DOI: 10.15514/ISPRAS-2021-33(4)-15

Об особенностях фаззинг-тестирования сетевых интерфейсов в условиях отсутствия исходных текстов

[1] *И.В. Шарков, ORCID: 0000-0002-9767-142X <sharkov@ispras.ru>*
[1,2] *В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*
[3] *П.В. Хенкин <Pkhenkin@yandex.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*
[3] *ПАО Сбербанк, 117312, Россия, Москва, ул. Вавилова, д. 19*

Аннотация. Цифровизация общества приводит к созданию большого количества распределенных автоматизированных информационных систем в различных областях современной жизни. Необходимость отвечать требованиям безопасности и надежности подталкивает к созданию инструментов их автоматизированного тестирования. Фаззинг-тестирование в рамках цикла безопасной разработки является необходимым инструментом решения указанной задачи, но не менее востребованы средства фаззинга бинарного кода, обеспечивающие поиск критических дефектов в уже функционирующих системах. Особенно остро этот вопрос стоит при исследовании безопасности проприетарных систем, функционирующих с использованием закрытых протоколов. В ходе проводимых исследований было выявлено, что для тестирования сетевых приложений в условиях отсутствия исходных текстов применение универсальных фаззеров затруднительно. Эти обстоятельства подталкивают к созданию простого в эксплуатации инструмента фаззинг-тестирования сетевых программ. В работе рассматриваются особенности фаззинг-тестирования такого рода программ и предлагаются возможные пути решения выявленных проблем.

Ключевые слова: фаззинг; тестирование; сетевые программы; спецификация протокола; DynamoRIO; протокольный автомат

Для цитирования: Шарков И.В., Падарян В.А., Хенкин П.В. Об особенностях фаззинг-тестирования сетевых интерфейсов в условиях отсутствия исходных текстов. Труды ИСП РАН, том 33, вып. 4, 2021 г., стр. 211-226. DOI: 10.15514/ISPRAS–2021–33(4)–15

Features of fuzzing network interfaces without source codes

[1] *I.V. Sharkov, ORCID: 0000-0002-9767-142X <sharkov@ispras.ru>*
[1,2] *V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*
[3] *P.V. Khenkin <Pkhenkin@yandex.ru >*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *PJSC Sberbank, 19, Vavilova Street, Moscow, 117312, Russia*

Abstract. The digital transformation of society is leading to the creation of a large number of distributed automated information systems in various areas of modern life. The need to meet security and reliability requirements prompts the creation of tools for their automated testing. Fuzzing within the security development lifecycle (SDL) is a strictly required tool for solving this problem. Tools for fuzzing binary-only applications are in demand too. These kind of fuzzing tools provide the search for critical defects in already functioning systems. It is especially acute when researching the security of proprietary systems operating using closed protocols. In the course of the research, it was found out that for fuzzing network applications in the absence of source codes, the use of universal fuzzers is complicated by many factors. These circumstances are pushing for the creation of an easy-to-use tool for network applications fuzzing. The paper discusses the features of fuzzing of this kind of programs and suggests possible solutions to the identified tasks.

Keywords: fuzzing; testing; network applications; protocol specification; DynamoRIO; protocol state machine

For citation: Sharkov. I.V., Padaryan V.A., Khenkin P.V. Features of network interfaces fuzzing without source codes. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 4, 2021. pp. 211-226 (in Russian). DOI: 10.15514/ISPRAS–2021–33(4)–15

1. Введение

В условиях стремительной цифровизации общества крайне востребованы инструментальные средства анализа кода программного обеспечения (ПО) на наличие дефектов. Растет количество распределенных автоматизированных информационных систем, обеспечивающих функционирование объектов критической инфраструктуры, безопасность, медицинскую деятельность, видеонаблюдение, электронную коммерцию, хранение и обработку персональных данных. Целенаправленная эксплуатация программных дефектов способна вывести из строя или захватить управление такой системой, что влечет за собой очень серьезные последствия.

В состав современных инструментальных средств входят средства динамического анализа (тестирования), где для обеспечения безопасности наиболее популярным видом тестирования является фаззинг.

Фаззинг-тестирование — это способ автоматизированного тестирования программ, способный находить дефекты в их исполняемом коде вне зависимости от наличия исходных текстов. Инструменты фаззинг-тестирования входят в стек технологий цикла безопасной разработки, регламентированного нормативными документами ведомств-регуляторов, проведение процедур фаззинг-тестирования предусмотрено ГОСТ Р 56939-2016 как необходимой меры разработки безопасного ПО.

Существуют различные хорошо зарекомендовавшие себя в практической работе комплексные инструменты и подходы к решению данной задачи. Например, к подобным средствам относится инструмент динамического анализа Crusher [1] (ИСП РАН). Нормативные требования, необходимые с точки зрения разработки безопасного ПО, приводят к формированию типовых методик фаззинг-тестирования и стандартизации технологических цепочек. Данные условия вынуждают создавать универсальные и сложные

в эксплуатации инструменты фаззинг-тестирования, функционирование которых основано на типовых приемах, что затрудняет реализацию оригинальных и нестандартных методик фаззинг-тестирования. Стоит отметить, что многие серьезные уязвимости ПО были найдены не с помощью сложных многофункциональных систем, а с помощью простых программ, созданных для решения конкретной задачи с использованием оригинальных подходов [2-5].

Фаззинг, осуществляемый в рамах цикла безопасной разработки, предназначен для решения задач выявления дефектов до начала эксплуатации программного средства, т. е. нацелен на предупреждение выпуска ПО с дефектами и уязвимостями. Однако, учитывая факты выявления новых уязвимостей в «старых» компонентах ПО, в заимствованном коде, очевидно, что полностью решить задачу этим путем не удается, примером может служить уязвимость OpenSSL CVE-2021-3449 [6]. Данное обстоятельство является двигателем развития средств автоматизированного тестирования бинарного кода в условиях наличия минимума информации и отсутствия исходных текстов.

Фаззеры можно рассматривать не только как технологический элемент жизненного цикла безопасного ПО. Одновременно с этим они являются незаменимыми инструментами исследования и оценки надежности и стабильности бинарного кода различных классов ПО, их целесообразно применять, как один из инструментов в тестировании на проникновение (penetration test). В таких условиях средства фаззинга не могут использоваться для «тяжеловесной» верификации программных средств и должны обеспечивать «недорогой», легковесный метод поиска ошибок и выявления уязвимостей. При этом слово «недорогой» следует относить не столько к использованию вычислительных ресурсов в процессе функционирования инструмента, сколько к требованиям по полноте и доступности исходных данных на подготовительном этапе: исходных текстов, документации ПО, конфигураций CI/CD, функциональных тестов и тестовой обвязки, и т.д.

Основная задача данного исследования заключается в систематизации проблем, возникающих при фаззинге в условиях отсутствия исходных текстов, и определении возможности создания гибкого, легковесного и «недорогого» в использовании программного средства, или фреймворка для автоматизированного тестирования сетевых серверных и клиентских программ, доступ к исходным текстам которых невозможен.

2. Постановка задачи и способы решения

В качестве исходных данных рассмотрим ситуацию, когда в нашем распоряжении имеются настроенные и корректно функционирующие на контролируемых компьютерах серверная и клиентская программы. Среда их функционирования ОС Linux x86-64. Исходных текстов нет. Протокол сетевого взаимодействия неизвестен (не документирован). В качестве целевого объекта выберем серверную программу. Основная задача заключается в выявлении ошибок, приводящих к нарушению ее функционирования. Проведя декомпозицию задачи получается, что для осуществления фаззинг-тестирования целевых приложений необходимо решить следующие задачи:

- проанализировать алгоритмы, выявить доступные для воздействия блоки;
- изучить спецификацию сетевого протокола, включая служебную составляющую;
- определить поверхность атаки;
- сформировать систему оценки результатов поиска ошибок;
- реализовать систему контроля состояний целевой программы с интерфейсом прямой и обратной связи;
- подготовить начальную последовательность и создать способ автоматической модификации (мутации) данных, предназначенных для отправки исследуемой программе.

2.1. Анализ алгоритмов сетевых программ

Создание универсального фаззера упирается в использование большого количества сильно различающихся алгоритмов в разных программах. В большинстве случаев ход автоматизированного тестирования подразумевает, что исследуемая программа корректно завершается самостоятельно по завершению теста или аварийно в процессе его выполнения, в противном случае принимается решения о зависании, выполняется ее принудительное завершение и запускается очередной цикл тестирования.

Сетевые программы, как правило, самостоятельно не завершаются поскольку их алгоритмы основаны на бесконечных циклах ожидания подключений, обработки входных данных и отправки исходящих сообщений.

В этом состоит одна из трудностей их фаззинг-тестирования в условиях отсутствия исходных текстов. Модифицировать бинарный код с целью ограничения количества итераций хоть и возможно, но крайне затратно; требуется другой подход.

Если абстрагироваться от деталей конкретных реализаций, абстрактный алгоритм функционирования сетевого сервера (рис. 1) заключается в следующих шагах:

- создание сокета;
- инициализация сетевого подключения в режиме прослушивания;
- ожидание запроса клиента на подключение;
- порождения дочернего процесса/потока для осуществления сетевого взаимодействия;
- переход в состояние ожидания очередного запроса на подключение.

Обработка входных данных осуществляется в цикле получения-отправки данных в дочернем процессе/потоке.

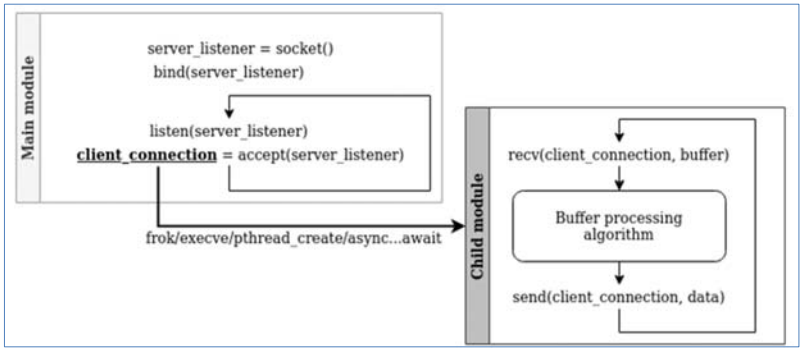


Рис. 1. Схема работы сетевого сервера
Fig. 1. Network server work scheme

Из описания алгоритма функционирования серверного приложения видно, что внешнее воздействие на его функционирование возможно оказать через сетевой интерфейс только в дочернем процессе/потоке после получения порции данных от подключенного клиента. Таким образом, изучение алгоритма серверной программы можно сузить до блока обработки входных данных (на схеме «Buffer processing algorithm», см. рис. 1), точкой входа следует рассматривать инструкцию следующую за функцией *recv(…)*, а точкой выхода – перед вызовом функции *send(…)*.

Клиентские сетевые программы функционируют по схожему алгоритму, за исключением того, что именно клиент инициирует создание канала вызовом функции *connect(…)*, после чего начинается цикл обмена информацией. Стоит обратить внимание на то, что блок

обработки входящих сообщений здесь может быть значительно больше, поскольку может включать часть кода, отвечающего за пользовательский интерфейс, журналирование и т.п.

2.2. Анализ спецификации сетевого протокола

Задача восстановления сетевого протокола разделяется на две дополняющие друг друга подзадачи: восстановление форматов сетевых сообщений и восстановление протокольного автомата. Для первой задачи были предложены различные подходы к ее автоматизированному решению, известные еще с нулевых годов, для второй задачи на данный момент эффективные автоматизированные решения авторам данной статьи неизвестны. Более того, даже в условиях наличия исходных текстов тестирование реализации протокола зачастую осуществляется посредством вручную написанной тестовой системы, которая в полной мере реализует ответную часть сетевого взаимодействия.

Методы восстановления сетевых протоколов можно разделить на четыре группы:

- запись и анализ сетевых трасс;
- дизассемблирование целевой программы и анализ полученного статического представления;
- отложенный анализ потока данных по записанной трассе выполнения тестируемой программы;
- динамический анализ потока данных непосредственно во время выполнения.

Запись сетевых трасс позволяет получить порядок обмена сетевыми сообщениями, просмотреть их содержимое, определить размер и назначение отдельных полей, но не дает возможность увидеть полную картину структур пакетов.

Дизассемблирование формирует статический код алгоритмов и потенциально позволяет решать задачу восстановления спецификации протокола, однако в случае анализа сложных громоздких программ этот процесс достаточно трудоемкий и требует значительного времени. Единственным средством автоматизации выступает декомпилятор, который восстанавливает переменные, в том числе – агрегатных типов. Но такой важный аспект, как семантику полей декомпиляция не захватывает, решать эту задачу приходится вручную, как и восстановление протокольного автомата.

Основная идея анализа потока данных заключается в контроле обращений к содержимому блока памяти и его распространению в процессе функционирования программы. В задаче восстановления спецификаций сетевых протоколов данный подход позволяет выделить основные элементы пакетов, их размеры, смещения относительно начала исходного блока. В результате формируется обобщенная структура сетевых пакетов, необходимая для рациональной модификации/мутирования сообщений в ходе осуществления фаззинг-тестирования. Опять же задача восстановления протокольного автомата в данном подходе не решается.

Запись и анализ трассы выполнения программы позволяет анализировать потоки данных, отслеживать их распространение, не только по самой программе, но и между процессами. Однако эта работа достаточно ресурсоемкая, особенно в случае записи трасс во время осуществления длительных сессий взаимодействия клиента и сервера, необходимых для воссоздания наиболее полных спецификаций.

Последний подход предполагает выполнение динамического анализа осуществляется «на лету», в процессе работы исследуемой программы. Данный подход может быть реализован путем динамической инструментации с использованием, например, систем Qemu, DynamoRIO или Pin и позволяет не только отслеживать поток управления, но контролировать его. Кроме того, динамический анализ позволяет осуществлять мониторинг только интересующей части алгоритма, что существенно уменьшает объем работы.

Сетевые протоколы разделяются на протоколы без состояния (stateless-протоколы) и требующие сохранения состояния (stateful-протоколы). При осуществлении взаимодействия между клиентом и сервером с помощью stateless-протокола каждое последующее сообщение не зависит от предыдущего, примером может служить HTTP, stateful-протоколы определяют строгий порядок отправки и получения сообщений, требуемый для перехода к следующему состоянию, нарушение которого приводит к откату к предыдущему состоянию или закрытию сетевого канала, например, протоколы FTP, TLS.

В условиях неизвестного сетевого протокола возникает отдельная задача классификации протокола как stateless или stateful. Решать такую задачу автоматизировано возможно при наличии представительных образцов трафика, по которым записывается трасса выполнения исследуемой программы. После выявления в ней мест, в которых принимаются входящие пакеты необходимо определить наличие связи в потоке данных между обработкой разных пакетов. В силу сложности и самодостаточности данной задачи в настоящей статье она не рассматривается.

Представляется целесообразным оценивать полноту достигнутого покрытия не только по покрытию кода, как того требуют действующие нормативные документы. При тестировании сетевого ПО не менее важно обеспечить максимально полное покрытие состояний протокольного автомата. Ниже по тексту будут использоваться понятия фаззинга «в ширину» и «в глубину», во избежание возникновения коллизий определим, что эти понятия в данном контексте рассматриваются применительно к спецификации сетевого протокола, а не к структуре бинарного кода исполняемой программы. Удлинение цепочки задействованных в процессе тестирования базовых блоков не указывает на увеличение глубины фаззинга, а переход на следующий уровень протокола как раз и является индикатором погружения и перехода протокола в новое состояние.

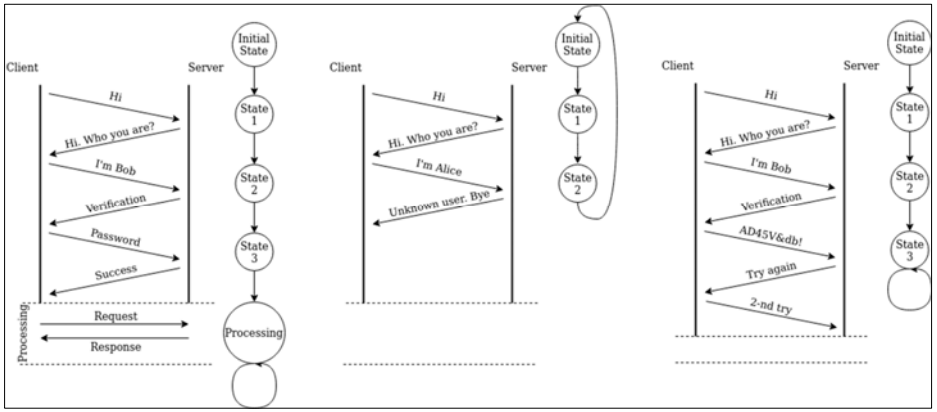
Фаззинг-тестирование сетевых программ, реализующих stateless-протоколы, представляет собой исследование «в ширину», поскольку порядок отправки пакетов неважен требуется максимально расширить количество возможных типов пакетов. Интересует процесс обработки входных данных, задача сводится к модификации входных данных таким образом, чтобы максимально увеличить количество задействованных в их обработке функций. При этом необходимо осуществлять контроль выполнения целевой программы путем отслеживания переходов между базовыми блоками.

Исследования ПО использующего stateful-протоколы сложнее, здесь важен и порядок и содержимое передаваемых сообщений, конечной целью можно назвать фаззинг «в глубину», т.е. осуществление результативной отправки наибольшего числа различных обрабатываемых (не отбрасываемых) целевой программой сообщений. Однако при достижении очередного уровня «погружения» в соответствии с имеющейся спецификацией протокола следует осуществлять фаззинг «в ширину», контролируя при этом состояние целевой программы и протокола для принятия решения об отправке очередного пакета. Для решения этой задачи может использоваться математический аппарат конечных автоматов или цепей Маркова [9].

Протокольный автомат определяет порядок и задает функцию перехода из состояния в состояние, как правило, является подсистемой инструмента фаззинг-тестирования, он задействован в процессе генерации набора входных данных и применяется для исключения случайных пакетов, не удовлетворяющих спецификации, и сокращения количества нерезультативных тестов. Наиболее удобный способ графического представления протокольного автомата – ориентированный граф, где вершины задают состояния, а ребра отражают возможные переходы.

Протокольный автомат строится итеративно. Рассмотрим пример абстрактной системы с авторизацией и аутентификацией. Осуществим подключение клиента к серверу. Проведем три таких эксперимента. В ходе первого выполним подключение с использованием валидной пары логин-пароль и реализуем после этого работу с системой (рис. 2а). Во втором

эксперименте используем неверное имя пользователя (рис. 2b. В третьем – неправильный пароль (рис. 2c). В результате будут получены три графа переходов состояний, каждый из которых описывает часть протокольного автомата данной системы.



a) сессия 1 *b) сессия 2* *c) сессия 3*

Рис. 2. Пример этапов построения протокольного автомата
Fig. 2. Stages of building a protocol state machine sample

Полученные графы переходов легко обобщаются в единый протокольный автомат (рис. 3), отражающий реализованную в рамках тестовых сессий часть протокола. Вместе с тем, важно отметить, что начинать фаззинг-тестирование в общем случае можно даже имея только часть автомата, полученную в рамках одной сессии, а его дополнение и модернизация может осуществляться в ходе тестирования с использованием вновь полученных данных.

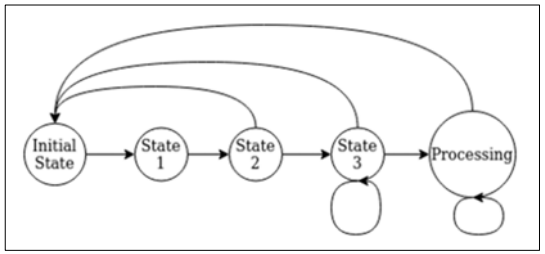


Рис. 3. Пример обобщенного протокольного автомата
Fig. 3. Composite protocol state machine example

Успешная практика применения протокольных автоматов в процессе фаззинга сетевых приложений наглядно отражена в инструменте AFLNet [12], основанном на широко известном фаззере AFL, позволяющем осуществлять фаззинг-тестирование программ без исходных текстов, для инструментации в этом случае используется эмулятор QEMU. Архитектура инструмента такова, что автомат строится вручную на основе анализа последовательности перехваченных сетевых пакетов исследуемого приложения и реализуется функцией на языке программирования C. Однако это вносит определенные неудобства в силу того, что для каждого нового протокола приходится перекомпилировать и отлаживать инструмент. Важно отметить, что в ходе написания функции, реализующей протокольный автомат, требуется спецификация протокола и понимание структуры сообщений. В противном случае результативность и успешность фаззинг-тестирования не гарантируется.

В ходе итерации фаззинга с использованием AFLNet выполняется последовательная отправка цепочки сообщений и анализируются ответы тестируемой программы, по завершению отправки осуществляется ее принудительное завершение командой *kill(…)*, что в некоторых случаях приводит к зависанию и необходимости прерывания фаззинга.

2.3. Определение поверхности атаки

При построении систем, функционирующих с использованием защищенных каналов связи, используется технология инкапсуляции протоколов, схематично это показано на рис. 4.

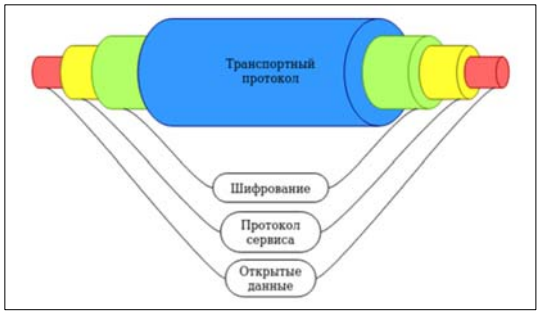


Рис. 4. Схематичный пример инкапсуляции протоколов
Fig. 4. Schematic example of protocol encapsulation

Открытые данные – это данные доступные в явном виде, например, сообщения, отображаемые с помощью элементов пользовательского интерфейса, или файлы, хранящиеся на носителе информации. Открытые данные кодируются и трансформируются в соответствии с формальным представлением протокола программы, формируется сообщение для отправки. Далее оно передается системе шифрования, формируется новое сообщение согласно спецификации используемого средства защиты. После чего транспортный протокол осуществляет его отправку адресату. На принимающей стороне для извлечения открытых данных осуществляется обратная процедура «снятия» слоев.

Программные дефекты, потенциально, могут присутствовать в программных модулях, обеспечивающих работу любого слоя, а значит проведение фаззинг-тестирования следует проводить для всех используемых программных модулей и задействованных протоколов. Причем чем больше «глубина погружения» между слоями тем больше требуется дополнительной информации.

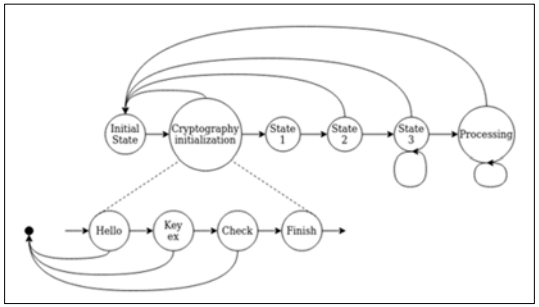


Рис. 5. Протокольный автомат программной системы с подсистемой защиты информации
Fig. 5. System with crypto protection subsystem protocol state machine example

Независимо от типа протокола, после инкапсуляции для обеспечения безопасности передаваемых данных (например, с использованием TLS), его следует рассматривать, как stateful-протокол.

Усложним рассмотренную ранее систему, введя в ее состав подсистему защиты передаваемой информации. Протокольный автомат новой системы получит дополнительное состояние – прохождения процедуры инициализации защищенного соединения (рис. 5).

Система построения защищенного канала связи имеет собственный протокол, значит, как показано на рис. 5, при необходимости возможно построить автомат, описывающий ее состояния. Однако осуществлять фаззинг-тестирование подобной системы лучше разделив ее на две: самой системы без защиты информации и подсистему защиты информации, - это позволит существенно упростить процесс.

2.4. Оценка результатов фаззинга и контроль состояния целевой программы

Современные фаззеры для оценки результатов тестирования используют измерение количества выполненных функций, базовых блоков, строк исполненного исходного текста, реберное покрытие графа переходов между базовыми блоками и т.д.

Прирост покрытия подсчитывается по результатам набора тестов, между которыми осуществляется перезапуск целевой программы и, как следствие ее переход в начальное состояние. В результате полученная оценка показывает, что та или иная часть кода достижима в ходе тестирования, однако, не гарантирует отсутствие в нем дефектов, которые могут проявиться в ходе длительной эксплуатации. Кроме того, в случае, когда исходные тексты недоступны и спецификация неизвестна, достигнуть 100% покрытие исполняемого кода невозможно.

В сетевых приложениях количество кода, отвечающего за непосредственное сетевое взаимодействие и обработку входных данных, может занимать лишь малую часть от его общего объема. В этом случае в процессе тестирования подсчитанное покрытие будет несущественным, а подход к оценке результатов на основе данной метрики мало информативен.

Два выше описанных фактора приводят к необходимости использования новой метрики для оценки результатов фаззинг-тестирования сетевых программ. Предполагается в качестве альтернативы использовать «покрытие» спецификации протокола. Подсчет реберного покрытия, вместе с тем, будет использоваться в качестве инструмента контроля процесса фаззинга, выявления условных переходов и подтверждения продвижения между базовыми блоками. Решить данную задачу возможно с применением средства динамической инструментации.

2.5. Подготовка начальной последовательности и модификация входных данных

В качестве начальной последовательности для инициализации процесса фаззинга целесообразно выбирать набор легитимных сетевых пакетов, отправленных целевой программе в процессе ее изучения и восстановления спецификации протокола. Кроме того, при проведении фаззинг-тестирования сетевых программ целесообразно не уменьшать корпус начальных данных, а максимально нагружать протокольный автомат.

Наличие спецификации позволяет формировать правильные с точки зрения назначения полей сообщения, однако, в случае использования в их структуре контрольных сумм возникают дополнительные трудности. Для правильного вычисления контрольной суммы сообщения необходимо знать алгоритм и понимать какие данные из сообщения используются в качестве параметров для функции ее вычисления. В условиях рассматриваемой задачи исходные

тексты целевой программы недоступны и алгоритм вычисления контрольной суммы неизвестен, указанную информацию возможно получить в ходе анализа алгоритмов целевой программы после дизассемблирования или в процессе анализа трассы выполнения программы, однако, как было отмечено ранее, эти подходы трудозатратные и ресурсоемкие.

Существуют подходы, основанные на генетических алгоритмах, позволяющие в ходе фаззинга генерировать сообщения, которые проходят проверку контрольной суммы и могут вызвать аварийное завершение тестируемой программы. Однако одним из основных недостатков этого пути можно назвать высокую вычислительную сложность и, соответственно, значительные временные затраты [7].

Ниже предлагается компромиссный способ, позволяющий решить задачу фаззинг-тестирования с некоторыми ограничениями, не меняющий ранее принятую парадигму о «легковесности» подхода.

Проверки контрольных сумм в бинарном коде реализуются инструкциями JZ или JNZ [8] обойти их можно инвертировав условные переходы, это позволит исключить необходимость подсчета контрольных сумм в процессе генерации входных данных и использовать случайное значение, поскольку пакеты с неверными контрольными суммами будут проходить проверку, а случайная генерация правильных контрольных сумм статистически маловероятна.

Для определения точек проверки контрольных сумм, подлежащих инверсии, можно обратиться к алгоритмам обработки входящих сетевых пакетов. Они подразумевают, что в первую очередь, во избежание выполнения ненужных, а возможно, опасных действий необходимо проверять целостность полученных данных, поэтому в общем случае можно предположить, что одна из первых инструкций условных переходов (JZ или JNZ) после получения сообщения и вычисления его контрольной суммы будет искомой, такое же предположение делается в исследовании [9].

При использовании данного подхода необходимо учитывать, что подтвердить наличие выявленных программных дефектов с использованием полученных последовательностей сообщений при работе с не инструментированной целевой программой напрямую не получится поскольку необходимо вычислить верные контрольные суммы. Для решения данной проблемы в ряде случаев возможно на основе имеющейся спецификации протокола выделять полезную нагрузку из сгенерированных в процессе фаззинга сетевых пакетов и использовать оригинальное клиентское приложение для передачи серверу, при этом валидные контрольные суммы будут вычислены автоматически.

3. Проблемы и перспективы развития инструментов фаззинг-тестирования сетевых программ

Осуществление фаззинг-тестирования сетевых программ связано с рядом проблем, часть из них обусловлена упрощениями алгоритмов и допущениями в процессе анализа программ:

- неверная инструментация инструкций может сломать алгоритм функционирования программы, в результате чего выявленные дефекты не будут проявляться в процессе работы оригинальной программы;
- фаззинг протоколов без обратной связи осложнен отсутствием информации о результатах обработки сообщений, для решения требуется синхронизировать протокольный автомат на стороне инструмента фаззинга и средство контроля целевого приложения в рамках инструментации;
- замедление работы, связанное с инструментацией, может провоцировать обрывы соединений;
- масштабирование фаззинг-тестирования. Основная идея масштабирования заключается в уменьшении времени, затраченного процесс тестирования. Достигаться это может за

счет параллельного выполнения нескольких экземпляров целевой программы или путем выполнения большого количества подключений к одной. Единого рецепта решения данной задачи нет. В некоторых случаях приходится довольствоваться осуществлением тестирования в режиме «one-to-one»;

- выбор стека технологий для разработки инструментов фаззинга. Разработка компонентов фаззера должна осуществляться на языке программирования получившем широкое распространение, обеспечивающем низкую стоимость, низкий порог входа и гибкость. С этой точки зрения в силу его широкого распространения подходит язык программирования C, однако, любое исправление требует перекомпиляции и отладки, что доставит немало неудобств. В случае фаззинга сетевого ПО, а особенно остро этот вопрос встает при исследовании программ, взаимодействующих по протоколу с закрытой спецификацией, необходимо иметь простой способ внесения изменений в исходные тексты и отладки модулей, осуществляющих сетевое взаимодействие с целевым объектом, модификацию данных, а также реализующих протокольный автомат statefull-протоколов. Это обстоятельство подталкивает к использования скриптовых языков, например, Python, несмотря на более низкую производительность программы.

Направления развития инструментов фаззинг-тестирования, в основном, направлены на автоматизацию подготовительного этапа и рутинных операций:

- совершенствование алгоритмов восстановления форматов;
- автоматизированный способ создания оптимальной начальной последовательности входных данных;
- оптимизация средств инструментации;
- «умный» in-memory фаззинг;
- автоматическое построение и расширение протокольного автомата;
- применение методов машинного обучения [10];
- создание коробочных решений и фреймворков.

4. Практические результаты предложенного подхода

В настоящее время исследования по данной тематике находятся в начальной стадии, однако, уже получены некоторые практические результаты. В процессе апробации предложенных подходов к решению задачи легковесного фаззинг-тестирования разработан макет клиента системы DynamoRIO [11] для инструментации сетевых серверных программ.

Реализованные алгоритмы позволяют отслеживать факты подключения клиентов к серверу, после выполнения которых наблюдение за потоком управления и контроль потока данных переносится в порожденные потоки или дочерние процессы, осуществляющие непосредственное взаимодействие между клиентом и сервером. Далее начинается анализ входящего потока согласно предложенной в п. 2.1 схеме, это позволяет в автоматическом режиме контролировать процесс обработки сообщений и выделять их составные блоки и формировать структуру согласно спецификации.

Для проведения экспериментов в качестве целевого объекта был выбран сервер баз данных PostgreSQL. Данная программа представляет интерес, во-первых, потому, что использует гибридный протокол, включающий как бинарную, так и текстовую части. С одной стороны, это существенно упрощает процесс оценки успешности экспериментов, поскольку часть структурных блоков сообщений представляют собой обычный текст, с другой – позволяет сделать предположение о потенциальной возможности анализа бинарных протоколов.

Во-вторых, сервер баз данных PostgreSQL, порождает процессы-потомки для каждого нового подключения и выполнения служебных операций. Это позволяет протестировать возможность переноса контроля потоков управления и данных в дочерние процессы.

Ход эксперимента:

- запуск сервера под инструментацией;
- подключение к серверу с использованием клиентской программы;
- выполнение запроса к базе данных.

Результаты анализа потока данных	Данные (HEX)	Описание
libc_recv(8, 0x7f5418292760, 81)	0000005100030000757365720074766973740064617461746162617365007006f7374677265730061706c69636174696f6e5f6e616d6500707371 6c00636c69656e745f656e636f64696e67005554463800000	Размер сообщения 81 байт
from 0x7f5418292760 + 0 to 0x7fff5d6608bc size 4	00000051	000000Q
from 0x7f5418292760 + 4 to 0x7f541830c5a0 size 77	0003000075736572007476697374006461746261736500706f737467726573007061706c69636174696f6e5f6e616d65007073716c00636c69656e745f656e636f64696e6700555446380000	00030000user00tvist00database00postgres00application_name00psql00client_encoding00UTF80000
from 0x7f541830c5a0 + 9 to 0x7f5418336568 size 6	747669737400	tvist00
from 0x7f541830c5a0 + 24 to 0x7f5418336580 size 9	706f73746772657300	postgres00
from 0x7f541830c5a0 + 33 to 0x7f5418307700 size 17	6170706c69636174696f6e5f6e616d6500	application_name00
from 0x7f541830c5a0 + 50 to 0x7f54183365f0 size 5	7073716c00	psql00
from 0x7f541830c5a0 + 55 to 0x7f5418336628 size 16	636c69656e745f656e636f64696e6700	client_encoding00
from 0x7f541830c5a0 + 71 to 0x7f5418336668 size 5	5554463800	UTF800
from 0x7f54183365f0 + 0 to 0x7f5418327590 size 5	7073716c00	psql00
from 0x7f5418327590 + 0 to 0x7f5198cd6980 size 4	7073716c	psql
from 0x7f5418336668 + 0 to 0x7f5418327380 size 5	5554463800	UTF800
from 0x7f5418327590 + 0 to 0x7f5198cd6980 size 4	7073716c	psql
from 0x7f5418327590 + 0 to 0x7f5418336bb0 size 5	7073716c00	psql00
from 0x7f5418336bb0 + 0 to 0x7f5418307061 size 5	7073716c00	psql00
from 0x7f5418327380 + 0 to 0x7f5418336bb0 size 5	5554463800	UTF800
from 0x7f5418336bb0 + 0 to 0x7f5418307060 size 5	5554463800	UTF800
from 0x7f5418336bb0 + 0 to 0x7f5418307062 size 3	6f6e00	on00
from 0x7f5418336bb0 + 0 to 0x7f541830705d size 3	6f6e00	on00
from 0x7f5418336bb0 + 0 to 0x7f5418307060 size 5	5554463800	UTF800
from 0x7f5418336bb0 + 0 to 0x7f541830705f size 5	31312e3300	11.300
from 0x7f5418336bb0 + 0 to 0x7f5418307066 size 6	747669737400	tvist00
from 0x7f5418336bb0 + 0 to 0x7f541830706c size 3	6f6e00	on00

Листинг 1. Вывод анализа потока данных в ходе обработки пакета авторизации клиента СУБД
Listing 1. Data flow analysis output during processing of a client authorization package

Вывод результатов анализа отдельных сетевых сообщений с минимальной детализацией (с целью уменьшения его объема), отражающей копирование блоков данных и игнорирующий чтение отдельных байтов, полученный с помощью средства инструментации в ходе эксперимента, приведен в листингах 1 и 2; в столбце «Результаты анализа потока данных» приведены траектории движения блоков данных, выделенных из полученных сообщений; столбец «Данные HEX» отражает данные в шестнадцатеричном виде; столбец «Описание» показывает комментарий или печатные символы, соответствующие шестнадцатеричным

кодам. Результаты разбора тех же пакетов с помощью анализатора Wireshark приведены на рис. 6 и 7.

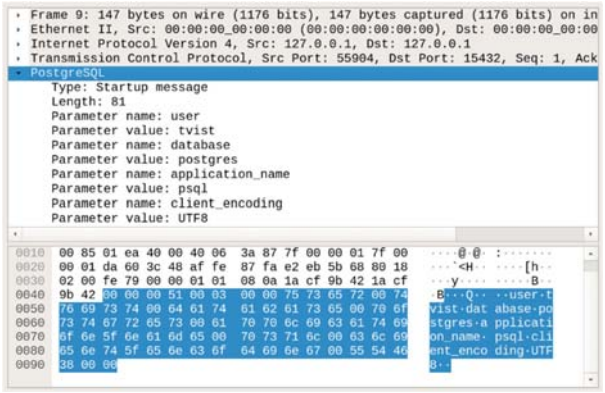


Рис. 6. Декодирование пакета авторизации клиента с помощью Wireshark
Fig. 6. Client authorization packet decoding using Wireshark

Результаты анализа потока данных	Данные (HEX)	Описание
libc_recv(8, 0x7f5418292760, 29)	510000001d73656c656374202a2066726f6d2070675f726f6c65733b00	Получено 29 байт
from 0x7f5418292760 + 1 to 0x7fff5d6605bc size 4	0000001c	0000001c
from 0x7f5418292760 + 5 to 0x7f541830b1c0 size 25	73656c656374202a2066726f6d2070675f726f6c65733b00	select * from pg_roles;00
from 0x7f541830b1c0 + 0 to 0x7f51994af380 size 24	73656c656374202a2066726f6d2070675f726f6c65733b	select * from pg_roles;
from 0x7f541830b1c0 + 0 to 0x7f541830baf0 size 24	73656c656374202a2066726f6d2070675f726f6c65733b	select * from pg_roles;

Листинг 2. Вывод анализа потока данных в ходе обработки пакета с запросом к базе данных
Listing 2. Data flow analysis output during processing of a client request to the database

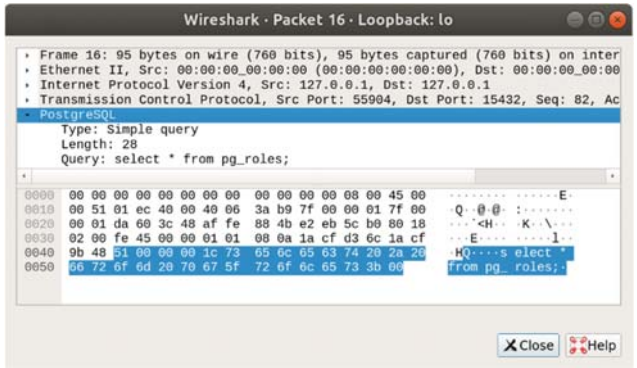


Рис. 7. Декодирование пакета с запросом клиента к базе данных с помощью Wireshark
Fig. 7. Client request to database decoding using Wireshark

Из приведенных листингов в сравнении с результатами декодирования тех же пакетов с использованием Wireshark видно, что определены все основные блоки полученных сервером

сообщений, их смещения относительно начала буфера, в который данные были записаны функцией *recv(...)*, и размеры блоков. Данное описание позволяет сформировать правила модификации данных в ходе фаззинг-тестирования.

Проведенные испытания подтвердили:

- эффективность предложенного подхода к анализу алгоритма сетевых программ на основе выделения блока, ограниченного функциями recv(…) приема и send(…) отправки данных;
- принципиальную возможность автоматического восстановления спецификации протокола в ходе динамического анализа алгоритма;
- возможность аналитическим итеративным путем восстанавливать протокольный автомат.

Ближайшие задачи, поставленные в рамках данной работы, это исследования:

- возможности и разработка методов построения автомата состояний исследуемой программы;
- способов автоматического построения и расширения протокольных автоматов;
- технологических подходов к реализации in-memory фаззинга.

Дальнейшие цели исследований по этой тематике определены в соответствии с перспективными направлениями развития инструментов фаззинг-тестирования.

5. Заключение

Активная цифровизация общества сопряжена с созданием большого количества распределенных автоматизированных систем в различных сферах жизни. Любая из данных систем может стать целью атак, реализуемых за счет уязвимостей входящего в их состав программного обеспечения, выявление которых может осуществляться с использованием инструментов фаззинга. В рамках данного исследования рассматриваются особенности тестирования сетевых приложений в условиях отсутствия исходных текстов, а также предлагается подход к разработке простой с точки зрения архитектуры и эксплуатации системы или фреймворка для решения указанной задачи.

Исследование находится на начальном этапе; в настоящее время определены основные задачи фаззинг-тестирования сетевых приложений, часть которых могут вылиться в отдельные исследования.

Во-первых, восстановление спецификаций и определение типов протоколов (stateful или stateless). В большинстве случаев в ходе решения данной задачи приходится полагаться на решение аналитика. Предложенный способ восстановления спецификации был опробован в ходе тестирования сервера баз данных PostgeSQL. Однако существует формальный способ – контроль отсутствия связей по данным между обработкой пакетов, при этом, что есть связь по данным и способы ее выявления – тема отдельных исследований.

Во-вторых, способ восстановления протокольного автомата. Современные техники подразумевают использование эмпирических подходов. Стоит все же отметить что задача восстановления автомата – тоже отдельная важная тема исследований.

И, наконец, вопрос оценки результатов фаззинга и принятия решения о необходимости продолжения или завершения. В большинстве случаев оценивается покрытие базовых блоков алгоритма программы или реберное покрытие, в рассматриваемом подходе предлагается оценивать «покрытие» спецификации протокола.

Список литературы / References

[1]. Мишечкин М.В., Акользин В.В., Курмангалеев Ш.Ф. Архитектура и функциональные возможности инструмента ИСП Фаззер. Открытая конференция ИСП РАН им. В.П. Иванникова,

2020 г. / Mishechkin M.V., Akolzin V.V., Kurmangaleev Sh.F. Architecture and functionality of the ISP Fuzzer tool. Ivannikov ISP RAS Open Conference, 2020. Slides are available at https://www.ispras.ru/technologies/docs/mishechkin-isprasopen2020.pdf (in Russian).
[2]. MsFontFuzz. Available at https://github.com/Cr4sh/MsFontFuzz, дата обращения 25.07.2021.
[3]. Уязвимости Windows, связанные с обработкой шрифтов / Windows font handling vulnerabilities. Available at blog.cr4.sh/2012/06/0day-windows.html, accessed 25.07.2021 (in Russian).
[4]. Ioctlfuzzer. Available at https://github.com/Cr4sh/ioctlfuzzer, accessed 25.07.2021.
[5]. Trinity. Available at https://github.com/kernelslacker/trinity, accessed 25.07.2021.
[6]. OpenSSL Security Adwisory. Available at https://openssl.org/news/secadw/20210325.txt, accessed 16.05.2021.
[7]. Embleton S., Sparks S. & Cunningham R. Sidewinder: An Evolutionary Guidance System for Malicious Input Crafting. Available at https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf, accessed 15.08.2021.
[8]. Liu X., Wei Q. et al. CAFA: A Checksum-Aware Fuzzing Assistant Tool for Coverage Improvement, Security and Communication Networks, 2018, Article ID 9071065, 13 p.
[9]. Wang T., Wei T. et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In Proc. of the IEEE Symposium on Security and Privacy, 2010, pp. 497-512.
[10]. Yamaguchi F., Maier A. et al. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In Proc. of the IEEE Symposium on Security and Privacy, 2015, pp. 797-812.
[11]. Bruening D., Zhao Q., and Amarasinghe S. Transparent dynamic instrumentation. In Proc. of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12), 2012, pp. 133-144.
[12]. AFLNet. Available at https://github.com/aflnet/aflnet, accessed 08.09.2021.

Информация об авторах / Information about authors

Иван Владимирович ШАРКОВ – ведущий инженер отдела компиляторных технологий. Сфера научных интересов: информационные технологии, системное программирование, фаззинг, компьютерные сети, исследования программ.

Ivan Vladimirovich SHARKOV is a leading engineer at the compiler technologies department. His research interests include information technologies, fuzzing, networks, software researching.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий ИСП РАН; доцент кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляцию и виртуализацию.

Vartan Andronikovich PADARYAN is a candidate of physical and mathematical sciences, leading researcher at the compiler technologies department of ISP RAS; associate professor of the system programming department of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation, and virtualization.

Петр Владимирович ХЕНКИН – исполнительный директор - начальник отдела, департамент кибербезопасности ПАО Сбербанк. Сфера научных интересов: информационная безопасность, криптография, анализ исходных текстов, методы аутентификации и идентификации, биометрия.

Petr Vladimirovich KHENKIN is an executive director - head of department, cybersecurity department of Sberbank. His research interests include information security, source codes analyses, authentication, authorization and identification methods, and biometry.