах памяти, например, *Doug Lea allocator* [48]. А также в случае перезаписи указателя при переполнении буфера на стеке [49] или куче [47] с последующей записью значения по перезаписанному указателю. Стоит отметить,

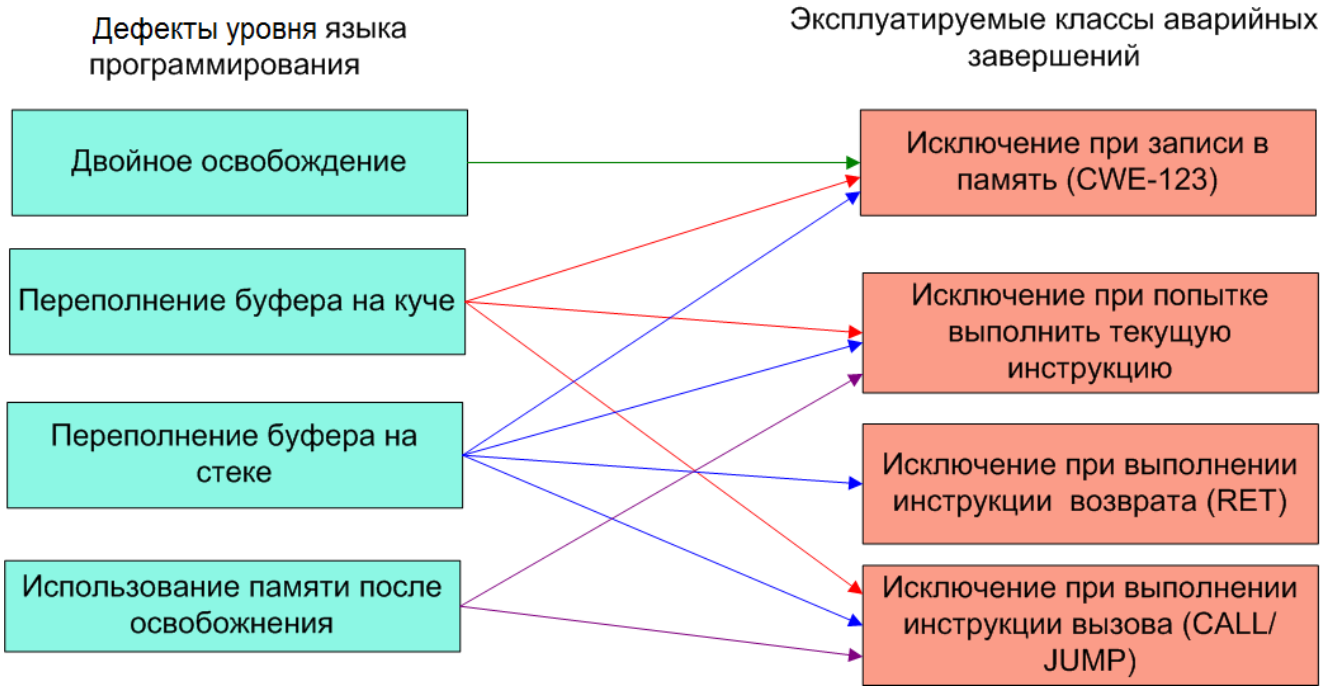

Рисунок 2.4 — Соответствие между дефектами на уровне языка программирования и эксплуатируемыми классами аварийных завершений программы.

что если такая ситуация произойдёт при переполнении буфера на стеке, то появляется возможность обойти «канарейку», путём точной перезаписи адреса возврата из функции.

Исключение при выполнении инструкции возврата, как правило, является следствием переполнения буфера на стеке. Если удаётся прочитать перезаписанное значение адреса возврата из функции, то происходит передача управления и тогда может возникнуть исключение при попытке выполнить текущую инструкцию.

Исключение при выполнении инструкции вызова возникает в случае переполнения буфера на стеке или куче, в результате которого перезаписывается указатель на функцию. Данное исключение может произойти при использовании памяти после её освобождения [50]. Если значение, по которому должна осуществиться передача управления доступно для чтения, то после передачи управления может возникнуть исключение при попытке выполнить текущую инструкцию.

Исходя из выше сказанного, эксплуатируемые классы аварийных завершений могут быть следствием различных программных дефектов.

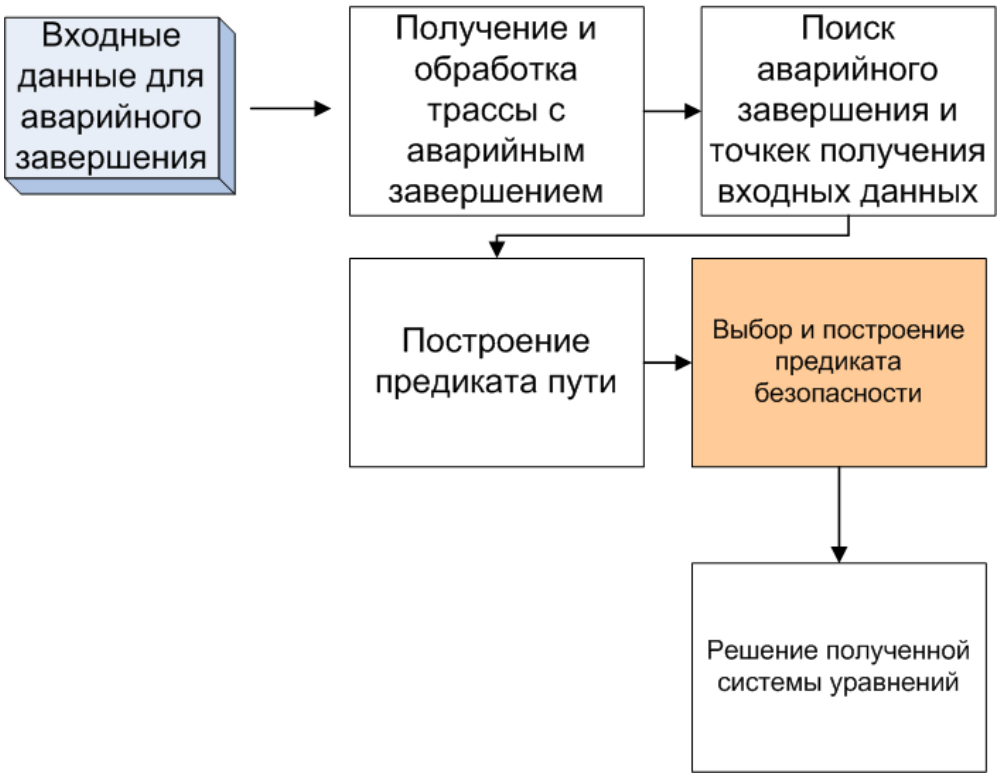

Рисунок 2.5 — Схема метода автоматической генерации эксплойтов

2.2 Метод автоматической генерации эксплойтов

В основе метода автоматической генерации эксплойтов лежит анализ трасс выполнения [51–53], полученных с помощью полносистемного эмулятора [54; 55]. Трасса представляет собой последовательный набор выполненных процессором инструкций и значении всех регистров на момент выполнения каждой инструкции. Номер выполненной инструкции называется шагом трассы. В виду того, что используется полносистемная эмуляция для получения трассы, в неё попадают инструкции не только исследуемой программы, но и всех остальных запущенных программ, а также инструкции, относящиеся к работе операционной системы. На рис. 2.5 представлена схема метода автоматической генерации эксплойтов.

Метод получает на вход набор входных данных для программы, который приводит её к аварийному завершению. В качестве первого этапа происходит получение и обработка трассы выполнения программы. Во время работы исследуемой программы происходит аварийное завершение. При обработке трассы повышается уровень её представления [52] и формируется вся необходимая информация для дальнейшего анализа. Диапазоны шагов трассы размечаются на

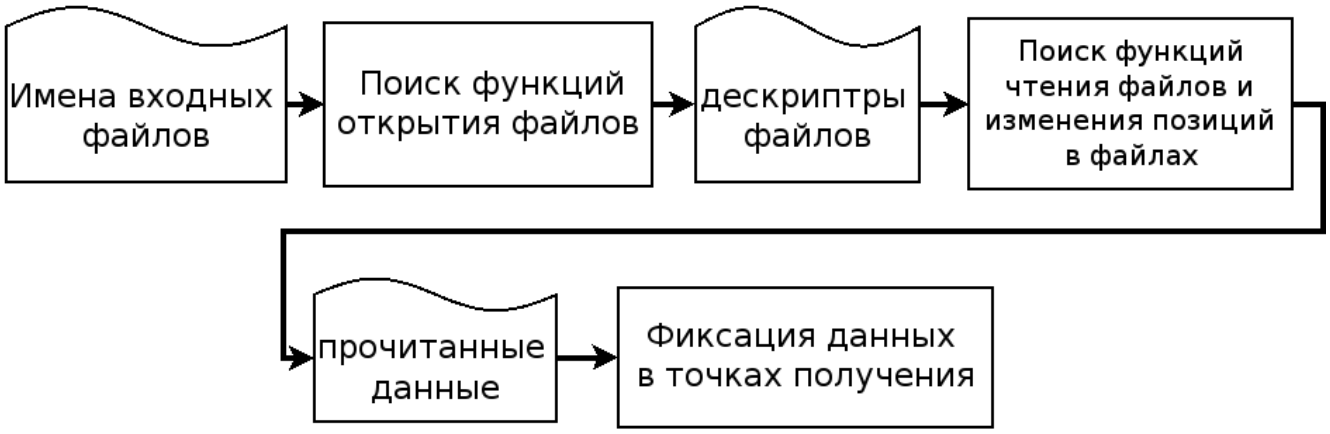

Рисунок 2.6 — Схема определения точек получения входных данных

принадлежность к определённому процессу и потоку выполнения. Определяются границы функций и обработчиков прерываний в трассе. Происходит построение карты исполняемых модулей для каждого процесса. Осуществляется вычисление входных и выходных параметров для функций.

2.2.1 Определение точек получения входных данных

Анализируемая программа может получать данные на вход из различных источников: аргументов командной строки, файлов, сети, переменных окружения. Обработку получения данных из этих источников можно обобщить, если организовать доставку данных в программу с помощью файлов. Перед тем, как попасть в программу, входные данные должны быть прочитаны из файла. Прочитанные из файла данные, до получения их программой отслеживаются с помощью анализа помеченных данных через физическую память. Учитываются только зависимости по копированию, предполагается, что от чтения файла до попадания в программу, данные не модифицируются. На рис. 2.6 представлена схема определения точек получения входных данных. Для определения точек получения входных данных потребуются имена файлов, которые содержат эти данные. Сперва происходит поиск вызовов функций открытия файлов. Если имя открываемого файла совпадает с одним из входных имён файлов, то затем начитается слежение за дескриптором файла, который получен после выполнения функции открытия. С помощью дескрипторов отслеживаются функции чтения файлов и функции изменения позиции в файле. Стоит отметить, что

для функций открытия файла, чтения файла и для функции изменения позиции в файле наиболее целесообразно использовать соответствующие системные вызовы. Это гарантировано позволит обработать все операции с файлами в текущей трассе. От шагов трассы, на которых произошло чтение входного файла начинается отслеживание прочитанных данных. Для отслеживания необходимо определить точки в трассе, на которых ожидается получение входных данных. Все эти точки будут принадлежать процессу исследуемой программы. Описание источников получения входных данных имеет следующий вид:

- Аргументы командной строки и переменные окружения представляются собой область стека, которая начинается с указателя стека и имеет достаточный размер (10000 байт) на шаге трассы с точкой входа (origin entry point) в исследуемой программе.
- Чтение данных из файла описывается библиотечной функцией чтения данных.
- Получение данных из потока ввода описывается библиотечной функцией чтения данных, которая принимает нулевой дескриптор файла.
- Получение данных из сети описывается библиотечной функцией получения данных из сети.

Если помеченные данные доходят до источников получения входных данных, то формируется описание точки получения входных данных. Точка получения входных данных представляет собой:

- буфер виртуальной памяти (адресс и размер), в котором находятся помеченные данные (данные из файла).
- шаг трассы с источником получения входных данных.
- имя файла из которого были получены данные.
- смешение в файле, с которого начинаются полученные входные данные.

Точки получения входных данных будут использоваться в дальнейшем для построения предиката пути.

2.2.2 Определение точки аварийного завершения

Как уже отмечалось в разделе 2.1, аварийное завершение программы происходит в следствии возникновения исключительной ситуации, вызванной сра-

батыванием дефекта. В качестве такой ситуации выступает прерывание процессора. Тем не менее, не каждое прерывание приводит к аварийному завершению. Например, асинхронные прерывания, приходящие от устройств, обеспечивают взаимодействие между операционной системой и этими устройствами. Синхронные прерывания, например, вызванные обращением к недопустимым адресам памяти, являются следствием срабатывания дефекта. Для поиска аварийного завершения происходит фильтрация прерываний, которые попали в трассу. Исключаются из рассмотрения те прерывания, после обработки которых было продолжено выполнение программы, а также прерывания, возникшие не во время работы исследуемого процесса.Из отфильтрованных таким образом прерываний формируется список из шагов трассы, на которых возникли прерывания. Список упорядочен по возрастанию шагов.

Для дальнейшего поиска аварийного завершения применяется слайсинг трассы [56]. Критерием слайсинга выступают точки получения входных данных, которые также формируют собой список, упорядоченный по возрастанию шагов трассы. Слайсинг используется прямой – анализ происходит по возрастанию шагов трассы. Если текущий шаг анализа совпадает с шагом источника получения входных данных, то буфер с входными данными добавляется в множество отслеживаемых элементов. Если шаг анализа совпадает с шагом потенциального аварийного завершения, проводятся необходимые проверки, подобно тем, которые используются для определения классов аварийных завершений. Благодаря наличию отслеживанию помеченных данных при построении слайса, можно определить, какие операнды инструкций, при выполнении которой возникло прерывание, зависят от входных данных. В связи этим возникает более расширенная классификация эксплуатируемых завершений.

1. Исключение, возникшие при выполнении инструкции возврата (ret), при условии, что значение указателя стека зависит от помеченных данных.
2. Исключение, возникшие при выполнении инструкции возврата (ret), при условии, что значение адреса возврата из функции зависит от помеченных данных.
3. Исключение, возникшие при выполнении инструкции вызова или безусловной передачи управления. Значение адреса назначения находится в регистре и зависит от входных данных.

4. Исключение, возникшие при выполнении инструкции вызова или безусловной передачи управления. Значение адреса назначения находится в памяти. Регистры, формирующие выражение для адреса этой памяти, зависят от входных данных.
5. Исключение, возникшие при выполнении инструкции записи в память. Адрес записи и записываемое значение зависят от входных данных. Размер записываемого значения равен 4 байта для 32-ух разрядных ОС, либо 8 для 64-ёх разрядных ОС.

Если исключение, соответствует одному описанных выше условий, то считается, что это и есть искомое аварийное завершение. Шаг и информация об аварийном завершении запоминается. Эта информация потребуется при построении предикатов пути и безопасности. Если во время анализа не одно исключение не удовлетворило описанных требований, то генерация эксплойта прекращается на этом шаге.

Условия с номерами 2 и 3 соответствуют классу аварийных завершений: исключение при попытке выполнить текущую инструкцию. Во время анализа аварийных завершений (раздел 2.1) фиксируется лишь следствие: нарушение доступа к памяти. А ситуации 2 и 3 описывают причину, по которой произошло это нарушение.

После определения точки аварийного завершения начинается построение предиката пути. Конечной точкой при построении предиката пути является шаг трассы, на котором произошло аварийное завершение программы. Построив предикат пути, и основываясь на информации, полученной при определении точки аварийного завершения, формируется предикат безопасности. Затем, объединив предикат пути и безопасности начинается процедура решения полученной системы. Если систему удалось решить, то решением является эксплойт, который представляется набором входных файлов.

2.3 Предикат пути

Предикат пути представляет собой набор символьных уравнений и неравенств, которые описывают процесс выполнения программы. Решением преди-

```
(define-sort Char() (_ BitVec 8))
(define-sort Index() (_ BitVec 32))
(declare-const file_0 (Array Index Char))
```

Рисунок 2.7 — Определение символьной переменной для входного файла

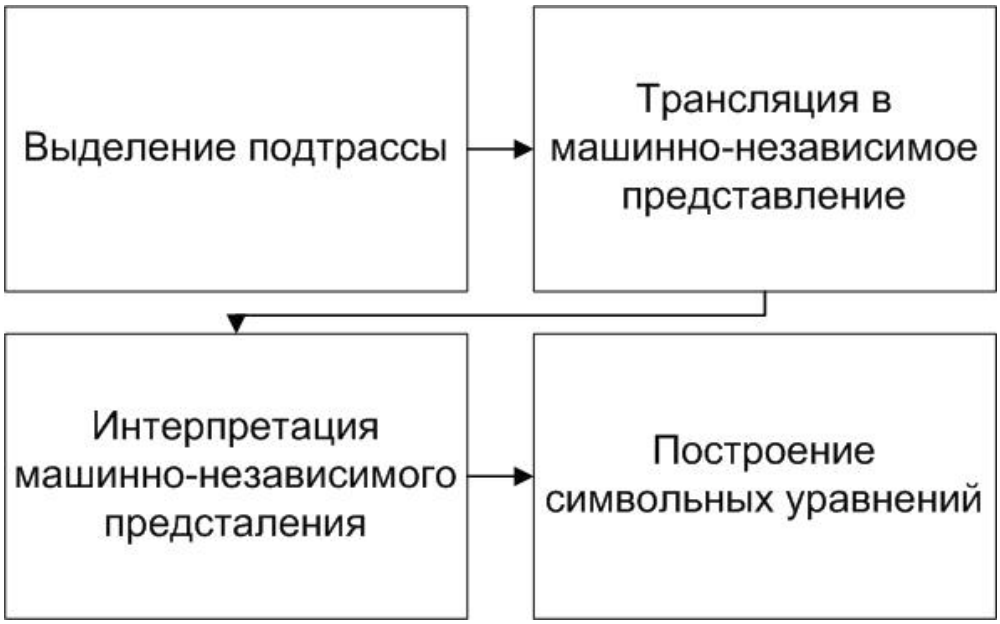

Рисунок 2.8 — Схема построения предиката пути.

ката пути является набор входных данных, на котором выполнение программы выбирает тот же путь, что использовался при построении предиката пути.

Символьными переменными являются входные файлы. Каждый входной файл представляет собой массив из битовых векторов. На рис. 2.7 показано определение символьной переменной для входного файла на языке smt-lib2 [57]. Из представленного определения видно, что создаётся массив битовых векторов с именем *file_ 0*. Индексом массива выступает 32-ух битный вектор, а элемент массива представляет собой битовый вектор размером 8.

На рис. 2.8 представлена схема построения предиката пути.

2.3.1 Выделение подтрассы

В основе построения предиката пути , как и при поиске точки аварийного завершения, лежит слайсинг трасс выполнения [56]. Начальным шагом для построения слайса является первая точка получения входных данных. На каждом шаге получения входных данных ко множеству отслеживаемых элементов добавляется буфер с входными данными. Конечным шагом является шаг с аварийным завершением программы. При построении слайсинга отслеживается физическая память, а не виртуальная. Это значит, что в слайс могут попасть инструкции, принадлежащие не только процессу исследуемой программы, но и другим процессам, а также ядру операционной системы. Т.к. аварийное завершение происходит в процессе исследуемой программы, то достаточно отбирать инструкции только из этого процесса. Это позволит сократить количество отобранных инструкций, как следствие уменьшить набор уравнений в предикате пути и сократить время решения полученной системы. Но к сожалению, нельзя удалить из слайса все ненужные инструкции, оставляя только те, которые принадлежат процессу исследуемой программы. Такое удаление может привести к тому, что предикат пути не будет иметь решений. Это произойдёт из-за внезапного добавления новых отслеживаемых значений, а также во время «не убитых» помеченных ячеек. Для того, чтобы избежать такой ситуации, используется следующий подход. Пусть W – набор отслеживаемых ячеек перед началом выполнения кода из другого процесса или ядра ОС. Рассмотрим выполнение этого кода как атомарную операцию, меняющую набор отслеживаемых элементов. W' – набор помеченных ячеек, перед возвратом к выполнению кода исследуемого процесса, имеет вид: $W' = (W \cup A) \setminus D$, где A – множество добавленных ячеек, а D – множество удалённых элементов. Тогда желаемый набор отслеживаемых ячеек имеет вид: $W'' = W' \cap W$. В результате этой операции, добавленные ячейки не будут присутствовать в конечном множестве, а удалённые ячейки в другом процессе или ядре ОС будут учтены. Вернувшись к выполнению кода исследуемого процесса, построение слайсинга проходит в обычном режиме. Слайс представляет собой набор инструкций, участвующих в обработке входных данных, а также множество помеченных ячеек на момент выполнения данной инструкции.

2.3.2 Трансляция в промежуточное представление

Каждая отобранная инструкция в результате выделения транслируется в промежуточное представление. Использование промежуточного представления обеспечивает возможность анализировать различные процессорные архитектуры единообразно. Кроме того, такие представления, как правило, обладают небольшим набором операторов, что упрощает анализ архитектур с большим числом команд, и команд, обладающих сложной семантикой. В качестве примера такой архитектуры можно привести архитектуру *x86*.

Трансляция машинных инструкций происходит в промежуточное представление *Pivot* [58; 59], которое отвечает необходимым требованиям. Система бинарной трансляции *Pivot* позволяет описывать операционную семантику как пользовательских, так и системных инструкций современных процессорных архитектур. Поддерживаются различные по своей природе процессорные архитектуры, как CISC так и RISC. Существуют трансляции для следующих архитектур: Intel 64, MIPS64, PowerPC , ARM v6, SPARC и Motorolla m68k. Описание семантики инструкций этих архитектур достаточно для моделирования всех эффектов выполнения инструкции, в том числе побочных (влияние на слово состояния машины).

Промежуточное представление *Pivot* приближено к SSA-представлению, что позволяет без существенных дополнительных затрат применять различные виды компиляторных оптимизаций, таких как удаление общих подвыражений или удаление мертвого кода. Обеспечивается возможность интерпретации промежуточного представления.

При моделировании современных наборов инструкций возникает ряд сложностей. Одной из таких сложностей является нетривиальный вид регистровых файлов современных процессоров. Регистры могут располагаться в нескольких теневых наборах (MIPS64), а также являться частями друг друга (Intel 64) или иметь организацию в виде регистровых окон (SPARC). Подобные сложности приводят к необходимости обеспечения достаточно гибкой модели регистрового файла. Такой моделью может служить модель адресных пространств, описание которой приводится далее.

Ещё одной сложностью, встречаемой в CISC-архитектурах, является наличие слова состояния машины. Изменением слова состояния является побочный

Рисунок 2.9 — Схема модели описания операционной семантики инструкции.

эффекта выполнения многих инструкций. В *Pivot* исключено неявное обновление слова состояния. На рис. 2.9 представлена схема модели операционной семантики инструкции.

Модель семантики отдельной инструкции является упорядоченным набором операторов,который описывает действия, производимые процессором при выполнении данной инструкции.Для применения операций, обращения в адресные пространства и ветвления используются отдельные операторы. Локальные переменные фигурируют в качестве параметров и возвращаемых значений операций. Осуществляется несложный контроль типов на уровне атомов.Операторы читаются (при интерпретации – выполняются) последовательно. Область видимости локальных переменных ограничена единицей трансляции, то есть одной инструкцией.

Атомы являются первой из концепций, вводимых в модели.Атом представляет собой машинный тип, то есть элементарный блок данных, которым могут оперировать машинные инструкции. В качестве примеров атомов можно привести: 32-битные целые числа, числа с плавающей точкой двойной точности и так далее. Атом характирезуется размером, описываемого блока данных. Для дальнейшего изложения под i8, i16, i32, i64 понимаются соответственно 8-, 16-, 32- и 64-битные целочисленные атомы.

Как и при любом применении операционного подхода к описанию семантики, в модели определены примитивные операции, через которые будет описываться поведение целевой машины. Принимая во внимание требованием о простоте модели, появляется требование к отсутствию побочных эффектов у операций.Такой подход привёл бы к огромным по объему моделям инструк-

ций Intel 64 из-за обновления большого количества флагов слова состояния. В модели операционной семантики *Pivot* используется облегчённое требование к побочным эффектам операций: операция не должна иметь побочных эффектов, кроме, возможно, чтения и записи некоторых флагов модельного слова состояния.

Модельное слово состояния представляется в виде 16-битного значения, которое включает флаги состояния Intel 64: AF, CF, OF, PF, SF, ZF. Практика показывает, что обновления этих флагов достаточно не только для моделирования Intel 64, но и RISC-машин, таких как MIPS64 или ARM.

Операции указывают явно i-маску и o-маску. i-маска является набором флагов слова состояния, которые данная операция просматривает, а o-маска специфицирует набор изменяемых опирацией флагов.Таким образом, операция обладает следующими характеристиками: имя, возвращаемый атом, атомы параметров, i-маска и o-маска. Как уже отмечалось ранее, для решения проблемы сложно организованного регистрового файла применяется модель адресных пространств. В этой модели все методы адресации, с которыми может работать машина (регистры, память, порты ввода-вывода), сводятся к единой схеме: каждое адресуемое данное представляется в виде ссылки на одно из адресных пространств и снабжается указателем в это адресное пространство. Отображения пространства оперативной памяти вводится естественным образом. Несложно расширить естественное представление и на пространство портов ввода-вывода Intel 64. Регистровый файл отображается в адресное пространство следующим образом. Выбирается размер адреса, позволяющий вместить все адресуемые регистры машины, после чего регистры размещаются в строящемся пространстве в произвольном порядке, но с учётом наложений и пересечений. При использовании такого подхода не только решается проблема сложной организации регистрового файла, но и вводится унифицированная модель адресуемых данных, что позволяет не рассматривать отдельно каждый из их видов. На рис. 2.10 представлена схема модели адресного пространства регистров для архитектуры *x86*. Например, регистру EAX соответствует отрезок [0;4], а регистру AX отрезок [0;2].Таким образом поддерживается вложенность регистров в друг друга.

Во время построения операторов в описываемой модели используются локальные переменные, время жизни которых ограничено единицей трансляции. То есть, при трансляции отдельных инструкций локальные переменные исполь-

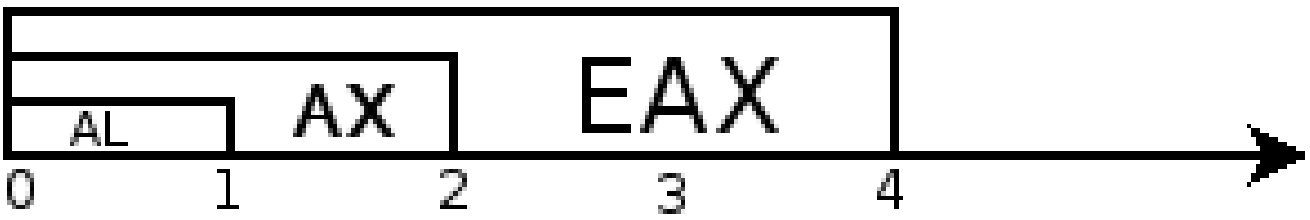

Рисунок 2.10 — Схема модели адресного пространства регистров для архитектуры x86.

зуются для хранения результатов промежуточных вычислений и переменные локальны в рамках данной инструкции. Локальные переменные описываются номерами. К локальным переменным предъявляются требования SSA-формы. Передача управления внутри машинных инструкций достаточна проста, поэтому оператор φне вводится. При определении локальной переменной с ней сопоставляется атом правой части соответствующего оператора. Локальные переменные обозначаются через t.N , где N — номер переменной. Модель содержит 8 операторов, в которые отображается операционная семантика транслируемых машинных инструкций:

- оператор NOP;
- оператор INIT;
- оператор LOAD;
- оператор STORE;
- оператор APPLY;
- оператор BRANCH;
- семейство операторов SPECIAL;
- операторы ANNOTATION.

Оператор NOP не производит никакого эффекта.

Оператор INIT инициализирует локальную переменную константным значением, которое задаётся в виде битовой последовательности и атома.

Оператор LOAD загружает значения из одного из адресных пространств в локальную переменную, например, значение регистра. Для оператора LOAD указывается адресное пространство, из которого осуществляется загрузка, атом загружаемых данных, локальная переменная с адресом данного и локальная переменная, принимающая результат. При этом атом локальной переменной адреса должен совпадать с атомом адреса пространства.

Оператор STORE записывает значение в одно из адресных пространств. Для оператора STORE указывается адресное пространство для записи и ло-

кальные переменные адреса и значения. Атом локальной переменной адреса должен совпадать с атомом адреса пространства.

Оператор APPLY применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.

Оператор BRANCH выполняет передачу управления. Существуют три вида передачи управления: безусловная, условная по маске и условная по сложному предикату. Во всех случаях указывается знаковое смещение для передачи управления (в качестве единицы адреса используется один модельный оператор). При использовании условной передачи управления по маске указываются два 16-битных числа: *rmask* и *lmask*. Передача осуществляется тогда и только тогда, когда все выставленные биты в *rmask* выставлены и в модельном слове состояния, а также все выставленные биты в *lmask* сброшены в модельном слове состояния. Условная передача управления по сложному предикату проверяет один из предикатов A (беззнаковое больше), AE (беззнаковое больше или равное). B (беззнаковое меньше), BE (беззнаковое меньше или равно), G (знаковое больше), GE (знаковое больше или равно), L (знаковое меньше), LE (знаковое меньше или равно).

Семейство операторов SPECIAL указывает на особое моделируемое состояние. Сюда относится оператор *HALT* (останов до возникновения внешнего события) и оператор *TRAP* (исключение или прерывание).

Операторы ANNOTATION. Эти операторы игнорируются при интерпретации отдельной инструкции и не влияют на модельную семантику. Они используются для задания дополнительных аннотаций, которые могут быть использованы в отдельных видах анализа.

На рис. 2.11 представлена трансляция машинной инструкции *sbb eax, ebx* архитектуры *x86* в промежуточное представление *Pivot*. Трансляция происходит статически – в результирующем наборе *Pivot-инструкций* присутствуют все возможные варианты (пути) выполнения машинной инструкций. В первом блоке на 2.11 происходит загрузка значения регистра флагов в локальную переменную. Затем выполняется операция «битового и» с единицей. Она используется для проверки установки флага CF. Если флаг CF выставлен управление передаётся в блок под номером 3, в противном случае управление попадает во второй блок. В блоке номер 2 происходит загрузка значений регистров в локальные переменные, осуществляется операция вычитания, сохранение результата в регистр *EAX*. Операция *x86.uf* обновляет значение регистра флагов, в соот-

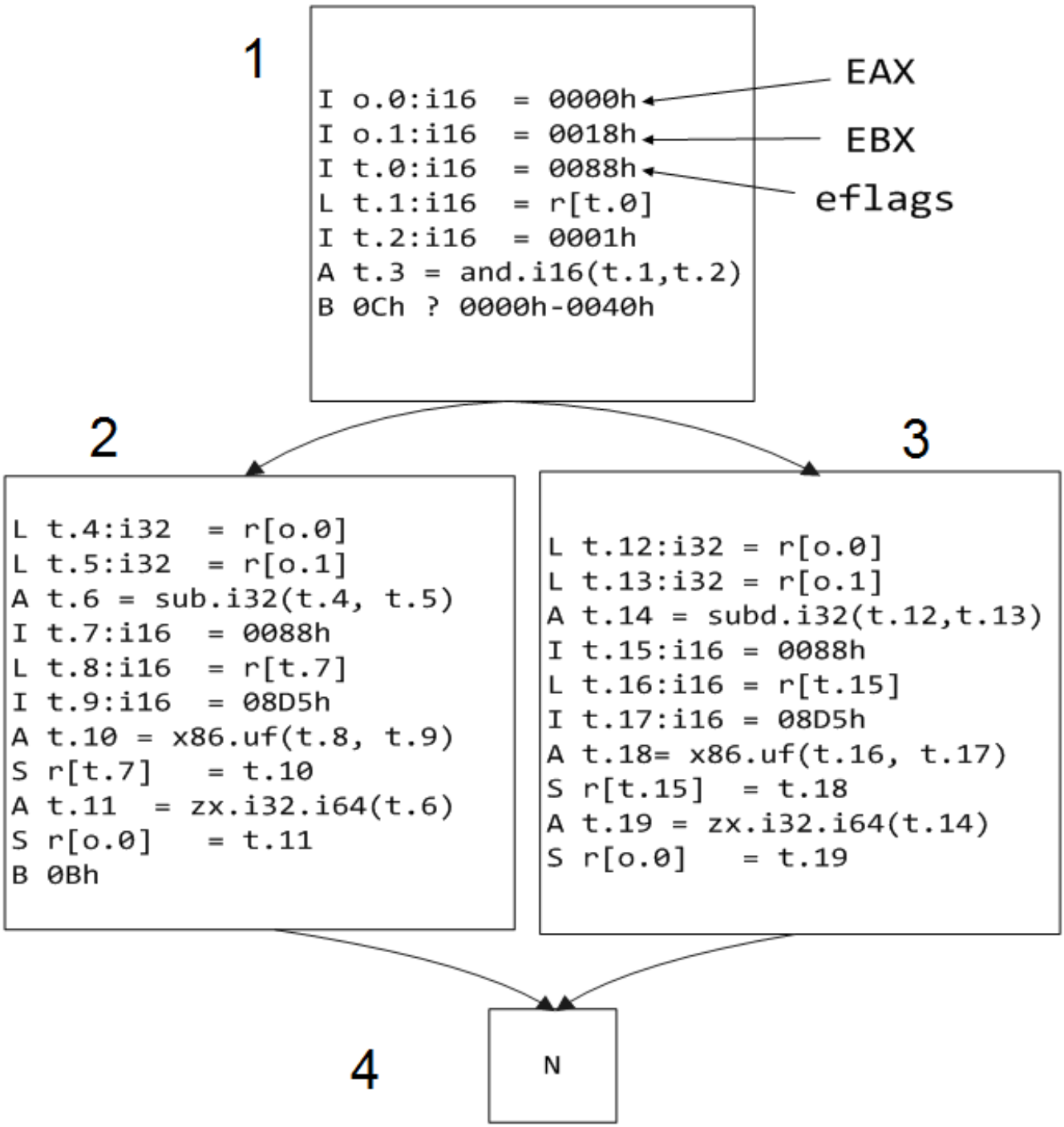

Рисунок 2.11 — Трансляция машинной инструкции в промежуточное представление.

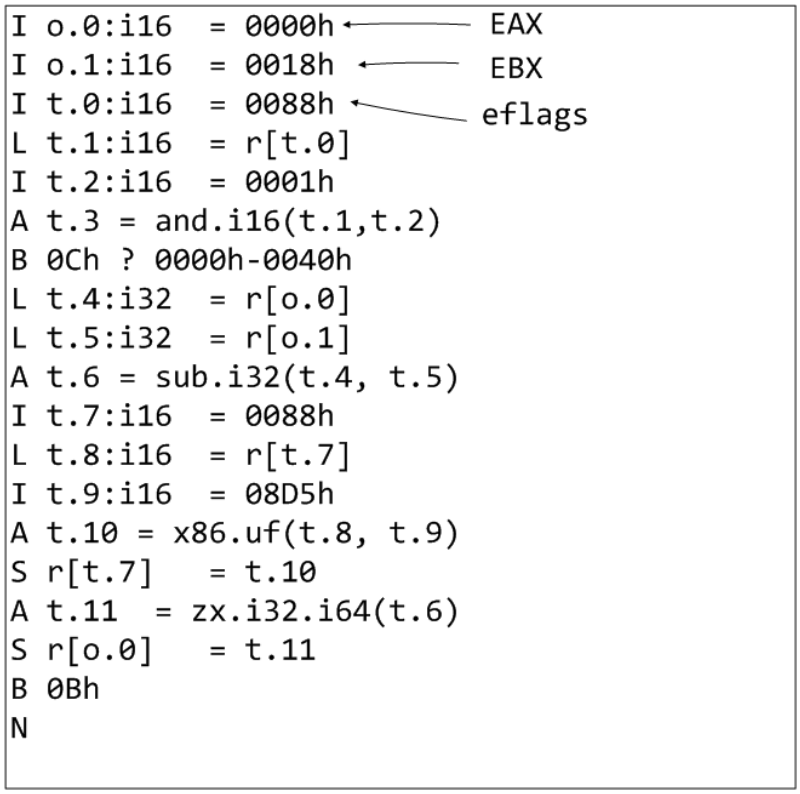

Рисунок 2.12 — Результат интерпретации промежуточного представления.

ветствии с заданной маской. Обновлённое значение также сохраняется. Блок 3 идентичен блоку 2, за исключением, того что вместо операции вычитания используется операция вычитания с декрементом. Из блоков 2 и 3 управление попадает в блок 4, в котором единственный оператор *NOP*.

2.3.3 Интерпретация промежуточного представления

Интерпретация *Pivot-кода* требуется для того, чтобы определить какие *Pivot-инструкций* выполнились в трассе во время выполнения машинной инструкции. Трансляция только выполненных инструкций позволяет сократить набор уравнений. Рассматривая всего один путь выполнения внутри машинной инструкции даёт возможность избежать сложной конструкции: *if then else*, которая использовалась бы для описания нескольких путей выполнения. На рис. 2.12 представлен результат интерпретации машинной инструкции *sbb eax, ebx* архитектуры *x86*. Предполагается, что на момент выполнения машинной инструкции флаг CF не был выставлен. Тогда исходя из рисунка 2.11 поток

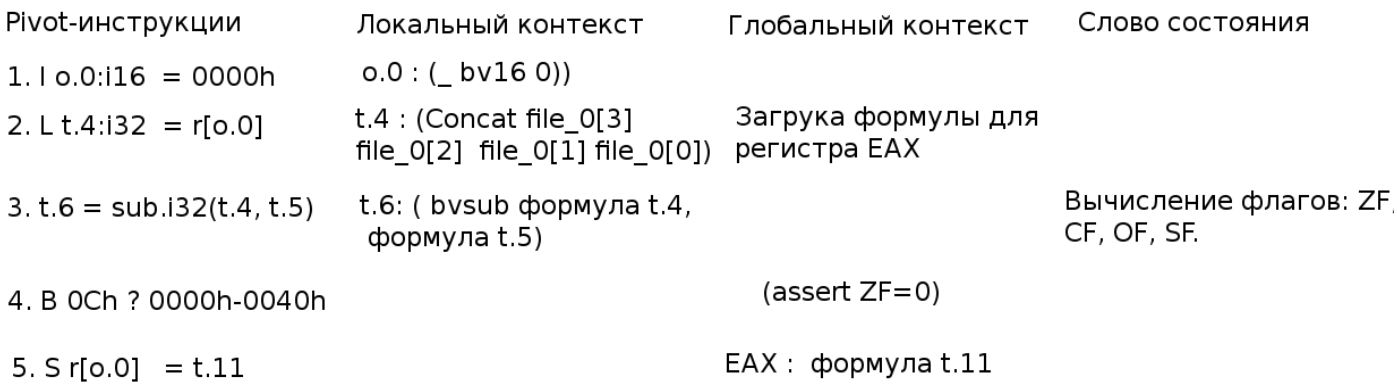

Рисунок 2.13 — Пример трансляции *Pivot-трассы* в символьные формулы.

управления пройдёт через блоки:1, 2, 4. По полученной таким образом, *Pivot-трассе* будет происходить построение символьных формул.

2.3.4 Построение символьных формул

Каждая инструкция *Pivot-трассы* транслируются в соответствующие ей символьные формулы. Во время построения предиката пути поддерживаются два контекста. Первый – глобальный контекст в нём содержится отображение элементов адресного пространства (регистры, память) на символьные формулы. Обновление отображения в глобальном контексте происходит в двух случаях:

1. На каждом шаге получения входных данных в отображение заносится следующая запись. Прочитанному буферу с данными из файла соответствует участок символьного файла, который начинается с заданного смещения.
2. На шаге трассы, в котором происходит запись помеченного значения или запись константы в помеченный элемент адресного пространства.

Второй контекст – локальный. Время жизни локального контекста – трансляция одной машинной инструкции. Локальный контекст содержит отображения локальных переменных в символьные формулы. Также при трансляции инструкции машины моделируется слово состояния *Pivot*, которое представляет собой 16-битный вектор. Обновление слова состояния происходит согласно маскам при выполнении операций. На рис. 2.13 представлен пример трансляции *Pivot-трассы* в символьные формулы.

Трансляция оператора *NOP* не происходит, т.к. он не оказывает никакого эффекта.

При трансляции оператора *INIT* происходит формирование константного битового вектора. Размер этого вектора равен атому локальной переменной, а значение соответствует инициализируемому значению. После этого обновляется отображение в локальном контексте. На рис. 2.13 трансляции оператора *INIT* соответствует трансляция инструкции под номером 1.

Трансляция оператора *LOAD* осуществляет формирование символьного значения для элемента адресного пространства. Для помеченных байтов элемента символьные формулы извлекаются из глобального контекста. Для константных значений формируются константные битовые вектора, значения констант получаются из трассы. Затем происходит склейка формул для отдельных байтов и обновление локального контекста. Размер полученного битового вектора равен атому локальной переменной. На рис. 2.13 трансляции оператора *LOAD* соответствует трансляция инструкции под номером 2. Значение всего регистра EAX в данном случае считается помеченным.

Во время трансляции оператора *APPLY* модельная операция заменяется соответствующей операцией над битовыми векторами. Операнды, используемые в операции заменяются битовыми векторами, которые берутся из локального контекста. Результирующая формула также сохраняется в локальном контексте. Обновляются необходимые флаги слова состояния. Флаги представляют собой формулы над битовыми векторами. На рис. 2.13 трансляции оператора *APPLY* соответствует трансляция инструкции под номером 3.

Трансляция оператора *BRANCH* формирует уравнение, которое добавляется в общую систему уравнений, хранящуюся в глобальном контексте. Именно уравнения для этого оператора и формируют предикат пути. Результат выполнения оператора известен благодаря интерпретации. На рис. 2.13 трансляции оператора *BRANCH* соответствует трансляция инструкции под номером 4.

В результате трансляции оператора *STORE* осуществляется обновление глобального контекста. Формула для записываемой переменной извлекается из локального контекста. На рис. 2.13 трансляции оператора *STORE* соответствует трансляция инструкции под номером 5.

Трансляция операторов *SPECIAL* и *ANNOTATION* не производится, семантика этих операторов не требуется при построении предиката пути.

2.3.5 Недостаточная и избыточная помеченности

Алгоритмы анализа помеченных данных обладают ограничениями, известными как недостаточная помеченность и избыточная помеченность.

Недостаточная помеченность возникает, как правило, из-за того, что во время анализа не учитываются некоторые виды зависимостей. В ходе работы программы символьные данные могут оказаться в адресном коде, определяя значение адреса памяти. Дальнейшая интерпретация кода либо предполагает обращение к любой ячейке адресуемой памяти, либо требует ограничения числа возможных адресов, вплоть до конкретизации значения адреса. Такая проблема известна в публикациях, как проблема «символьных адресов». В рамках предлагаемого подхода символьные пометки не распространяются через адреса, что может приводить к недостаточной помеченности. В свою очередь, недостаточная помеченность может привести к тому, что набор входных данных, полученный в результате решения предиката пути, не проведет программу по тому же самому пути выполнения. В некоторых случаях последствий недостаточной помеченности можно избежать, добавляя дополнительные ограничения на символьные переменные. Обладая некоторыми сведениями об устройстве программы, аналитик может добавить ограничения на входные символьные файлы, параметры функций, а также на произвольные ячейки памяти и регистры. В качестве примера, можно привести ситуацию, когда данные копируются при помощи функции *sscanf*. Аналитик может добавить ограничения на копируемый буфер, таким образом, что в нём не будут содержаться терминальных символов и пробелов.

Избыточная помеченность может привести к росту количества отобранных машинных инструкций, не все из которых оказывают существенное влияние на ход анализа. В основном, такие ситуации происходят из-за выполнения кода библиотек. Прекращение отслеживания помеченных данных при выполнении кода библиотек может привести к потере нужных зависимостей, например, если копирование помеченных данных происходит с использованием функций копирования. Поэтому предлагается следующий подход. Аналитик указывает список исключённых из анализа функций. Помимо этого, для каждой функции задаётся набор параметров, с которых необходимо снять пометки. Таким образом, в предикат пути не попадут те инструкции внутри функции, на ре-

зультат выполнения которых могли повлиять выбранные параметры. Примером такой функции может послужить функция *malloc* из библиотеки *libc*. Использование данного подхода позволяет значительно снизить количество отобранных инструкций.

2.4 Предикат безопасности

Предикат безопасности описывает состояние программы, при котором осуществляется эксплуатация, т.е. выполнение кода полезной нагрузки. При составление предиката безопасности учитываются защитные механизмы. Предоставляется возможность обхода защитных механизмов: рандомизация адресного пространства (ASLR), защита исполнения данных (DEP), а также их совместное применение.

Для обхода ASLR используется подход на основе инструкций «трамплинов», описанный в подразделе 1.1.1. При обходе DEP, а также одновременной работы DEP и ASLR, используется подход на основе технологии ROP, описанный в подразделе 1.1.2. Предполагается, что код полезной нагрузки представляет собой ROP-цепочку в бинарном виде. Тем не менее, не для всех ситуаций достаточно ROP-цепочки, а только для случая, когда аварийное завершение происходит на инструкции возврата из функции (RET). В этом случае указатель стека уже указывает в контролируемую область и в этой области можно разместить цепочки, а потом передать управление. Для остальных случаев необходимо сначала переместить указатель стека в контролируемую область и потом приступать к размещению цепочки. Для этого можно использовать следующие типы гаджетов.

– Гаджет, сдвигающий указатель стека на константу (сложение или вычитание константы). При эксплуатации аварийного завершения во время выполнения инструкции вызова (CALL), стоит учитывать, что происходит вычитание из указателя стека значения 4 (для 32-ух разрядных систем) или 8 (для 64-ёх разрядных систем). Пример гаджета: *ADD ESP, 42; RET*.
– Гаджет, сдвигающий указатель стека на значение регистра (сложение или вычитание константы). При обходе ASLR необходимо, чтобы зна-

чение регистра не было помеченным. Для аварийных завершений на инструкции вызова также необходимо учесть изменение указателя стека самой инструкцией. Пример гаджета: *SUB ESP, EDI; RET.*

– Гаджет, загружающий значение регистра в указатель стека. При обходе ASLR необходимо, чтобы значение регистра не было помеченным. Пример гаджета: *MOV ESP, EAX; RET.*

Если в после выполнения гаджета, принадлежащему одному из этих типов, указатель стека указывает в помеченную область, то там размещается цепочка, а управление передаётся на этот гаджет. В результате произойдёт выполнение цепочки. Такие типы гаджетов будем называть *гаджеты-трамплины.*

Ниже приводятся алгоритмы построения предиката безопасности для типов аварийных завершений, описанных в подразделе 2.2.2.

Алгоритм 1 формирует предикат безопасности для исключения при выполнении инструкции возврата (RET), при условии, что значение адреса возврата из функции зависит от входных данных.

Предикат безопасности при отсутствии защитных механизмов в системе представляет собой совокупность всех возможных размещений кода полезной нагрузки. При каждом размещении в значение ячейки адреса возврата записывается адрес буфера с кодом полезной нагрузки. Для обхода ASLR адрес возврата перезаписывается адресом трамплина. Для обхода DEP, а также DEP и ASLR происходит размещение кода полезной нагрузки в символьном буфере, начало которого совпадает с адресом ячейки, содержащей адрес возврата. Код полезной нагрузки в этом случае представляет собой ROP-цепочку в бинарном виде.

Алгоритм 2 формирует предикат безопасности для исключения при выполнении инструкции возврата (RET), при условии, что значение указателя стека зависит от входных данных. В случае, когда помечено значение указателя стека, его необходимо исправить на корректное значение. Это значение соответствует указателю стека на момент вызова функции, в которой произошло переполнение буфера на стеке. При таком исправлении, пропадает возможность обойти ASLR, т.к. идёт явное указание адреса на стеке. В остальном, алгоритм схож с алгоритмом 1.

Алгоритм 3 формирует предикат безопасности для исключения при выполнении инструкции вызова или безусловного перехода, при условии, адрес передачи управления зависит от входных данных и находится в регистре.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная с адресом возврата retCell;

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

Если *!ASLR && !DEP* **тогда**

  Для каждого *символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (retCell.value == sbuf.address))$;

    конец

  конец

конец

иначе если *ASLR && !DEP* **тогда**

  Для каждого *трамплина trampoline и символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (retCell.value == trampoline))$;

    конец

  конец

конец

иначе

  символьный буфер, адрес которого совпадает адресом ячейки с адресом возврата retSBuf;

  $Sp = Sp \vee (retSBuf.value == shell)$;

конец

Алгоритм 1: Построение предиката безопасности для исключения при выполнении инструкции возврата, при условии, что значение возврата из функции помечено.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная с адресом возврата retCell; символьная переменная с указателем стека stackPointer; значение указателя стека после вызова функции svalue;

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

$Sp = Sp \wedge (stackPointer == svalue)$;

Если *!ASLR && !DEP* **тогда**

  Для каждого *символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (retCell.value == sbuf.address))$;

    конец

  конец

конец

иначе если *DEP && !ASLR* **тогда**

  символьный буфер, адрес которого совпадает адресом ячейки с адресом возврата retSBuf;

  $Sp = Sp \vee (retSBuf.value == shell)$;

конец

Алгоритм 2: Построение предиката безопасности для исключения при выполнении инструкции возврата, при условии, что значение указателя стека помечено.

Для обхода DEP, или одновременно работающих DEP и ASLR, применяются гаджеты-трамплины и размещение кода полезной нагрузки происходит в соответствующем этому гаджету буфере.

Алгоритм 4 формирует предикат безопасности для исключения при выполнении инструкции вызова или безусловного перехода. Адрес, по которому происходит передача управления находится в памяти. Регистры, формирующие выражение для адреса этой памяти зависят от входных данных.

Для осуществления передачи управления необходимо, чтобы адресное выражение операнда инструкции передачи управления указывало на помеченную область памяти. В связи с этим обход ASLR не предоставляется возможным.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная с адресом передачи управления callCell;

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

Если *!ASLR && !DEP* **тогда**

  Для каждого *символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (callCell.value == sbuf.address))$;

    конец

  конец

конец

иначе если *ASLR && !DEP* **тогда**

  Для каждого *трамплина trampoline и символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (callCell.value == trampoline))$;

    конец

  конец

конец

иначе

  Для каждого *гаджета-трамплина stackPivot и символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (callCell.value == stackPivot))$;

    конец

  конец

конец

Алгоритм 3: Построение предиката безопасности для исключения при выполнении инструкции вызова или безусловной передачи управления, при условии, что адрес передачи управления помечен и находится в регистре.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная для адресного выражения callMemAddr; размер адреса bitwidth

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

Если *!ASLR && !DEP* **тогда**

  Для каждого *символьного буфера sbuf* **выполнять**

    Если *sbuf.size - bitwidth >= shell.size* **тогда**

      $callCell = sbuf[0..bitwith - 1]$;

      $sbuf.address = sbuf.address + bitwidth$;

      $sbuf.value = sbuf[bitwidth..sbuf.size - 1].value$;

      $Sp = Sp \vee ((callMemAddr == callCell.address) \wedge$
      $(sbuf.value == shell) \wedge (callCell.value == sbuf.address))$;

    конец

  конец

конец

иначе если *DEP && !ASLR* **тогда**

  Для каждого *гаджета-трамплина stackPivot и символьного буфера sbuf* **выполнять**

    Если *sbuf.size - bitwidth >= shell.size* **тогда**

      $callCell = sbuf[0..bitwith - 1]$;

      $sbuf.address = sbuf.address + bitwidth$;

      $sbuf.value = sbuf[bitwidth..sbuf.size - 1].value$;

      $Sp = Sp \vee ((callMemAddr == callCell.address) \wedge$
      $(sbuf.value == shell) \wedge (callCell.value == stackPivot))$

    конец

  конец

конец

Алгоритм 4: Построение предиката безопасности для исключения при выполнении инструкции вызова или безусловной передачи управления, при условии, что адрес ячейки с адресом передачи управления зависит от входных данных.

Значение адреса, по которому осуществляется передача управления, располагается в начале буфера. Сразу после адреса размещается код полезной нагрузки. При обходе DEP адрес гаджета-трамплина размещается также в начале буфера. Результат выполнения этого гаджета перемещает указатель стека на значение равное адресу начала буфера плюс размерность адреса.

Алгоритм 5 формирует предикат безопасности для исключения при выполнении инструкции записи в память без учёта работы защитных механизмов.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная для адреса записи dest; символьная переменная для записываемого значения src; адрес ячейки с адресом возврата из функции retCellAddress;

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

Если *!ASLR && !DEP* **тогда**

  Для каждого *символьного буфера sbuf* **выполнять**

    Если $sbuf.size >= shell.size$ **тогда**

      $Sp = Sp \vee ((sbuf.value == shell) \wedge (dest == retCellAddress) \wedge (src == sbuf.address))$;

    конец

  конец

конец

Алгоритм 5: Построение предиката безопасности для исключения при выполнении инструкции записи в память без учёта работы защитных механизмов.

В качестве адреса записи выступает адрес возврата из функции, в которой произошло исключение. Записываемым значением является адрес буфера, в котором находится код полезной нагрузки. Таким образом, при возврате из функции произойдёт передача управления на внедрённый код.

Для программ, работающих под управлением ОС Linux, существует возможность обойти защитные механизмы DEP и ASLR, при соблюдении двух условий:

– исполняемый модуль программы не рандомизируется;

– секция GOT-доступна для записи.

Алгоритм построения предиката безопасности делится на два этапа. На первом этапе предикат безопасности представляет собой уравнение вида : $Sp : src == validAddress$, где src символьная переменная для записываемого значения, а validAddress значение адреса, по которому разрешён доступ на запись. Это делается для того, чтобы при следующем снятии трассы аварийного завершения не произошло. Второй этап представляет собой анализ полученной трассы. Предполагается, что после инструкции, на которой раньше происходило аварийное завершение, будет вызов библиотечный функции. Тогда для каждого вызова библиотечной функции будет производиться перезапись адреса из GOT-таблицы адресом гаджета-трамплина. Алгоритм 6 формирует предикат безопасности для исключения при выполнении инструкции записи в память при работе DEP и ASLR.

Исходные параметры: Множество символьных буферов; код полезной нагрузки shell; символьная переменная для адреса записи dest; символьная переменная для записываемого значения src; множество GOT-слотов;

Результат: Предикат безопасности Sp

$Sp = \varnothing$;

Если $ASLR$ && DEP **тогда**

  Для каждого *GOT-слота gotSlot* **выполнять**

    Для каждого *гаджета-трамплина stackPivot и символьного буфера sbuf* **выполнять**

      Если $sbuf.size >= shell.size$ **тогда**

        $Sp = Sp \vee ((sbuf.value == shell) \wedge (dest == gotSlot) \wedge (src == stackPivot))$;

      конец

    конец

  конец

конец

Алгоритм 6: Построение предиката безопасности для исключения при выполнении инструкции записи в память при работе DEP и ASLR.

Вычислительную сложность представленных алгоритмов (кроме алгоритма 6) можно оценить как $O(n)$, где n - количество символьных буферов в

памяти, на момент аварийного завершения. Для алгоритма 6 вычислительная сложность $O(n * m)$, где n - количество символьных буферов в памяти, на момент вызова библиотечной функции, а m количество вызовов библиотечных функций после инструкции, записи в память.

Глава 3. Реализация системы оценки эксплуатируемости дефектов

В этой главе рассматривается система оценки эксплуатируемости программных дефектов, схема архитектуры которой представлена на рис. 3.1. Си-

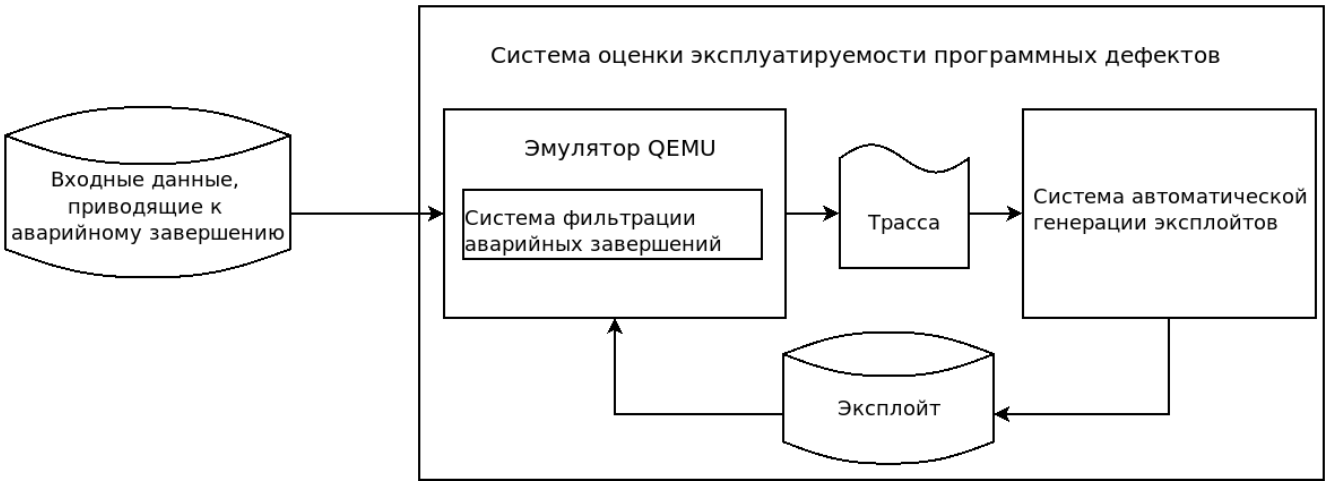

Рисунок 3.1 — Архитектура системы оценки эксплуатируемости программных дефектов.

стема состоит из нескольких программных компонентов, взаимодействие между которыми обеспечивается посредством программной инфраструктуры, реализованный на языке *python*. Эта инфраструктура также обеспечивает взаимодействие с пользователем. В состав системы оценки эксплуатируемости входят следующие инструменты:

– эмулятор QEMU;
– система предварительной фильтрации аварийных завершений;
– система автоматической генерации эксплойтов.

Полносистемный эмулятор QEMU используется для достижения нескольких целей. Одной из целей является получение трасс для автоматической генерации эксплойтов. Эмулятор поддерживает механизм детерминированного воспроизведения, который позволяет снизить влияние на работу гостевой системы, которое возникает в результате замедления во время трассировки. Также внутри эмулятора работает система предварительной фильтрации аварийных завершений (раздел 3.1). Система автоматической генерации эксплойтов представляет собой отдельный инструмент, который описан в (разделе 3.2). Проверка работоспособности эксплойта основана на запуске исследуемой программы в

эмуляторе QEMU. В разделе 3.3 приведён перечень технических ограничений текущей реализации системы.

3.1 Система предварительной фильтрации аварийных завершений

Система предварительной фильтрации обеспечивает отсев неэксплуатируемых дефектов и работает внутри гостевой системы в эмуляторе. Взаимодействие системы предварительной фильтрации и системы оценки эксплуатируемости программных дефектов осуществляется с помощью последовательного порта. Предварительная фильтрация реализована на базе динамического инструментатора бинарного кода *DynamoRIO* [45], который находится в открытом доступе и реализован на языке Си. Реализация системы представляет собой модуль-расширение для инструментатора.

Класс аварийного завершения описывается в виде структуры, которая имеет следующие поля:

- Группа аварийных завершений (эксплуатируемые или неэксплуатируемые);
- Короткое название;
- Полное название;
- Описание.

Первое поле указывает принадлежность класса к определённой группе аварийных завершений. Второе поле представляет короткий идентификатор класса. Третье поле – название класса аварийных завершений. Четвёртое поле содержит пояснение к ситуации, из-за которой возникло аварийное завершение.

Для определения аварийного завершения используется перехват сигналов (ОС Linux) или исключений (ОС Windows), а также отслеживание некоторых вызовов библиотечных функций. Перехватываются следующие сигналы:

- SIGSEGV;
- SIGABRT;
- SIGPFE.

Сигнал SIGSEGV соответствует исключению нарушения доступа к памяти. При обработке этого сигнала сперва проверяется возможность доступа к памяти, на которую указывает счётчик команд. Если счётчик команд указы-

вает на память, недоступную для чтения, то исключение соответствует классу нарушения доступа при попытке выполнить текущую инструкцию. В противном случае, если память доступна, происходит дизассемблирование инструкции по этому адресу и процесс классификации продолжается. Следующим шагом идёт определение типа инструкции:

- инструкция вызова;
- инструкция безусловной передачи управления;
- инструкция возврата;
- инструкция записи в память.
- инструкция чтения из памяти.

Если инструкция соответствует одному из перечисленных типов, то происходит дальнейший анализ её операндов и делается вывод о принадлежности к определённому классу аварийных завершений. В противном случае, считается, что класс аварийных завершений – исключение при попытке доступа к памяти.

Сигнал SIGABRT оповещает об принудительном завершении процесса. Если до получения сигнала был вызов функции *malloc_printerr* из библиотеки *libc*, то считается, что сработала защита менеджера памяти. В случае вызова функции *__stack_chk_fail* из библиотеки *libc*, подразумевается, что сработала «канарейка». Вызов функции *__chk_fail* из библиотеки *libc* свидетельствует о том, что сработала защита в безопасной функции.

Сигнал SIGPFE рапортует об исключении во время выполнении арифметической инструкции. Если инструкция, на которой произошло исключение инструкция деления, а делитель равен нуль, то считается, что произошло деление на ноль.

При анализе аварийных завершений программ, работающих под управлением ОС Windows, перехватываются следующие исключения:

- EXCEPTION_ACCESS_VIOLATION;
- EXCEPTION_INT_DIVIDE_BY_ZERO;
- EXCEPTION_FLT_DIVIDE_BY_ZERO;
- STATUS_HEAP_CORRUPTION.

Анализ исключения EXCEPTION_ACCESS_VIOLATION аналогичен анализу сигнала SIGSEGV. Обработка исключении EXCEPTION_INT_DIVIDE_BY_ZERO и EXCEPTION_FLT_DIVIDE_BY_ZERO схожа с обработкой сигнала SIGPFE. Исключение STATUS_HEAP_CORRUPTION свидетельствует о повреждении кучи.

Одной из особенностей реализации анализа аварийных завершений для программ, работающих под управлением ОС Windows, является то, что обработчик исключения после срабатывания «канарейки» сбрасывает все установленные ранее обработчики исключений и завершает программу. Это не позволяет перехватить искючение обычным образом. Для этого перехватывается вызов функции *UnhandledExceptionFilter*, и проверяется код исключения. Если код исключения равен STATUS_STACK_BUFFER_OVERRUN, то считается, что произошло срабатывание «канарейки».

3.2 Система автоматической генерации эксплойтов

Система автоматической генерации эксплойтов реализована на базе среды анализа бинарного кода, разрабатываемой в ИСП РАН. Среда имеет модульную архитектуру и обладает рядом преимуществ:

- обеспечивается независимость от реализации конкретной архитектуры процессора, с которого была получена трасса выполнения. В текущей версии поддерживаются следующие архитектуры: x86, x86-64, MIPS, ARM, PowerPC. Для этих архитектур реализована трансляция в машинно-независимое представление Pivot.
- поддерживается возможность единообразно анализировать различные операционные системы. На данный момент поддерживается анализ гостевых систем на базе операционных систем Windows и Linux.
- реализовано большое количество алгоритмов, позволяющие повысить уровень представления трассы машинных команд. Эта информация необходима для генерации эксплойта.

Для решения набора символьных уравнений и неравенств, полученных в результате работы системы автоматической генерации эксплойтов, используется SMT-решатель Z3, интегрированный в среду анализа бинарного кода в качестве библиотеки.

Предобработка трассы выполнения представляет собой повышение уровня представления трассы машинных команд и включает в себя восстановление высокоуровневых сущностей в трассе. Это обеспечивается уже реализованным набором алгоритмов в среде анализа:

- Алгоритм разметки процессов и потоков выполнения позволяет соотнести каждый шаг трассы с соответствующим ему процессом и потоком выполнения.
- Алгоритм восстановления информации о статической памяти позволяет получить карту статической памяти процессов, необходимую для алгоритма поиска модулей.
- Алгоритм поиска исполняемых модулей в трассе позволяет сопоставить инструкцию на каждом шаге трассы с модулем, которому она соответствует.
- Алгоритм выделения прерываний позволяет определить входы и выходы из прерываний.
- Алгоритм выделения вызовов в трассе позволяет обнаружить границы вызова, а именно шаги вызова и возврата функции.
- Алгоритм восстановления графа потока управления по трассе выполнения требуется для формирования статической информации о функциях.
- Алгоритм построения графа вызовов функций в трассе.
- Набор алгоритмов работы с моделями функций позволяет описывать функцию и её формальные параметры. Также эти алгоритмы дают возможность вычислять фактические значения описанных параметров для каждого конкретного вызова в трассе.
- Алгоритм восстановления буфера обеспечивает возможность получения значения памяти на заданном шаге трассы. Этот алгоритм требуется в связи с тем, что во время снятия трассы сохраняются только значения регистров. Сохранение значений памяти является чрезмерно затратным.
- Алгоритм поиска потенциальных аварийных завершений выделяет такие прерывания, из которых не было возврата к выполнению кода процесса, во время работы которого возникло это прерывание.

Процесс предобработки запускается в самом начале анализа трассы и проходит в автоматическом режиме.

В состав системы генерации эксплойтов входят:

- подсистема поиска точек получения входных данных;
- подсистема поиска точки аварийного завершения программы;
- подсистема построения предиката пути;
- подсистема построения предиката безопасности.

3.2.1 Подсистема поиска точек получения входных данных

Подсистема поиска точек получения входных данных основана на использовании отслеживания помеченных данных. Механизм отслеживания помеченных данных реализован в виде отдельной компоненты среды анализа, которая позволяет:

- анализировать различные зависимости, в том числе и зависимости по данным, а также отбирать те инструкции в трассе, которые участвовали в обработке помеченных данных;
- приостанавливать процесс анализа помеченных данных на заранее заданных шагах трассы для проведения дополнительного анализа;
- добавлять и удалять регистры, ячейки памяти из набора отслеживаемых элементов.

Отслеживание помеченных данных происходит в порядке увеличения шагов трассы. Для реализации дополнительного анализа используется механизм итераторов. Итератор содержит набор объектов, над которыми будет проводиться дополнительный анализ. Каждый объект имеет обязательный атрибут в виде шага трассы, используемый компонентой анализа помеченных данных для приостановки отслеживания помеченных данных, с целью проведения дополнительного анализа. Объекты упорядочены по возрастанию шагов трассы. При поиске точек получения входных данных используется итератор по вызовам функций работы с файлами и источникам получения входных данных в трассе. Источниками получения входных данных могут являться:

- шаг трассы с вызовом функции получения входных данных и буфер с входными данными;
- шаг трассы с точкой входа в программу и область стека, начиная с указателя стека и размером 10000 байт;
- шаг трассы с вызовом функции чтения файла с нулевым дескриптором.

Первый тип отвечает за получение данных из файлов и сети. Второй тип – аргументы командной строки и переменные окружения. Используется эвристика, что на точке входа в программу выше постеку в области размером 10000 байт располагаются аргументы командной строки и переменные окружения для исследуемой программы. В зависимости от вида объекта итератора происходят соответствующие действия. Отслеживания помеченных данных заканчивается,

когда в итераторе не останется ни одного источника получения входных данных. К этому момент уже сформированы точки получения входных данных, которые содержат всю необходимую информацию:

– шаг получения входных данных;
– буфер с входными данными;
– смещение в файле откуда были получены данные;
– идентификатор входного файла.

3.2.2 Подсистема поиска точки аварийного завершения

Подсистема поиска точки аварийного завершения также использует в своей основе анализ помеченных данных. Для работы подсистемы необходимы следующие итераторы:

– Итератор по точкам получения входных данных. Этот итератор используется для добавления буферов с входными в набор отслеживаемых элементов.
– Итератор по потенциальным аварийным завершениям проводит дополнительный анализ с целью обнаружения аварийного завершения программы. При успешном обнаружении аварийного завершения отслеживание помеченных данных прекращается и подсистема поиска точек получения входных данных и точки аварийного завершения заканчивает свою работу.

Итератор по точкам получения входных данных формируется на основе полученной информации от подсистемы поиска точек получения входных данных. На каждой точке получения входных данных происходит добавление буфера с входными данными во множество отслеживаемых элементов.

Итератор по потенциальным аварийным завершениям формируется на основе информации, предоставляемой алгоритмом поиска потенциальных аварийных завершений.

Результатом работы подсистемы является точка аварийного завершения, которая представляет собой:

– шаг трассы, на котором произошло аварийное завершение;
– тип аварийного завершения.

Тип аварийного завершения соответствует классификации, приведённой в подразделе 2.2.2.

3.2.3 Подсистема построения предиката пути

Подсистема построения предиката пути использует для своей работы механизм отслеживания помеченных данных. Набор итераторов, входящий в состав подсистемы выглядит следующим образом:

- Итератор по точкам получения входных данных.
- Итератор по дополнительными ограничениям, который позволяет описывать диапазоны допустимых значений памяти и регистров. Эти ограничения задаются аналитиком и применяются при достижении определённого шага трассы, линейного адреса инструкции или вызова функции.
- Итератор по моделям функций, с параметров которых необходимо снять пометки. Описание этих моделей и их параметров задаётся аналитиком.

Итератор по точкам получения входных данных необходим для добавления буферов с входным данными во множество помеченных буферов, а также обновления отображения элементов адресного пространства на символьные переменные. Итератор по дополнительным ограничениям реализует механизм борьбы с недостаточной помеченностью. Итератор по моделям функций, с параметров которых необходимо снять пометки позволяет бороться с избыточной помеченностью. Последние два итератора из вышеприведённого списка не являются обязательными, построение предиката пути может проходить в их отсутствии. Построение предиката пути происходит пока шаг анализа не станет больше или равен шагу трассы, на котором произошло аварийное завершение.

В подсистеме построения предиката пути каждая отобранная в результате отслеживания помеченных данных инструкция, обрабатывается анализатором инструкций. Процесс обработки инструкции разбивается на три стадии:

1. статическая трансляция инструкции в машинно-независимое представление;
2. интерпретация Pivot-представления;

3. трансляция Pivot-трассы в символьные формулы.

Статическая трансляция в Pivot-представление происходит средствами среды анализа бинарного кода. Для последующей интерпретации этого представления используется интерпретатор, выполняющий Pivot-инструкции на процессоре x86-64, и входящий в состав среды анализа. Трансляция полученной трассы Pivot-инструкций реализована непосредственно в самом анализаторе инструкций. Важной особенностью является возможность проведения дополнительного анализа после завершения каждой стадии обработки инструкции. Допускается разработка различных анализаторов инструкций, которые могли бы учитывать необходимую специфику анализа.

3.2.4 Подсистема построения предиката безопасности

Подсистема построения предиката безопасности реализует алгоритмы, описанные в разделе 2.4. Для формирования предикатов безопасности, преодолевающих защитные механизмы. Подсистема производит поиск инструкций трамплинов (обход ASLR) и гаджетов-трамплинов (обход DEP и ASLR).

Поиск трамплинов реализован на основе анализа исполняемых файлов, который доступен в среде анализа бинарного кода. Среда позволяет определить секции в файле, доступные для выполнения. В этих секциях происходит поиск опкодов инструкций трамплинов. В дальнейшем формируется список линейных адресов трамплинов, указывающих в помеченную область памяти на шаге аварийного завершения. Этот список трамплинов и соответствующих им буферов используется в алгоритмах построения предиката безопасности.

Гаджеты-трамплины задаются аналитиком в начале анализа. Аналитик указывает список Гаджеты-трамплинов, который содержит следующую информацию:

- линейный адрес начала гаджета;
- тип гаджета;
- дополнительная информация, специфичная для каждого типа.

В качестве дополнительной информации для гаджетов, сдвигающих указатель стека на константу являются: тип операции (сложение или вычитание) и размер смещения. Для гаджетов, сдвигающий указатель стека на значение реги-

стра указываются: тип операции (сложение или вычитание) и имя регистра. Для гаджета, загружающего значение регистра в указатель стека указывается имя загружаемого регистра. Основываясь на этой информации, формируется список адресов гаджетов-трамплинов и соответствующих им буферов с помеченными данными.

Созданный предикат безопасности объединяется с предикатом пути, после чего решается полученная система. В результате успешного решения получается эксплойт, который в последствии будет проверяться в эмуляторе.

3.3 Технические ограничения

Система предварительной фильтрации дефектов обладает незначительным ограничением. В системе отсутствует легковесный анализ помеченных данных, представленный в работах [24; 26]. Использование этой техники анализа позволяет расширить группу эксплуатируемых дефектов. В неё могли бы попасть аварийные завершения, возникшие в результате чтения из памяти, при условии, что считанное значение в последствии будет использовано для передачи управления. Это ограничение может быть преодолено посредством реализации соответствующего вида анализа в рамках существующей системы.

В системе автоматической генерации эксплойтов есть несколько небольших ограничений. Подсистема построения предиката пути использует алгоритм восстановления буфера для получения несимвольных значений. Алгоритм восстановления буфера, к сожалению, не всегда может восстановить значения. Это происходит в случае, если из ячейки памяти не было явного чтения из регистра, или в неё не была явная запись через регистр. В этом случае в качестве восстановленного значения выступает дополнительная символьная переменная без ограничений. Такой подход ослабляет ограничения в предикате пути. Это может привести к тому, что решение предиката пути не проведёт программу по тому пути, на котором оно было получено. Для решения данной проблемы можно использовать восстановление буферов памяти используя симулятор. В этом случае симулятор выставляет позицию в журнале событий равную шагу трассы, после чего происходит получение необходимого значения памяти.

В подсистеме построения предиката безопасности поиск трамплинов реализован только для архитектур *x86 и x86_64*. Для поддержки других архитектур необходимо определить шаблоны инструкций-трамплинов и сформировать соответствующие опкоды для поиска в исполняемом файле.

Глава 4. Применение

В данной главе рассматривается применение оценки эксплуатируемости программных дефектов для трёх различных ситуаций. В первом разделе продемонстрирована оценка эксплуатируемости программных дефектов, полученных в результате фаззинга. Во втором разделе показана работа метода на примерах дефектов, полученных из сторонних источников, таких как открытые базы дефектов и уязвимостей [60]. В третьем разделе приводятся результаты тестирования системы на модельных примерах.

4.1 Оценка эксплуатируемости результатов фаззинга

В качестве объекта фаззинга был выбран набор программ из директории */usr/bin* дистрибутива Debian 6.0.10 для платформы x86. Общие число программ: 8115. Во время фаззинга входные данные программы получали из аргументов командной строки. В результате фаззинга было получено 274 аварийных завершения, каждое из которых относится к различным программам. Результат предварительной фильтрации аварийных аварийных завершений представлен в табл. 4. После предварительной фильтрации было выявлено 13 эксплуатируемых дефектов. Остальные 261 были определены системой как неэксплуатируемые.

Для всех аварийных завершений, относящихся к группе эксплуатируемых дефектов применялась автоматическая генерация эксплойта. Все эксплуатируемые аварийные завершения были следствием исключения, возникшего во время выполнения инструкции возврата по контролируемому адресу. Для эксплуатируемых аварийных завершений проводилась генерация эксплойтов. В табл. 5 приведены результаты генерации эксплойтов. Для всех 13 эксплуатируемых удалось получить работоспособный эксплойт. Для 5-и аварийных завершений при генерации эксплойта были составлены коды полезной нагрузки в виде ROP-цепочек, способные обойти защиту DEP. При генерации эксплойта для одного из аварийных завершений использовалась цепочка, позволяющая обойти одновременно работающие защиты DEP и ASLR.

Таблица 4 — Результаты предварительной фильтрации аварийных аварийных завершений.

Группа аварийных завершений	Класс аварийного завершения	Количество аварийных завершений
Эксплуатируемые	Исключение при попытке выполнить текущую инструкцию	13
Неэксплуатируемые	Ошибка при работе с менеджером памяти	23
Неэксплуатируемые	Нарушение доступа к памяти	238

Таблица 5 — Результаты генерации эксплойтов.

Эксплойтов сгенерировано	Без учёта защит	Обход DEP	Обход DEP и ASLR
13	7	5	1

Таким образом, при оценки эксплуатируемости дефектов, полученных в резульате фаззинга набор программ из директории */usr/bin* дистрибутива Debian 6.0.10 удалось выявить 13 эксплуатируемых дефектов.

4.2 Оценка эксплуатируемости дефектов из доступных источников

Для всех рассмотренных примеров предварительная фильтрация обнаружила эксплуатируемые эксплуатируемые аварийные завершения. Проводилась

оценка эксплуатируемости дефектов для программ, работающих под управлением 32-ух разрядной Windows XP SP3. Все аварийные завершения возникали из-за исключения при выполнении инструкции возврата. Входные данные, приводящие к аварийному завершению программ, были получены при помощи скриптов с ресурса [60]. Ниже приводится список исследуемых программ:

- AudioCoder [61]. Источник входных данных: файл.
- CoolPlayer [62]. Источник входных данных: файл.
- VuPlayer [63]. Источник входных данных: файл.
- Pcman [64]. Источник входных данных: сеть.

Для программ AudioCoder и VuPlayer удалось получить эксплойты, способные обойти защиту DEP. Остальные эксплойты были сгенерированы без учёта защитных механизмов.

Для программы *pbs-server* из пакета *torque-server* [65] удалось сгенерировать эксплойт, способный преодолеть защиты DEP и ASLR. Программа работа под управлением 32-ух разрядной ОС Debian 8.3.0.

Аварийное завершение программы *nullhttpd* [66] произошло в результате исключения во время инструкции записи в память. Программа работа под управлением 32-ух разрядной ОС Debian 8.3.0. Был получен работоспособный эксплойт, не учитывающий работу защитных механизмов.

Тестирование оценки эксплуатируемости для программ, которые работают под управлением 64-ёх разрядной ОС Debian 8.3.0, проводилось на следующих примерах программ: *mkfs.jfs, faad, dvips*. Для всех экземпляров был получен эксплойт, не учитывающий работу защитных механизмов.

Исходя из приведённого выше тестирования оценки эксплуатируемости программных дефектов, можно заключить, что разработанный метод удовлетворяет заявленным требованиям. Метод позволяет получать эксплойты, способные обходить защитные механизмы. Метод не зависит от операционной системы и архитектуры процессора, на котором выполняется исследуемая программа.

4.3 Оценка эксплуатируемости программ из DARPA Cyber Grand Challenge

Набор содержит 241 исполняемый файл. Исходный набор примеров предусматривал их запуск в специально разработанной для конкурса операционной системе на базе Linux, в которой отсутствуют защиты DEP и ASLR. После проведения соревнований, тесты были перенесены на различные платформы [67], в том числе Linux.

В качестве операционной системы для запуска примеров был выбран 32-ух разрядный Debian 8.3.0. Для 11 программ были сформированы входные данные, приводящие программу к аварийному завершению. В результате предварительной фильтрации 8 аварийных завершений были оценены как эксплуатируемые, а для 6 из них удалось получить работоспособный эксплойт. Три неэксплуатируемых аварийных завершения программ (*FablesReport, Diary_Parser, greeter*) соответствуют классу нарушения доступа к памяти. Эксплуатируемые аварийные завершения обнаружены для следующих программ из тестового набора:

- Bloomy_Sunday;
- Charter;
- Movie_Rental_Service;
- Multi_User_Calendar;
- Palindrome;
- PKK_Steganography;
- Sample_Shipgame;
- ValveChecks.

Аварийное завершение программы *Charter* относиться классу исключения при попытке выполнить запись в память. В результате генерации эксплойта было установлено, что записываемое значение не зависит от входных данных, и, таким образом, эксплойт не был сгенерирован.

Все остальные эксплуатируемые аварийные завершения относятся к классу исключения при попытке выполнить текущую инструкцию. В программе *Bloomy_Sunday* присутствует уязвимость переполнения буфера на стеке. Для этой уязвимости не удалось получить работоспособный эксплойт. На этапе проверки эксплойт не продемонстрировал желаемого поведения. Такая ситуация может быть связана с тем, что во время построения предиката пути не бы-

ли учтены все зависимости (недостаточная помеченность). Для решения этой проблемы можно прибегнуть к анализу трассы с неработающим эксплойтом, с целью понять какие ограничения были упущены. После этого следует перегенерировать эксплойт с учётом этих ограничений.

Для программ *Movie_Rental_Service, Multi_User_Calendar, Palindrome, PKK_Steganography, Sample_Shipgame, ValveChecks* удалось получить работоспособные эксплойты. В программе *Movie_Rental_Service* была проэксплуатирована уязвимость использования памяти после освобождения. В остальных программах присутствовала уязвимость переполнения буфера на стеке.

Стоит отдельно отметить эксплуатацию программы *PKK_Steganography*. Входные данные для этой программы представляют собой файл-изображение со специальным форматом. В этом файле стеганографическим методом наименьшего значащего бита записано текстовое сообщение, которое при обработке изображения декодируется в буфер на стеке фиксированного размера. Переполнение этого буфера является эксплуатируемой уязвимостью. Таким образом, результирующий эксплойт содержит код полезной нагрузки, который закодирован методами стеганографии.

4.4 Оценка эксплуатируемости модельных примеров

На модельных примерах, исходные тексты которых, приведены в **Приложении А**, тестировались алгоритмы формирования предикатов безопасности, описанные в разделе 2.4. Для всех поддерживаемых предикатов безопасности удалось получить работоспособные эксплойты. Примеры представляют собой 32-ух разрядные программы, написанные на языке Си. Запуск этих программ происходил в ОС Debian 8.3.0.

Программа, приведённая в листинге А.1, позволяет провести эксплуатацию исключения при выполнении инструкции возврата. Для обхода защиты ASLR в текст программы вставлена инструкция-трамплин.

В листинге А.2 содержится программа, которая даёт возможность эксплуатировать исключение при выполнении инструкции возврата при работе DEP и ASLR. Программа получает входные данные из файла в бинарном виде. Это

упрощает формирование ROP-цепочки, т.к. нет ограничения на нулевые символы. Исполняемый файл слинкован с набором гаджетов для цепочки.

Листинг A.3 демонстрирует программу для эксплуатации исключения при выполнении инструкции возврата с помеченным указателем стека.

Показанная в листинге A.4 программа, позволяет эксплуатировать исключение при выполнении инструкции вызова. Значение адреса передачи управления содержится в регистре. Исключение возникает из-за переполнения буфера на стеке. Указатель на буфер, из которого происходит копирование на стек, содержится в регистре. Таким образом, для обхода ASLR можно воспользоваться трамплинами, которые есть в исполняемом файле.

Листинг A.5 содержит программу, позволяющую эксплуатировать исключение при выполнении инструкции вызова с учётом работы DEP и ASLR. Значение адреса передачи управления содержится в регистре. Исполняемый файл слинкован с набором гаджетов для цепочки.

В листинге A.6 приводится пример программы, которая даёт возможность провести эксплуатации исключения при выполнении инструкции вызова с учётом работы DEP. Исключение возникает из-за переполнения буфера, в результате чего перезаписывается указатель на массив указателей на функцию.

Листинг A.7 представляет собой программу, позволяющую эксплуатировать исключение при записи в память с учётом работы DEP и ASLR. Исключение возникает из-за перезаписи в цикле указателя *ptr* и последующей записи по этому указателю. Для обхода защит используется ROP-цепочка, а перезаписывается GOT-слот для функции *free* адресом гаджета-трамплина.

Заключение

В диссертации описан подход к оценке эксплуатируемости программных дефектов. Подход представляет собой совокупность двух разработанных методов: метода предварительной фильтрации аварийных завершений и метода автоматической генерации эксплойтов.

На основе предложенных методов автором была реализована система оценки эксплуатируемости программных дефектов. Реализованная система применялась к оценке эксплуатируемости дефектов, полученных в результате фазинга, дефектов из открытых источников, а также на модельных примерах. Результаты применения показали состоятельность разработанных методов.

Основные возможности метода предварительной фильтрации аварийных завершений, отличающих его от близких подходов, таковы:

1. предлагаемая классификация аварийных завершений позволяет эффективно применять метод совместно с методом автоматической генерации эксплойтов;
2. разработанный метод не зависит от архитектуры процессора или от операционной системы, на которых выполняется исследуемая программа.

Среди особенностей метода автоматической генерации эксплойтов можно выделить следующие:

1. метод позволяет получать эксплойты, способные обходить защитные механизмы DEP и ASLR;
2. метод обладает более широким набором алгоритмов построения предикатов безопасности, чем схожие подходы;
3. разработанный метод не зависит от архитектуры процессора или от операционной системы, на которых выполняется исследуемая программа.

В завершении приведём список важный направлений дальнейших работ по теме диссертации.

1. Снятие технических ограничений, перечисленных в разделе 3.3.
2. Проведение предварительной фильтрации аварийных завершений не только для аварийных завершений, полученных на исходном наборе входных данных, но и на наборах, сформированных из исходного путём удаления некоторого количества входных данных. Это позволит обна-

руживать новые классы аварийных завершений, а также уменьшить размер входных данных для эксплуатируемых дефектов.

3. Расширение набора алгоритмов построения предикатов безопасности. Эти алгоритмы будут учитывать специфику други распространённых процессорных архитектур (ARM, MIPS, PowerPC). А также особенности работы программ на уровне ядра операционных систем.

Список литературы

1. *Падарян В.А., Каушан В.В., Федотов А.Н.* Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке // *Труды Института системного программирования РАН.* — 2014. — Vol. 26, no. 3.

2. *Федотов А.Н.* Метод оценки эксплуатируемости программных дефектов // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 4.

3. *Padaryan V.A., Kaushan V.V., Fedotov A.N.* Automated exploit generation for stack buffer overflow vulnerabilities // *Programming and Computer Software.* — 2015. — Т. 41, № 6. — С. 373–380.

4. Оценка критичности программных дефектов в условиях работы современных защитных механизмов / А.Н. Федотов, В.А. Падарян, В.В. Каушан и др. // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 5. — С. 73–92.

5. *Каушан В.В., Федотов А.Н.* Развитие технологии генерации эксплойтов на основе анализа бинарного кода // Материалы 24-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации». — 2015.

6. The shellcoder's handbook: discovering and exploiting security holes / C. Anley, J. Heasman, F. Lindner, G. Richarte. — John Wiley & Sons, 2011. — 61 с.

7. *Durden Tyler.* Bypassing PaX ASLR protection. — 2002. — URL: http://phrack.org/issues/59/9.html#article (дата обращения: 10.05.2017).

8. *Team Corelan.* Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR. — 2009. — URL: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/ (дата обращения: 11.05.2017).

9. *Daniele Del Basso Luca Argento, Patrizio Boschi.* NX bit - A hardware-enforced BOF protection. — 2004. — URL: http://index-of.es/EBooks/NX-bit.pdf (дата обращения: 12.05.2017).

10. *Nergal.* The advanced return-into-lib(c) exploits: PaX case study. — 2001. — URL: http://www.phrack.org/issues/58/4.html#article (дата обращения: 12.05.2017).

11. Return-oriented programming: Systems, languages, and applications / Ryan Roemer, Erik Buchanan, Hovav Shacham, Stefan Savage // *ACM Transactions on Information and System Security (TISSEC).* — 2012. — Vol. 15, no. 1. — P. 2.

12. *Schwartz Edward J, Avgerinos Thanassis, Brumley David.* Q: Exploit Hardening Made Easy. // USENIX Security Symposium. — 2011. — Pp. 25–41.

13. *One Aleph.* Smashing The Stack For Fun And Profit. — 1996. — URL: http://phrack.org/issues/49/14.html#article (дата обращения: 12.05.2017).

14. *gera.* Advances in format string exploitation. — 2002. — URL: http://phrack.org/issues/59/7.html#article (дата обращения: 12.05.2017).

15. Метод поиска уязвимости форматной строки / И.А. Вахрушев, В.В. Каушан, В.А. Падарян, А.Н. Федотов // *Труды Института системного программирования РАН.* — 2015. — Т. 27, № 4.

16. CWE-123. — URL: https://cwe.mitre.org/data/definitions/123.html (дата обращения: 12.05.2017).

17. Once upon a free(). — 2001. — URL: http://phrack.org/issues/57/9.html#article (дата обращения: 12.05.2017).

18. *Thomas Reji, Reddy Bhasker.* Dynamic Linking in Linux and Windows, part one. — 2006. — URL: https://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one (дата обращения: 13.05.2017).

19. *Thomas Reji, Reddy Bhasker.* Dynamic Linking in Linux and Windows, part two. — 2006. — URL: https://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-two (дата обращения: 13.05.2017).

20. Non-Control-Data Attacks Are Realistic Threats. / Shuo Chen, Jun Xu, Emre Can Sezer et al. // Usenix Security. — Vol. 5. — 2005.

21. *Team Corelan.* Exploit writing tutorial part 3 : SEH Based Exploits. — 2009. — URL: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/ (дата обращения: 13.05.2017).

22. *Miller Matt.* Preventing the exploitation of seh overwrites // *Uninformed Journal.* — 2006. — Vol. 5.

23. Hardening. — URL: https://wiki.debian.org/Hardening (дата обращения: 14.05.2017).

24. Crash Course: Analyze Crashes to Find Security Vulnerabilities in Your Apps / Adel Abouchaev, Damian Hasse, Scott Lambert, Greg Wroblewski // *MSDN Magazine.* — 2007.

25. gdb exploitable plugin. — URL: https://github.com/jfoote/exploitable (дата обращения: 17.05.2017).

26. Automated Crash Filtering for ARM Binary Programs / Ki-Jin Eom, Joon-Young Paik, Seong-Kyun Mok et al. // Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual. — Vol. 2. — 2015. — Pp. 478–483.

27. *Avgerinos T., Cha S.K., Brumley D.* AEG: Automatic Exploit Generation // In Proceedings of the Network and Distributed System Security Symposium. — 2011. — Pp. 283–300.

28. Automatic exploit generation / Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. // *Communications of the ACM.* — 2014. — Vol. 57, no. 2. — Pp. 74–84.

29. Unleashing mayhem on binary code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Security and Privacy (SP), 2012 IEEE Symposium on. — 2012. — Pp. 380–394.

30. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations / Shih-Kun Huang, Min-Hsiang Huang,

Po-Yen Huang et al. // Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. — 2012. — Pp. 78–87.

31. The Art of War: Offensive Techniques in Binary Analysis / Y Shoshitaishvili, R Wang, C Salls et al. // IEEE Symposium on Security and Privacy (S&P). — 2016.

32. Windbg. — URL: https://developer.microsoft.com/en-us/windows/hardware/download-windbg (дата обращения: 17.05.2017).

33. *Newsome James, Song Dawn.* Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. — 2005.

34. !exploitable. — URL: https://msecdbg.codeplex.com/ (дата обращения: 20.05.2017).

35. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/ (дата обращения: 19.05.2017).

36. REIL-The Reverse Engineering Intermediate Language. — URL: http://www.zynamics.com/binnavi/manual/html/reil_language.htm (дата обращения: 21.05.2017).

37. Zynamics BinNavi. — URL: http://www.zynamics.com/binnavi.html (дата обращения: 21.05.2017).

38. CrashFilter. — URL: https://github.com/killbug2004/CrashFilter (дата обращения: 21.05.2017).

39. *Schwartz Edward J, Avgerinos Thanassis, Brumley David.* All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) // Security and privacy (SP), 2010 IEEE symposium on. — 2010. — Pp. 317–331.

40. *Heelan Sean.* Automatic generation of control flow hijacking exploits for software vulnerabilities. — 2009.

41. *Lattner Chris, Adve Vikram.* LLVM: A compilation framework for lifelong program analysis & transformation // Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. — 2004. — P. 75.

42. *Chipounov Vitaly, Kuznetsov Volodymyr, Candea George.* S2E: A platform for in-vivo multi-path analysis of software systems // *ACM SIGPLAN Notices.* — 2011. — Vol. 46, no. 3. — Pp. 265–278.

43. CRAX. — URL: https://github.com/SQLab/CRAX (дата обращения: 23.05.2017).

44. REX. — URL: https://github.com/shellphish/rex/tree/master/rex (дата обращения: 23.05.2017).

45. DynamoRIO. — URL: http://www.dynamorio.org/ (дата обращения: 01.08.2017).

46. CWE-415. — URL: https://cwe.mitre.org/data/definitions/415.html (дата обращения: 12.05.2017).

47. CWE-122. — URL: https://cwe.mitre.org/data/definitions/122.html (дата обращения: 12.05.2017).

48. Doug Lea allocator. — URL: http://g.oswego.edu/dl/html/malloc.html (дата обращения: 01.07.2017).

49. CWE-121. — URL: https://cwe.mitre.org/data/definitions/121.html (дата обращения: 12.05.2017).

50. CWE-416. — URL: https://cwe.mitre.org/data/definitions/416.html (дата обращения: 12.05.2017).

51. *Тихонов А.Ю., Аветисян А.И.* Комбинированный (статический и динамический) анализ бинарного кода // *Труды Института системного программирования РАН.* — 2012. — Vol. 22.

52. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода / В.А. Падарян, А.И. Гетьман, М.А. Соловьев et al. // *Труды Института системного программирования РАН.* — 2014. — Vol. 26, no. 1.

53. *Тихонов А.Ю., Аветисян А.И., Падарян В.А.* Методика извлечения алгоритма из бинарного кода на основе динамического анализа // *Проблемы информационной безопасности. Компьютерные системы.* — 2008. — no. 3. — Pp. 73–79.

54. *Довгалюк П.М., Фурсова Н.И., Дмитриев Д.С.* Перспективы применения детерминированного воспроизведения работы виртуальной машины при решении задач компьютерной безопасности // *Системы высокой доступности.* — 2013. — Vol. 9, no. 3. — Pp. 046–050.

55. Применение программных эмуляторов в задачах анализа бинарного кода / В.А. Довгалюк, П.М.and Макаров, В.А. Падарян, М.С. Романеев, Н.И. Фурсова // *Труды Института системного программирования РАН.* — 2014. — Vol. 26, no. 1.

56. *Тихонов А.Ю., Падарян В.А.* Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». — 2009. — P. 131.

57. SMT-LIB. — URL: http://smtlib.cs.uiowa.edu/ (дата обращения: 03.08.2017).

58. *Падарян В.А., Соловьев М.А., Кононов А.И.* Моделирование операционной семантики машинных инструкций // *Труды Института системного программирования РАН.* — 2010. — Vol. 19.

59. *Соловьев М.А.* Восстановление алгоритма по набору бинарных трасс: дис. ... канд. физ.-мат. наук: 05.13.11; [Место защиты: Федеральное государственное бюджетное учреждение науки Институт системного программирования РАН]. — 2013.

60. Exploit database. — URL: https://www.exploit-db.com/ (дата обращения: 08.08.2017).

61. AudioCoder example. — URL: https://www.exploit-db.com/exploits/26448/ (дата обращения: 08.08.2017).

62. CoolPlayer example. — URL: https://www.exploit-db.com/exploits/29613/ (дата обращения: 08.08.2017).

63. VuPlayer example. — URL: https://www.exploit-db.com/exploits/40018/ (дата обращения: 08.08.2017).

64. pcman example. — URL: https://www.exploit-db.com/exploits/38003/ (дата обращения: 08.08.2017).

65. torque example. — URL: https://www.exploit-db.com/exploits/33554/ (дата обращения: 08.08.2017).

66. Null httpd example. — URL: https://www.exploit-db.com/exploits/21818/ (дата обращения: 08.08.2017).

67. Multi OS DARPA-cgc tests. — URL: https://github.com/trailofbits/cb-multios (дата обращения: 05.09.2017).

Список рисунков

Список таблиц

Приложение А

Исходные тексты модельных примеров

В данном приложении приводятся исходные тексты модельных примеров на языке Си. Данные примеры использовались при тестировании системы оценки эксплуатируемости программных дефектов.

Листинг А.1 Программа для эксплуатации исключения при выполнении инструкции возврата по помеченному адресу при работе защиты ASLR.

```
#include <stdio.h>
#include <string.h>

void foo(char* ptr)
{
  char buf[100];
  strcpy(buf, ptr);
}

int main(int argc, char* argv[])
{
  foo(argv[1]);
  return 0;
}
void special()
{
  asm("call %esp");
}
```

Листинг А.2 Программа для эксплуатации исключения при выполнении инструкции возврата по помеченному адресу при работе защиты DEP и ASLR.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void
foo(char *fname)
{
  int c;
  int fd = open64(fname, O_RDONLY);
  char buf[100];
  read(fd, buf, 200);
}

int
main(int argc, char *argv[])
{
  foo(argv[1]);
  return 0;
}
```

Листинг А.3 Программа для эксплуатации исключения при выполнении инструкции возврата с помеченным указателем стека.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
  char buf[100];
  strcpy(buf, argv[1]);
  return 0;
}
```

Листинг А.4 Программа для эксплуатации исключения при выполнении инструкции вызова с учётом ASLR. Адрес назначения содержится в регистре.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int goodfunc(const char *str)
{
  printf("%s",str);
  return 0;
}
void foo(char *ptr)
{
  int (*funcptr)(const char *str);
  char buf[128];
  register char* heapArr;
  heapArr = malloc(256);
  strncpy(heapArr, ptr, 256);
  funcptr = (int (*)(const char *str))goodfunc;
  strncpy(buf, heapArr, strlen(heapArr));
  (void)(*funcptr)(buf);
  free(heapArr);
}

int main(int argc, char **argv)
{
  foo(argv[1]);
  return 0;
}

void special()
{
  asm("call %esp");
  asm("call %eax");
  asm("call %ebx");
  asm("call %ecx");
  asm("call %edx");
  asm("call %edi");
  asm("call %esi");
}
```

Листинг А.5 Программа для эксплуатации исключения при выполнении инструкции вызова при работе DEP и ASLR. Адрес назначения содержится в регистре.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int goodfunc(const char *str)
{
printf("%s",str);
return 0;
}

int main(int argc, char **argv)
{
  int (*funcptr)(const char *str);
  char buf[128];

  funcptr = (int (*)(const char *str))goodfunc;
  int fd = open64(argv[1], O_RDONLY);
  read(fd, buf, 200);
  (void)(*funcptr)(buf);
  return 0;
}
```

Листинг А.6 Программа для эксплуатации исключения при выполнении инструкции вызова при работе DEP. Адрес назначения содержится в памяти.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void fun1()
{

}

void fun2()
{

}

void fun3()
{

}

int main(int argc, char **argv)
{
  void (*func_ptr[3])();
  func_ptr[0] = fun1;
  func_ptr[1] = fun2;
  func_ptr[2] = fun3;
  char buf[128];
  int c = atoi(argv[1]);
  int fd = open64(argv[2], O_RDONLY);
  read(fd, buf, 200);
  func_ptr[c]();
  return 0;
}
```

Листинг А.7 Программа для эксплуатации исключения при выполнении инструкции записи в память с учётом работы DEP и ASLR.

```
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

void func(int *userInput)
{
int *ptr;
int arr[32];
int i;

ptr = &arr[31];

for (i = 0; i <=32; i++)
arr[i] = userInput[i];

*ptr = arr[0];
}

int main(int argc, char *argv[])
{
int res, fd = -1;
register int *heapArr = NULL;
fd = open64(argv[1], O_RDONLY);

heapArr = malloc(64*sizeof(int));
res = read(fd, heapArr, 64*sizeof(int));
func(heapArr);
free(heapArr);
return 0;
}
```