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»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2017

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В.П. Иванникова Российской академии наук.

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

Официальные оппоненты: **Ильин Вячеслав Анатольевич,**
доктор физико-математических наук,
начальник отдела Курчатовского комплекса НБИКС-технологий Национального исследовательского центра «Курчатовский институт»

Козачок Александр Васильевич,
кандидат технических наук, сотрудник ФГКВОУ ВО «Академия Федеральной службы охраны Российской Федерации»

Ведущая организация: Межведомственный суперкомпьютерный центр Российской академии наук – филиал Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Защита состоится 21 декабря 2017 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В.П. Иванникова Российской академии наук по адресу: 109004, Москва, ул. Александра Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования им. В.П. Иванникова Российской академии наук.

Автореферат разослан «___»_____ 2017 г.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физико-математических
наук Зеленов С.В.

Общая характеристика работы

Актуальность темы. 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 интерпретация 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 обнаруживать 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, четырех глав, заключения и одного приложения. Полный объем диссертации составляет 98 страниц, включая 17 рисунков и 5 таблиц. Список литературы содержит 67 наименований.

Краткое содержание работы

Во **введении** представлена цель диссертационной работы, обосновывается её актуальность. Формулируются основные результаты работы. Рассматриваются возможности практического применения предложенных методов и реализованных инструментов. Перечисляются основные печатные работы по теме диссертации.

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

В разделе 1.1 описаны современные механизмы защиты, препятствующие эксплуатации уязвимостей, которые можно разделить на две категории: защитные механизмы операционной системы и защитные механизмы, предоставляемые компилятором. К первой категории относятся рандомизация 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.

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.

В разделе 1.2 рассматриваются существующие инструменты, позволяющие оценить эксплуатируемость программных дефектов: средства анализа аварийных завершений программ и системы автоматической генерации эксплойтов. К первой группе относятся такие инструменты как: !exploitable, gdb exploitable plugin. Из инструментов отнесённых ко второй группе стоит выделить системы AEG, MAYHEM, REX и CRAX.

Из проведённого обзора вытекают следующие выводы. Большинство рассмотренных инструментов могут быть использованы для оценки эксплуатируемости программных дефектов.

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 пользовательски-

ми программами. Также эти инструменты находятся в закрытом доступе, а в публикациях нет информации о том, как учитывается работа современных защитных механизмов при генерации эксплойта. Система CRAX, основанная на полносистемном символьном выполнении, представленном в системе S2E, позволяет генерировать эксплойты при наличии входных данных, приводящих к аварийному завершению программы. В публикации по системе CRAX предлагаются подходы к генерации эксплойтов для ситуаций, когда:

- 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 находится в форматной строке во время вызова функции работы с форматной строкой.

Описываются способы преодоления защитных механизмов, таких как: защита выполнения данных (DEP) и рандомизация адресного пространства (ASLR). К сожалению, реализация системы, представленная в открытых источниках, 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.

Вторая глава посвящена описанию предлагаемого метода оценки эксплуатируемости программных дефектов и, в частности, метода автоматической генерации эксплойтов. Схема метода оценки эксплуатируемости представлена на рис. 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 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.

В случае, если аварийное завершение классифицируется как эксплуатируемое, начинается второй этап – автоматическая генерация эксплойта. Предлагаемый подход к автоматической генерации эксплойта основан на

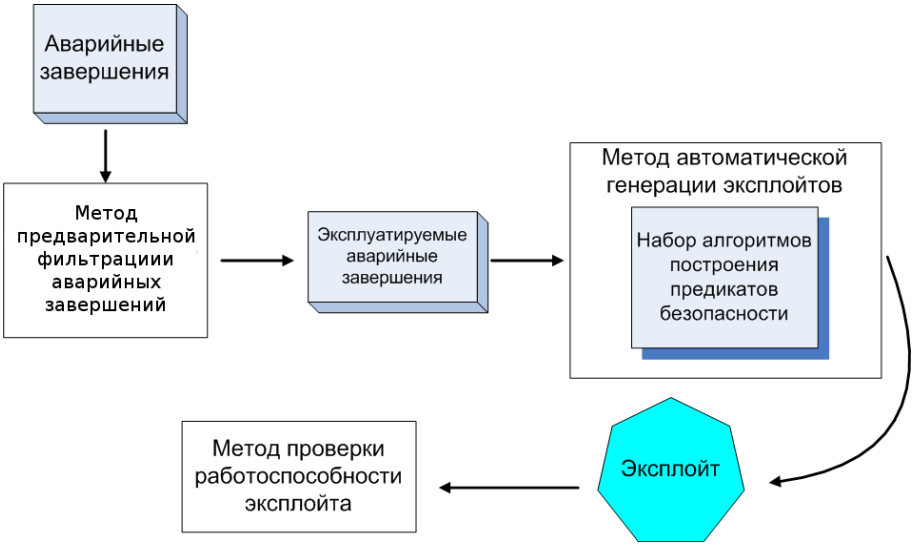

Рис. 1 — Схема метода оценки эксплуатируемости программных дефектов

символьной интерпретации 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. 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. 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.

В разделе 2.1 рассматривается применяемый подход предварительной фильтрации аварийных завершений. Метод основан на анализе аварийных завершений, который реализован в инструментах !exploitable и gdb

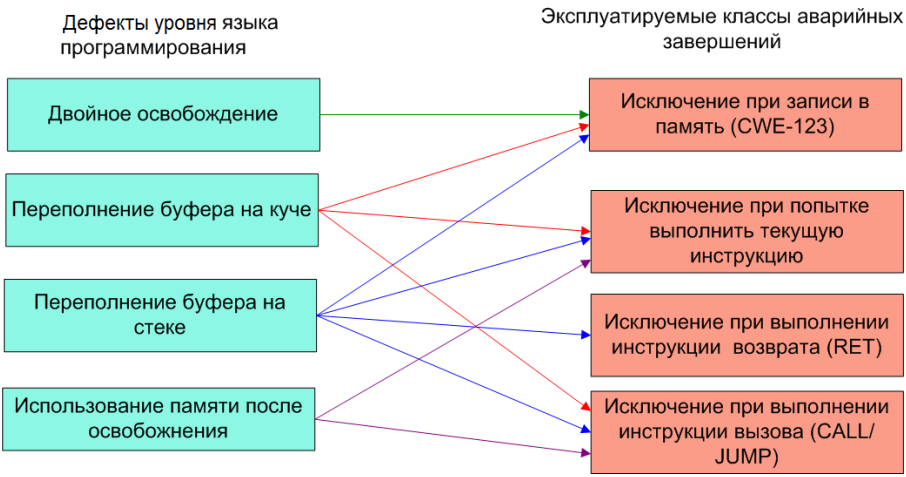

Рис. 2 — Соответствие между дефектами на уровне языка программирования и эксплуатируемыми классами аварийных завершений программы.

exploitable plugin. При анализе аварийных завершений изучается 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. В предлагаемом подходе представлены 18 классов аварийных завершений, которые в свою очередь относятся к двум группам:

– эксплуатируемые аварийные завершения (4 класса);
– неэксплуатируемые аварийные завершения(14 классов).

К эксплуатируемым аварийным завершениям относятся те аварийные завершения, эксплуатация которых наиболее вероятна. Именно для них будет проводится автоматическая генерация эксплойтов в дальнейшем. На рис. 2 представлено соответствие между дефектами на уровне языка программирования и эксплуатируемыми классами аварийных завершений программы.

Исключение во время записи в память, при условии, что адрес записи в память и записываемое значение находится под контролем атакующего, может быть проэксплуатировано. Эта ситуация описана в перечне слабостей программного обеспечения (*Common weakness enumeration*) под номером 123 и имеет высокую вероятность успешной эксплуатации. Появляется возможность перезаписать область памяти, которая впоследствии будет использована для передачи управления, адресом кода полезной нагрузки. Этот класс аварийных завершений может быть следствием двойного освобождения памяти или переполнения буфера на куче в некоторых менеджерах памяти, например, *Doug Lea allocator*. А также в случае пе-

резаписи указателя при переполнении буфера на стеке или куче. Стоит отметить, 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 (RET), 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. Эта ситуация позволяет передать управление на код полезной нагрузки.

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. Если значение, по которому должна осуществиться передача управления доступно для чтения, то после передачи управления может возникнуть исключение при попытке выполнить текущую инструкцию. Эта ситуация также даёт возможность выполнить произвольный код.

Таким образом, 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.

К неэкслуатируемым аварийным завершениям относятся такие классы как: разыменование нулевого указателя, срабатывание защиты стека и т.д. Для таких аварийных завершений автоматическая генерация эксплойта проводится не будет. Полный список всех классов аварийных завершений, а также их принадлежность к группам приводятся в диссертации.

В разделе 2.2 представлен метод автоматической генерации по аварийным завершениям программы. Метод учитывает работу современных защитных механизмов. На рис. 3 представлена схема метода автоматической генерации эксплойтов. В качестве первого этапа происходит получение и обработка трассы выполнения программы. Во время работы исследуемой программы происходит аварийное завершение. Трасса представляет собой последовательный набор выполненных процессором инструкций и значении всех регистров на момент выполнения каждой инструкции. Номер выполненной инструкции называется шагом трассы. Во время обработки трассы формируется вся необходимая информация для дальнейшего анализа: разметка трассы на процессы и потоки выполнения, разметка исполняемых модулей в трассе, выделение функций и построение стека вызовов.

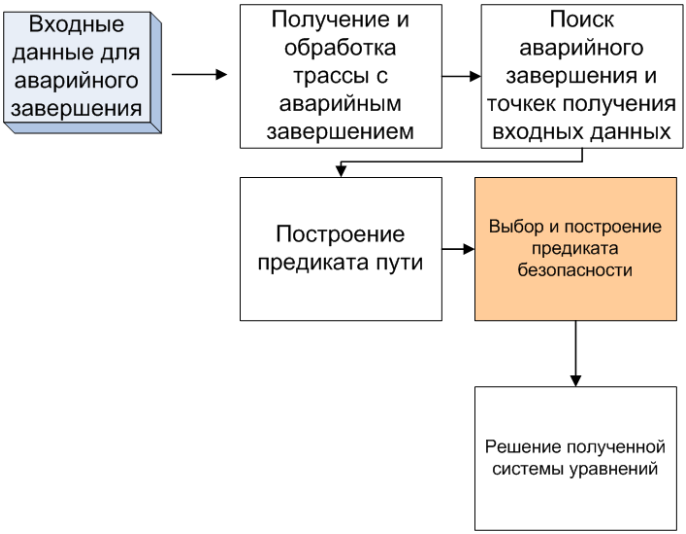

Рис. 3 — Схема метода автоматической генерации эксплойтов

Следующим этапом идёт поиск аварийного завершения, а также 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. Аналитиком задаются функции работы с файлами, а также функции, которые принимают данные, например, функция получения сетевого пакета. Использование такого подхода возможно благодаря полносистемному отслеживанию помеченных данных. Помимо функций, аналитик также задаёт имена файлов, в которых содержаться входные данные. С помощью этой информации определяются соответствующие функции чтения данных из файлов, а также буферы в памяти, в которые считывались эти данные. Затем эти буферы помечаются и выполняется распространение пометок. Когда помеченные данные попадают в функции получения данных, формируются точки получения входных данных в исследуемой программе.

Аварийное завершение программы происходит в следствии возникновения исключительной ситуации, вызванной срабатыванием 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 прерванной инструкции. Отфильтрованные таким образом прерывания рассматриваются во время анализа помеченных данных. Если у инструкции, на которой произошло прерывание, окажутся помеченными какие-то операнды, то считается, что шаг трассы с этой инструкцией и есть искомое аварийное завершение. Примером такой ситуации может быть прерывание, возникшее во время записи значения в память инструкцией *mov* ассемблера x86. У этой инструкции адрес записи и записываемое значение являются помеченными.

После определения шагов трассы, на которых происходят получения входных данных, а также буферов с входными данными и шага с аварийным завершением начинается построение предиката пути. Буферы с входными данными выступают в качестве символьных значений. Процесс преобразования входных данных в результате работы программы описывается в виде формул над символьными значениями. Построение начинается с первого шага получения входных данных и заканчивается на шаге аварийного завершения. Условные ветвления, результат выполнения которых зависит от входных данных, порождают уравнения, отвечающие прохождению по определённой ветке в программе. Совокупность всех таких условий представляет предикат пути.

На этапе построения предиката безопасности с помощью символьных уравнений и неравенств описываются условия эксплуатации аварийных завершений, которые и формируют предикат безопасности. При составлении предиката безопасности учитывается требования по обходу защитных механизмов DEP и ASLR.

После построения предиката безопасности и объединения его с предикатом пути, начинается последний этап – решение полученной системы символьных уравнений и неравенств. При наличии решения, результирующая подстановка и будет являться эксплойтом.

В разделе 2.3 рассматривается процесс построения предиката пути, а также способы борьбы с возникающими при построении недостаточной и избыточной помеченностями. На рис. 4 представлена схема построения предиката пути. Построение предиката пути основано на применении слайсинга трассы, который используется во время выделения подтрассы. Критерием слайсинга выступают буферы с входными данными, начальным шагом является первая точка получения входных данных, а конечным шагом служит шаг с аварийным завершением программы. Каждая отобранная инструкция транслируется в машинно-независимое представление Pivot. Трансляция происходит статически. Использование промежуточного представления позволяет единообразно проводить анализ для различных процессорных архитектур, а также упрощает последующую трансляцию в сим-

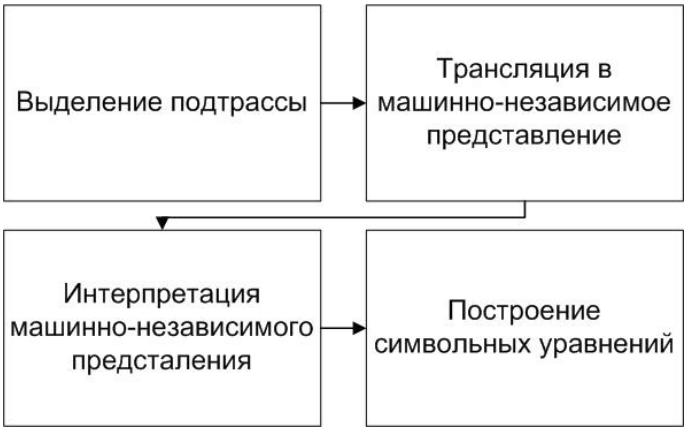

Рис. 4 — Схема построения предиката пути.

вольные формулы. Упрощение достигается благодаря тому, что Pivot имеет небольшой набор инструкций промежуточного представления. Машинные инструкции со сложной семантикой могут содержать произвольное количество ветвления в полученном Pivot-коде, в то время как выполнение машинной инструкции в трассе соответствует одному пути выполнения в Pivot-коде. Для того, чтобы в последствии упростить символьные формулы, происходит интерпретация Pivot-кода, в результате которой получается Pivot-трасса. Pivot-трасса отображает семантику выполнения машинной инструкции в трассе. Каждая инструкция Pivot-трассы транслируется в символьные формулы. Последовательная трансляция всех инструкций Pivot-трассы обеспечивает выражение выполнения машинной инструкции в виде символьных формул.

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 перемен-

ные. Обладая 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 библиотечных 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 результат 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.

В разделе 2.4 описывается построение предиката безопасности. Благодаря наличию отслеживания помеченных данных при поиске точки аварийного завершения, 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.

Условия с номерами 2 и 3 соответствуют классу аварийных завершений: исключение при попытке выполнить текущую инструкцию.

Для противодействию ASLR используются специальные инструкции «трамплины», поиск которых происходит в нерандомизируемых модулях. Трамплин представляет собой инструкцию вида: *call/jump reg.* Таким образом, сперва передаётся управление на трамплин, а после уже на код полезной нагрузки. Для борьбы с DEP используется возвратно-ориентированное программирование *(ROP)*. Код полезной нагрузки представляется в виде набора гаджетов – последовательности инструкций, заканчивающейся инструкцией передачи управления, как правило, инструкцией возврата. Гаджеты используются из области памяти доступной для выполнения. Для обхода ASLR и DEP гаджеты должны быть из нерандомизируемой и доступной для выполнения области памяти. Последовательность гаджетов, реализующая код полезной нагрузки, называется ROP-цепочкой.

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 цепочку, 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 значение 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 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*.

Для приведённых выше типов аварийных завершений происходит простроение предикатов безопасности. Алгоритмы построения предикатов безопасности приведены в диссертации.

Третья глава посвящена описанию программного инструмента, реализующего метод оценки эксплуатируемости программных дефектов. В этой главе также приводится описание программной реализации метода автоматической генерации эксплойтов и метода предварительной фильтрации аварийных завершений. Схема архитектуры программного инструмента, реализующего метод оценки эксплуатируемости программных дефектов представлена на рис. 5. Система состоит из нескольких программ-

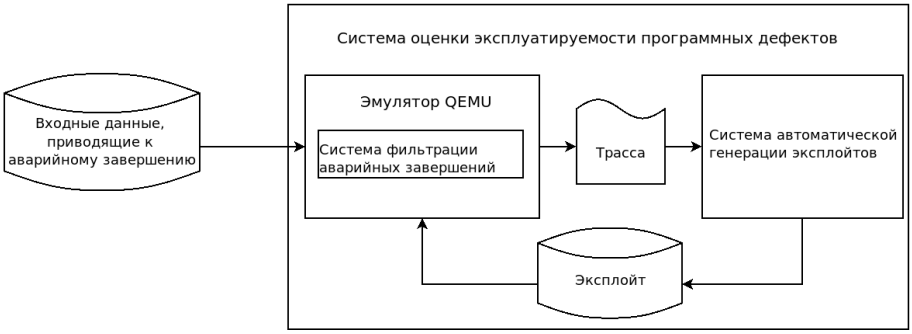

Рис. 5 — Архитектура системы оценки эксплуатируемости программных дефектов.

ных 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.

В разделе 3.1 рассматривается система предварительной фильтрации аварийных завершений, которая обеспечивает отсев неэксплуатируемых аварийных завершений и работает внутри гостевой системы в эмуляторе. Взаимодействие системы предварительной фильтарции и системы оценки эксплуатируемости программных дефектов осуществляется с помощью последовательного порта. Предварительная фильтрация реализована на базе динамического инструментатора бинарного кода *DynamoRIO*, который находится в открытом доступе и реализован на языке Си. Его использование обусловлено возможностью анализировать программы, работающие под управлением операционных систем Linux и Windows. Кроме того, инструментирование потенциально позволяет проводить анализ во время выполнения программы, который в дальнейшем может быть использован при анализе аварийного завершения. Работа инструмента вносит незначительное замедление в работу программы, что позволяет его использовать для предварительной фильтрации.

Система автоматической генерации эксплойтов, представленная разделе 3.2, 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. Проверка работоспособности эксплойта основана на запуске исследуемой программы в эмуляторе QEMU. В качестве входных данных для программы используется сгенерированный эксплойт. Код полезной нагрузки эксплойта записывает определённое значение в последовательный порт. Это записанное значение проверяется системой оценки эксплуатируемости. Если оно совпадает с эталонным значением, то эксплуатация считается успешной.

Предобработка трассы выполнения представляет собой повышение уровня представления трассы машинных команд и включает в себя восста-

новление 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.

В состав системы генерации эксплойтов входят: подсистема поиска точек получения входных данных, подсистема поиска точки аварийного завершения программы и подсистема построения предиката пути и подсистема построения предиката безопасности. Первые три из перечисленных систем для своей работы использует механизм отслеживания помеченных данных, реализованный в виде отдельной компоненты среды анализа бинарного кода. Компонента 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. При получении начального набора отслеживаемых данных используются алгоритмы работы с моделями функций, а также алгоритм восстановления буфера. 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.

Подсистема поиска точек получения входных данных использует для своей работы итератор по вызовам функций работы с файлами и источникам получения входных данных в трассе.

В подсистеме поиска точки аварийного завершения программы используются 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 точки аварийного завершения заканчивает свою работу.

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.

В подсистеме построения предиката пути используются следующие итераторы:

- 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.

Последние два итератора из вышеприведённого списка не являются обязательными, построение предиката пути может проходить в их отсутствии.

Подсистема построения предиката безопасности осуществляет построение символьных уравнений и неравенств, формирующих предикат безопасности. Также в этой системе происходит поиск трамплинов и гаджетов-трамплинов для обхода защитных механизмов. Созданный предикат безопасности объединяется с предикатом пути, после чего решается полученная система.

В разделе 3.3 приведён перечень технических ограничений текущей реализации системы. Обсуждаются способы преодоления некоторых ограничений.

В **четвертой главе** описываются результаты применения разработанных методов и реализованных инструментов.

В разделе 4.1 представлена оценка эксплуатируемости программных дефектов, полученных в результате 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 завершений представлен в табл. 1. 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. 2 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

Таблица 1 — Результаты предварительной фильтрации 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 2 — Результаты генерации эксплойтов.

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

цепочка, позволяющая обойти одновременно работающие защиты DEP и ASLR.

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.

В разделе 4.2 рассматривается оценка эксплуатируемости программных дефектов, полученных из доступных источников. Проводилась генерация эксплойтов для программ (*AudioCoder, CoolPlayer, VuPlayer, Pcman*), работающих под управлением 32-ух разрядной ОС Windows XP SP3. Эксплойты для *AudioCoder и VuPlayer* способны преодолеть защиту DEP.

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.

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

Для всех выше перечисленных примеров, аварийное завершение происходило в результате исключения, при выполнении инструкции возврата.

Аварийное завершение программы *nullhttpd* 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.

В разделе 4.3 представлена оценка эксплуатируемости дефектов в программах из тестового набора для *DARPA Cyber Grand Challenge*. Набор содержит 241 исполняемый файл. 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.

В разделе 4.4 приводится оценка эксплуатируемости модельных примеров, 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.

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.

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

В диссертации исследованы и разработаны метод оценки эксплуатируемости программных дефектов, метод предварительной фильтрации аварийных завершений и метод автоматической генерации эксплойтов по информации об аварийном завершении программы. Методы учитывают работу современных защитных механизмов, и их применение не зависит от архитектуры процессора и от операционной системы. На основе разработанных методов реализована система оценки эксплуатируемости программных дефектов. Применение разработанной системы позволило оценить эксплуатируемость набора дефектов, найденного с использованием фаззинга
.

В качестве дальнейшего направления исследований можно выделить расширение метода автоматической генерации эксплойтов на другие типы и способы эксплуатации уязвимостей. Также одним из перспективных направлений работ является улучшение фильтрации аварийных завершений.

Следует проводить анализ не только на исходном наборе входных данных, приводящих к аварийному завершению, но и на наборах, полученных из исходного набора путём удаления части входных данных. Это позволит обнаруживать новые классы аварийных завершений, а также уменьшить размер входных данных для эксплуатируемых дефектов.

Основные результаты диссертационной работы

1. Разработан 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. Разработан метод оценки эксплуатируемости программных дефектов, использующий автоматическую генерацию эксплойтов и предварительную фильтрацию аварийных завершений.
3. На основе предложенных автором методов, разработана и реализована система оценки эксплуатируемости программных дефектов.

Публикации по теме диссертации

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. — Т. 26, № 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. — Vol. 41, no. 6. — Pp. 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.