DOI: 10.15514/ISPRAS-2021-33(5)-9

Динамический анализ IoT-систем на основе полносистемной эмуляции в QEMU

Р.Д. Коваленко, ORCID: 0000-0002-2225-5560 <r.kovalenko@ispras.ru>
А.Н. Макаров, ORCID: 0000-0003-2237-3396 <alex1118@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Стремительное развитие Интернета-вещей (IoT) требует развития методов и средств анализа подобных устройств. Существенная часть таких устройств работает под управлением операционных систем (ОС) семейства Linux. Непосредственное применение существующих средств для анализа программного обеспечения (ПО) данного класса устройств не всегда возможно. В процессе исследования встраиваемых ОС Linux был создан инструмент ELF (Embedded Linux Fuzz), который и представлен в данной работе. В статье рассматривается анализ систем, построенных исключительно на базе ядер Linux. Среда ELF предназначена для динамического анализа устройств на основе полносистемной эмуляции в QEMU. В основу ELF были положены следующие аспекты: выполнение тестирования и анализа ПО реальных устройств в среде, максимально приближенной к их «родной» среде выполнения; интеграция с существующими средствами фаззинга; возможность проведения распределенного анализа.

Ключевые слова: фаззинг; тестирование; IoT-системы; Linux

Для цитирования: Коваленко Р.Д., Макаров А.Н. Динамический анализ IoT-систем на основе полносистемной эмуляции в QEMU. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 155-166. DOI: 10.15514/ISPRAS–2021–33(5)–9

Dynamic analysis of IoT systems based on full-system emulation in QEMU

R.D. Kovalenko, ORCID: 0000-0002-2225-5560 <r.kovalenko@ispras.ru>
A.N. Makarov, ORCID: 0000-0003-2237-3396 <alex1118@ispras.ru>

Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow,109004, Russia

Abstract. The sweeping evolution of the Internet of Things (IoT) requires the development of methods and tools for analyzing such devices. A significant part of similar devices run under operating systems (OS) of the Linux family. Direct application of existing tools for analyzing software (SW) of this class of devices is not always possible. In the process of researching embedded Linux OS, the ELF (embedded linux fuzz) tool was created, which is presented in this work. The article deals with the analysis of systems built exclusively on the basis of Linux kernels. ELF environment is designed for dynamic analysis of devices based on full-system emulation in QEMU. ELF was based on the following aspects: performing software testing and analysis of real devices in an environment as close as possible to their «native» execution environment; integration with existing fuzzing tools; the ability to conduct distributed analysis.

Keywords: fuzzing; testing; IoT-systems; Linux

For citation: Kovalenko R.D., Makarov A.N. Dynamic analysis of IoT systems based on full-system emulation in QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 155-166 (in Russian). DOI: 10.15514/ISPRAS–2021–33(5)–9

1. Введение

В процессе проведения динамического анализа ПО IoT-устройств обычно возникает ряд проблем, которые прежде всего связаны с особенностями реализации и функционирования таких устройств. Во-первых, аппаратура и архитектура Интернета-вещей разнообразны (ARM, MIPS, x86 и т.д.) и зависят от реализации конкретного IoT-устройства, из-за чего анализ ПО данного класса может быть существенно затруднен. Во-вторых, для прикладного ПО IoT-систем характерно отсутствие исходных кодов, что само по себе не является уникальной особенностью IoT-систем, однако создает некоторые дополнительные трудности при выполнении их анализа, в том числе динамического. Анализ бинарного кода внутри десктопных систем, как правило, не вызывает никаких затруднений на самом раннем его этапе (дизассемблирование, запуск и т.д.), поскольку объект анализа полностью доступен. В случае же IoT-систем, процесс анализ обычно начинается с получения доступа к бинарным кодам объектов анализа. В данной статье мы не будем останавливаться подробно на этом шаге, поскольку распаковка образов IoT-устройств является отдельной темой для исследования. Отметим только тот факт, что в простом случае можно воспользоваться средством binwalk [1] или аналогичными, а также обратиться на сайт производителя, который может предоставлять обновления для своих IoT-устройств, а в некоторых случаях даже исходные коды для отдельных частей их систем.

Запуск бинарного кода IoT-устройства вне предполагаемой производителем среды выполнения может быть либо вовсе невозможен, либо затруднен, в силу чего необходимо будет провести ряд действий по настройке среды выполнения и самого ПО. Это, в свою очередь, может привести к тому, что поведение объекта анализа будет существенно отличаться от предполагаемого в оригинальном окружении. Такое положение вещей существенно ограничивает применение динамических методов анализа, в том числе фаззинга. Использование эмулятора QEMU позволяет только частично решить указанную проблему.

Часто для организации динамического анализа бинарного кода необходимо внесение дополнительных инструментов в среду выполнения (отладчики, профилировщики и т.д.). Добавление подобного инструментального кода внутрь среды выполнения IoT-устройства не всегда возможно, а в ряде случаев сопряжено со множеством трудностей. Во-первых, объём памяти IoT-устройства может быть сильно ограничен. Во-вторых, производитель IoT-устройства может использовать файловые системы (ФС), в которых на операции записи в ФС накладываются дополнительные ограничения. В-третьих, для того, чтобы пересобрать образ IoT-устройства, может потребоваться дополнительный анализ реализации загрузчиков, кода проверки целостности и т.п [2]. При этом даже в тех случаях, когда удалось поместить, например, отладчик, внутрь IoT-устройства, его использование может быть затруднено по самым различным причинам (например, ограниченность функциональности – в ядре недоступен системный вызов prtace).

Одним из эффективных подходов для организации проведения динамического анализа бинарного кода (и, в частности, фаззинга) является применение средств инструментального анализа бинарного кода (Pin Tool [3], DynamoRio [4]). Их использование внутри IoT-устройства не всегда представляется возможным, поскольку используемое окружение внутри среды, в которой функционирует код IoT-устройства, не позволяет этого сделать. В частности, внутри большинства IoT-устройств работает ОС на базе старых ядер Linux (линейки вторых и третьих версий), что также накладывает дополнительные ограничения на инструменты анализа таких систем.

Основываясь на опыте анализа IoT-систем, можно выделить два основных подхода к построению программной среды для анализа кода ядра, сервисов и пользовательских утилит, функционирующих внутри IoT-устройства. Суть первого подхода (назовем его внешним) заключается в том, что конкретные сервисы и/или утилиты извлекаются из IoT-устройства и запускаются во внешнем окружении, близком к оригинальному. Далее к ним применяются различные технологии анализа (фаззинг, символьная интерпретация, и т.д.). Второй подход (внутренний), наоборот, заключается в том, что к функционирующим в оригинальной программной среде IoT-устройства сервисам и утилитам применяются в каком-либо виде технологии анализа.

Оба подхода обладают рядом достоинств и недостатков. Самым существенным недостатком внешнего подхода является тот факт, что зачастую IoT-устройство обладает своим специфическим кодом ядра (как само ядро, так и различные модули ядра), от особенностей реализации которого сильно зависит поведение сервисов. Это делает слишком трудоемким процесс переноса анализируемых сервисов в инструментальную программную среду (даже в том случае, когда удалось подобрать полностью или частично подходящее по параметрам стандартное ядро Linux). Также зачастую сервисы, функционирующие внутри IoT-устройства, сильно связаны друг с другом, в том числе общими данными, которые неочевидным образом инициализируются в ходе трудоемкого для анализа процесса загрузки устройства. Поэтому извлечь из IoT-устройства и заставить штатным образом функционировать какой-либо один интересующий сервис (особенно сетевой) не всегда представляется возможным.

Основным преимуществом внешнего подхода является тот факт, что если удалось построить инструментальную программную среду для корректного функционирования интересующих сервисов, то дальнейший процесс их анализа (в том числе фаззинга) упрощается. Появляется возможность запускать в этой же среде различные продвинутые средства анализа, осуществлять фаззинг с учетом покрытия кода, отслеживать аварийные завершения, выходы за границы буферов памяти и т.д. Однако даже в случае успешного запуска сервисов во внешней среде, существуют ограничения, обусловленные свойствами анализируемого ПО. Часто в IoT-системах используют старые версии ядер Linux и библиотек. Инструментальные средства для анализа ПО разрабатываются с учетом работы на современных версиях ОС. Поэтому в полученном внешнем окружении такие средства почти наверняка не заработают «из коробки», их необходимо будет дополнительно настраивать, возможно компилировать специально созданными или подобранными тулчейнами.

Основным преимуществом внутреннего подхода к анализу IoT-систем является сохранение штатной среды функционирования анализируемого программного кода. Это практически исключает появление ложных ошибок (например, вследствие того, что какие-либо данные неправильно инициализированы, либо отсутствуют зависимости и т.д.), в отличие от использования внешнего подхода к анализу. Однако при этом необходимо иметь возможность запуска своего кода внутри IoT-устройства, что не всегда возможно без проведения дополнительных исследований, что является недостатком данного подхода. Еще одной трудностью, с которой можно столкнуться при использовании внутреннего подхода к анализу IoT-систем, является наличие естественных ограничений на выполняемый код, что связано с особенностями реализации того или иного IoT-устройства – платформозависимость, зависимость от версий используемых библиотек, ядра и т.д. Подобные ограничения, как уже упоминалось выше, могут возникнуть и в случае применения внешнего подхода.

Исходя из перечисленных достоинств и недостатков описанных подходов к анализу IoT-устройств, становится ясно, что в большинстве случаев целесообразно применять комбинированный подход, который заключается в том, что если в IoT-устройстве есть некоторый бинарный код, который обладает небольшим количеством внешних зависимостей, и его можно оттуда относительно просто извлечь и запустить, то анализ такого

кода можно осуществлять вне IoT-устройства (например, различные библиотеки, используемые сервисами внутри устройства, несложные утилиты, и т.д.). Анализ же сложных модулей, таких как например сетевые сервисы, целесообразно осуществлять внутри IoT-устройства.

Также в качестве отдельного обширного и довольно перспективного направления стоит выделить применение технологии виртуализации к фаззингу IoT-систем.

Данная работа состоит из следующих разделов: (2) приведен обзор существующих методов и средств для анализа IoT-устройств, в том числе на основе полносистемной эмуляции; (3) приведена схема работы инструмента ELF, его основные компоненты; (4) приведены результаты измерений на синтетических тестах, а также ряд сравнений с существующими инструментами; (5) в заключение предлагаются дополнительные новшества и улучшения в работе, которые появятся в будущих версиях инструмента ELF.

2. Существующие методы и средства

Обобщим основные подходы к динамическому анализу IoT-систем, классифицируя их по методу запуска встраиваемого кода.

1) Аппаратная отладка с использованием интерфейса JTAG или подобных средств [5]. Данный метод может применяться для анализа только части кода IoT-устройства, но трудно реализуем для анализа кода пользовательского режима. Метод аппаратной отладки зависит от IoT-устройства. Он, как правило, выполняется вручную, его трудно автоматизировать и масштабировать.

2) Технология эмуляции выполнения кода пользовательского режима (QEMU-usermode) [6] является простым и быстрым методом динамического анализа пользовательского кода, позволяющим запускать бинарный код, предназначенный для различных аппаратных архитектур (arm, mips и т.д). Суть использования данного подхода заключается в извлечении usermode-среды (бинарный код, файлы настроек и т.п.) из IoT-устройства и применения chroot, что позволяет решить проблему отсутствия файлов динамических библиотек и прочих ссылок во время выполнения. Кроме того, возможен запуск gdb для отладки кода. Однако серьезное ограничение накладывает сам механизм работы эмуляции пользовательского режима. Как только в коде встречается системный вызов, он переадресуется хостовой системе, а та, в свою очередь, не всегда может выполнить его корректно, поскольку для обработки системного вызова может требоваться специфичная аппаратура или проприетарные модули ядра, отсутствующие в ядре хостовой системы.

3) Полносистемная эмуляция в одной из сред виртуализации, например, в системном режиме QEMU. Если удается выполнить эмуляцию IoT-устройства подобным образом, то на основе данного подхода можно получить хорошие результаты. Например, такой инструмент, как FIRMADYNE [7] основан на полносистемной эмуляции.

Ключевое слово здесь – «если». Полносистемная эмуляция IoT-устройства не может быть выполнена для произвольного устройства, из-за их большого разнообразия. Современные среды виртуализации (vmware, qemu, virtualbox и т. д.) поддерживают лишь типовой набор виртуальных устройств для каждой архитектуры. Использование полносистемной эмуляции также не означает, что эмулируется абсолютно весь код IoT-устройства. Например, код загрузчика может быть заменен, если это не принципиально для проведения исследования.

4) Дополненная (augmented) эмуляция – это технология совмещения эмуляции пользовательского режима и полносистемной эмуляции. Основным преимуществом данного подхода является повышение скорости работы динамического анализа. Например, FIRM-AFL основана на этом методе [8]. Однако такой подход сам по себе не устраняет недостатков полносистемной эмуляции, он не может решить проблему аппаратной зависимости. Кроме того, переключение и обеспечение синхронизации между пользовательским и системным режимами приводит к дополнительным издержкам.

5) Частичная эмуляция Qemu – это технология эмуляции оборудования, информация о котором передается Qemu в виде описания на скриптах lua. LuaQEMU [9] — это среда для прототипирования целевых систем. Интерпретатор LuaJIT встроен в QEMU и предоставляет доступ к внутреннему API QEMU для эмуляции оборудования. Основным недостатком применения указанного подхода к анализу реальных IoT-систем является трудоемкость прототипирования в общем случае, даже если представлена документация на микросхему устройства.

6) Анализ, основанный как на реальных IoT-устройствах, так и на технологии эмуляции. Данный подход направлен на решение проблемы точности выполнения кода IoT-системы, которая не может быть решена с помощью эмуляции, и проблем автоматического анализа, которые не могут быть решены с помощью аппаратной отладки. Аватар [10] и Аватар 2 [11] являются примерами этого подхода, пытающиеся перенаправить операции ввода-вывода на реальные устройства, если их невозможно выполнить в среде эмуляции. Однако Аватар и Аватар 2 обладают концептуальной проблемой, заключающейся в том, что многократные переключения между средой эмуляции и реальными устройствами негативно влияют на скорость работы при полномасштабном фаззинг-тестировании.

Из общедоступных известных современных реализаций, основанных на полносистемной или дополненной эмуляции, отметим следующие: инструмент FIRMADYNE [7], FIRM-AFL [8], TriforceAFL [12], IoTFuzzer [13], luaqemu [9]. Опишем немного более детально основные особенности некоторых из этих реализаций.

Инструмент Firmadyne [7] представляет собой фреймворк для автоматического тестирования заданных IoT-устройств на наличие в них известных уязвимостей. В его составе имеется таблица производителей устройств, с соответствующими URL-адресами, по которым с использованием некоторого автоматизированного инструмента происходит получение с сайтов производителей бинарных образов устройств, запуск их под управлением QEMU в режиме специальной полносистемной эмуляции и последующая автоматическая проверка на наличие известных уязвимостей с использованием Metasploit Framework[14]. В результате тестовых запусков фиксируются как удачные попытки эксплуатации уязвимостей, так и аварийные завершения внутренних процессов. При этом в базе данных данного инструмента для каждого из хотя бы раз запущенных IoT-устройств сохраняется информация для возможности выполнения последующих тестовых запусков уже без повторного выкачивания образа. Поддерживаются архитектуры ARM и MIPS. В данной схеме интерес представляет подход FIRMADYNE к полносистемной эмуляции IoT-устройства в QEMU. Образ каждого выкачанного устройства разворачивается путем запуска эмулятора со специально подготовленным заранее ядром Linux (для всех образов используется одно и то же ядро, собранное разными кросс-компиляторами под соответствующую архитектуру) и образом корневой файловой системы. В данное ядро на уровне исходных кодов встраивается драйвер, который выставляет заглушки на узлы дерева устройств, псевдо-файловой системы /proc и т.д., тем самым заставляя IoT-устройство видеть необходимые ему для работы элементы системы. В качестве содержимого корневой файловой системы выступает извлеченная в автоматическом режиме из образа IoT-устройства файловая система, в которую добавлена поддержка дополнительной консоли для трансляции управляющих команд, а также библиотека для управления эмуляцией nvram через упомянутый выше драйвер ядра. Более подробное описание и исходный код проекта FIRMADYNE приведены в [7].

Инструмент TriforceAFL [12] сам по себе не является ни фаззером, ни фреймворком для полносистемной эмуляции IoT-устройств, а представляет из себя механизм для фаззинга ядра Linux с использованием AFL. Данный механизм реализован путем добавления в код эмулятора QEMU функционала, реализующего, во-первых, несуществующие инструкции для архитектур x86 и ARM с целью передачи через них мутированных данных от AFL внутрь гостевой системы на системные вызовы, во-вторых, fork-сервер внутри QEMU, и в-третьих, сбор карты покрытия AFL внутри QEMU. Поддерживается только версия QEMU 2.3. Для

организации фаззинга ядра Linux с использованием TriforceAFL необходимо на целевом ядре запустить корневую ФС, содержащую в качестве процесса init так называемый user-mode драйвер фаззинга. Этот драйвер содержит несуществующие инструкции процессора, выполнение которых приводит к определенной последовательности действий внутри IoT-устройства и QEMU: запуск fork-сервера; получение мутированных данных от фаззера; начало и завершение выполнения целевого кода для фаззинга. Этот код представляет собой обертку, распределяющую мутированные данные по параметрам целевого системного вызова. Примеры организации фаззинга с использованием TriforceAFL приведены в [12]. Также следует отметить существенную особенность TriforceAFL – фаззинг выполняется в RAM-памяти, на каждом тестовом запуске гостевой системе недоступен жесткий диск, что годится для фаззинга ядра, но может существенно повлиять на достоверность результатов фаззинга в случае использования указанного подхода к фаззингу user-mode кода.

Инструмент FIRM-AFL [8] предназначен непосредственно для фаззинга IoT-устройств. Новые подходы, используемые в данном инструменте, прежде всего направлены на оптимизацию по скорости процесса фаззинга. В качестве фаззера используется AFL. Полносистемная эмуляция IoT-устройств выполняется с использованием FIRMADYNE [7], за исключением того, что выбрана более новая версия ядра, на которой запускаются целевые устройства. Также, как и в случае TriforceAFL [12], используется встроенная в QEMU карта покрытия AFL, однако в отличие от него, используется не встроенный в QEMU fork-сервер, а механизм легковесных снимков (snapshots), при этом выбор момента сохранения состояния гостевой виртуальной машины, так же, как и другие вспомогательные моменты, выполняется с использованием DECAF [15] и встроенной в нее интроспекции. Одним из средств оптимизации по скорости является использование дополненной эмуляции, суть которой заключается в том, что в процессе инициализации целевого IoT-устройства выполняется полносистемная эмуляция, а после того, как гостевая система перешла в нужную точку, делается снимок ВМ, и фаззинг запускается в режиме user-mode эмуляции, поскольку выполняется тестирование прикладных программ. При этом в некотором файле хранится состояние полносистемного контекста, используемого целевым приложением, а системные вызовы транслируются в ядро хостовой системы. Такой подход, со слов авторов, позволяет значительно увеличить скорость фаззинга. В [8] указано, что данный проект поддерживает архитектуры MIPS и ARM, однако по факту анализ кода из открытого репозитория проекта показал, что в работоспособном состоянии находится только код для архитектуры MIPS, а код для ARM не тестировался. При этом анализ кода также показал, что реализация довольно сильно привязана к архитектуре, и добавление, например, поддержки архитектуры x86 представляется трудоемкой задачей.

3. Инструмент ELF

Инструмент ELF представляет собой инструментальную платформу на основе полносистемной эмуляции QEMU, обеспечивающую внесение различных инструментов анализа (в том числе фаззеров) в штатную среду функционирования программного кода IoT-устройств. Как правило, для запуска дистрибутива ОС Linux (в том числе и для IoT-систем) в среде виртуализации QEMU нужен образ ФС, ядро, образ начальной корневой ФС (initrd) и загрузчик. Для работы с ELF требуется предварительно выделить все эти компоненты из IoT-устройства. Условно такой подход можно назвать – «разделяй и властвуй», т.к. в этом случае дополнительный инструментальный код может быть добавлен в любой из указанных компонентов в зависимости от решаемой задачи.

В некоторых случаях удаётся восстановить версию и конфигурацию ядра IoT-системы. Тогда ядро может быть заново собрано из исходных кодов с добавлением различных модулей, полезных для отладки и анализа.

Если образ ФС позволяет вносить изменения, то очевидным образом, инструментальные средства добавляются на уровне ФС.

Однако в ходе проведенных исследований было установлено, что наиболее простым способом является внесение в целевую среду дополнительных инструментальных средств анализа через начальную корневую ФС. При таком способе инжектирования инструментальные средства остаются прозрачны для программного кода IoT-устройства и не влияют на его функциональность.

В состав ELF входит набор скриптов, обеспечивающих автоматическую генерацию начальной корневой ФС при каждом запуске QEMU в соответствии с запускаемым сценарием теста (выбор инструментальных средств, входные параметры и т. п.).

На рис. 1 приведена схема работы ELF и его основные компоненты.

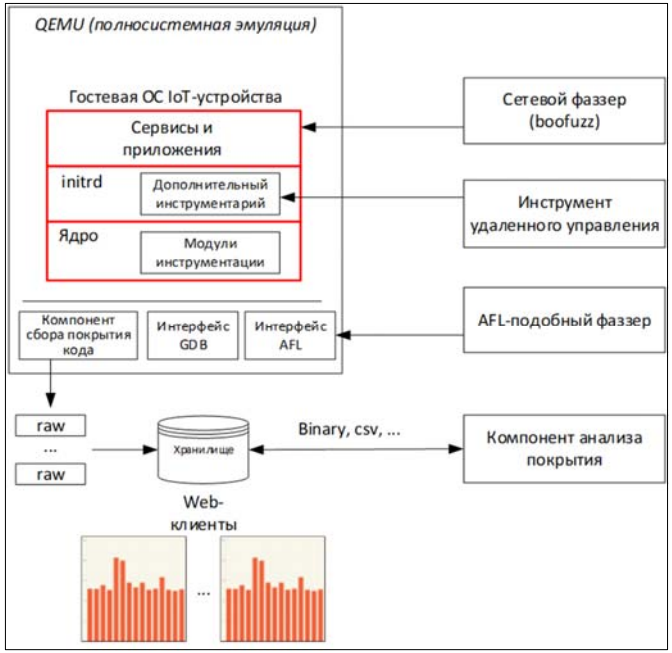

Рис. 1. Основные компоненты инструмента ELF
Fig. 1. The main components of the ELF tool

В настоящее время в ELF авторами реализована поддержка AFL-подобных [16] и сетевых фаззеров Sulley/Boofuzz [17, 18] для архитектур x86_64 и ARM. Для использования режима отладки QEMU может быть запущена со встроенным GDB-сервером. Данный режим используется для работы с такими фаззерами, как Sulley/Boofuzz.

Далее рассматриваются основные компоненты ELF.

а) *QEMU*. Основная часть ELF – это полносистемный эмулятор QEMU. В качестве базовой версии выбрана специализированная версия QEMU ИСП РАН [19]. В эту версию QEMU авторами внесены следующие дополнительные компоненты:

- интеграции с AFL фаззером (частично заимствованно из проекта TriforceAFL);
- сбора покрытия кода;
- работы со снимками.

В текущей версии ELF интегрирован только с AFL фаззером через интерфейс AFL и драйвера фаззинга, по аналогии как это сделано в TriforceAFL. Схематично взаимодействие AFL и QEMU в инструменте ELF показано на рис. 2.

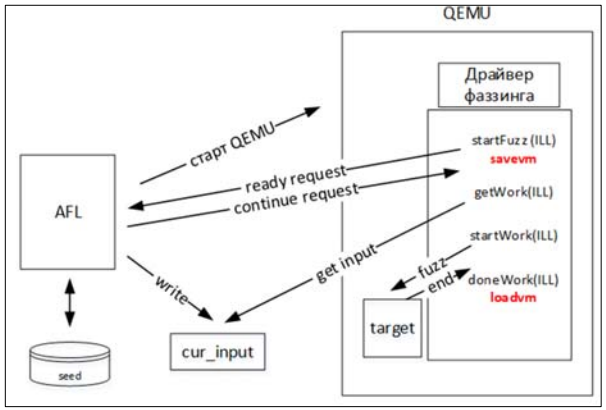

Рис. 2. Протокол взаимодействия AFL и QEMU
Fig. 2. Protocol of interaction AFL and QEMU

В драйвер фаззинга вставляется специальная несуществующая машинная инструкция, которая обрабатывается QEMU особым образом и сигнализирует в какой момент начинать фаззинг. Фаззер AFL запускается непосредственно на хосте, где и генерирует входные данные для исследуемого приложения. Эти данные транслируются в QEMU и уже в среде IoT-устройства передаются приложению. Таким образом, нет необходимости портировать фаззер для работы в среде IoT-устройства. Аналогичным способом можно адаптировать не только AFL, но и другие существующие фаззеры без необходимости выполнять их сборку кросс-компилятором под другую архитектуру.

В инструменте ELF авторами реализован собственный механизм подсчета покрытия кода, не связанный с каким-либо фаззером. Компонент сбора покрытия кода выполняет фиксацию адресов выполненных базовых блоков. Данный механизм, помимо традиционной оценки процесса фаззинга конкретным фаззером, может использоваться также для сравнения различных фаззеров, работающих на платформе ELF над одной и той же целью.

В отличие от TriforceAFL, в инструменте ELF в QEMU для фаззинга применяется не fork-сервер, а механизм снимков (vm-snaphots). Несмотря на то, что последний является более медленным, использование fork-сервера предполагает тестирование исключительно в ram-памяти после порождения дочернего процесса. Особенности реализации fork-сервера в QEMU таковы, что подобный подход влечет за собой отсутствие фиксации тех ошибок, на возникновение которых влияют данные и код считываемый с жесткого диска ВМ. Такое поведение негативно влияет и на качество проводимого фаззинга.

б) *Компонент анализа покрытия*. В ходе работы ELF на хостовой машине сохраняется информация о полученном покрытии, а именно: 1) виртуальные адреса всех блоков кода (как для ядра, так и пользовательского режима), 2) карта покрытия AFL (для совместимости с этим фаззером), 3) в том случае если возможно перекомпилировать ядро (с опцией GCOV_KERNEL), то сохраняется покрытие ядра в формате gcov.

Компонент анализа покрытия агрегирует указанные виды собираемого покрытия, обрабатывает и отображает. Для пользователя покрытие представляется в базе данных Ida Pro для определенного модуля или через Web-браузер для всей системы в целом с разбиением по коду пользовательского режима и коду ядра (рис. 3).

Рис. 3. Отображение покрытия кода в инструменте ELF: в базе данных Ida Pro и в Web-интерфейсе для ядра и пользовательского режима

Fig. 3. Displaying code coverage in the ELF tool: in the Ida Pro database and in the web interface for kernel and user mode

в) *Дополнительный инструментарий*. Данный компонент ELF включает в себя дополнительные usermode-модули, разработанные авторами для управления и тестирования:

- conroot – это инструмент, который обеспечивает удаленное управление посредством shell-команд, позволяя выполнять команды в ОС IoT-устройства или запускать приложения из набора дополнительного инструментария;
- драйверы для фаззинга – дополнительные usermode-библиотеки, которые определяют цель фаззинга и позволяют проводить тестирование не только отдельных утилит, но и библиотек или драйверов в ОС IoT-устройства;
- набор утилит для работы с покрытием;
- тестовые утилиты – набор приложений с внесенным ошибками для самотестирования.

г) *Отладочные инструменты*. Для анализа ОС IoT-устройства добавляются такие утилиты, как gdb, gdbserver, strace и т.п. Кроме того, если есть необходимость, то возможно включить в сборку специальный собранный busybox, расширяющий штатный функционал IoT-устройства.

д) *Интерфейсом инструмента ELF* для взаимодействия с пользователем является набор скриптов, который обеспечивают не только запуск тестов, различных фаззеров, но и выполняет при необходимости пересборку всех компонент в зависимости от сценария тестирования. Тем самым исключаются ошибки в работе ELF, связанные с рассогласованностью отдельных компонентов и модулей.

4. Синтетические тесты

В данном разделе приводятся результаты работы инструмента ELF на различных синтетических тестах. Для проверки и контроля работоспособности инструмента ELF в его состав входит три группы синтетических тестов:

- тесты на основе Juliet v1.3 [20];
- тесты от Google на основе fuzzer-test-suite [21];
- сетевой сервер vulnserver, основанный на проверочном тесте для Boofuzz [22].

Для первого и второго набора тестов созданы драйвера фаззинга. С их помощью осуществляется тестирование AFL-подобным фаззером. Третий тест может быть выполнен большинством сетевых фаззеров. В проводимом исследовании в качестве сетевого фаззера выбран Boofuzz.

Первый набор тестов из Juliet v1.3 сформирован из 105 тестов с различными ошибками переполнения буфера: CWE-121 (Stack Based Buffer Overflow), CWE-122 (Heap Based Buffer

Overflow), CWE-124 (Buffer Underwrite), CWE-369 (Divide by Zero) и CWE-476 (NULL Pointer Dereference). Данный набор тестов предназначен для проверки возможности успешного фаззинга и нахождения различного типа ошибок, регистрации аварийных завершений в среде исследуемого IoT-устройства. Для каждого теста в течение нескольких минут должно быть найдено хотя бы одно аварийное завершение. Успешное выполнение данного набора тестов демонстрирует, что стенд с исследуемым IoT-устройством корректно настроен и находится в работоспособном состоянии.

Второй набор тестов (fuzzer-test-suite) предназначен для определения скорости работы фаззеров и получаемого ими покрытия. Каждый тест представляет собой библиотеку, которая содержит известные ошибки, выявленные ранее. С помощью кросс-компилятора, библиотеки из состава fuzzer-test-suite собираются для работы в среде IoT-устройства. Данный набор тестов может также использоваться для сравнения работы различных фаззеров. В том числе, в ходе проведения исследований было выполнено сравнение двух версий инструмента ELF, реализующих разные подходы — с использованием fork-сервера и механизма снимков ВМ: 1) на основе QEMU версии 2.3 с fork-сервером (TriforceAFL) и 2) на основе QEMU версии 5.2 с механизмом снимков ВМ. Начальные входные данные (seeds) использовались одинаковые для каждой версии.

В табл. 1 представлены результаты фрагмента испытаний на 4-х библиотеках из состава fuzzer-test-suite. В столбцах, отмеченных «v1», представлен результат работы ELF с Qemu 2.3 и fork-сервером (TriforceAFL), в столбцах с отметкой «v2» — результат работы ELF с Qemu 5.2 и механизмом снимков, а в столбцах с отметкой «v2-ram» — результат работы ELF с Qemu 5.2 и механизмом снимков с использованием RAM-диска.

Табл. 1. Сравнение производительности AFL фаззеров для Qemu с Fork-сервером

Table 1. Comparison of performance of AFL fuzzers for Qemu with Fork-server

	Время работы (ч.)			Число запусков (млн.)			Средняя скорость (запусков/ с.)			Покрытие (AFL, total paths)		Ошибки	
	v1	v2	v2-ram	v1	v2	v2-ram	v1	v2	v2-ram	v1	v2	v1	v2
libjpeg	113	113	75	16.4	2.71	2.49	40.3	6.7	9.2	701	**1359**	-	-
libc-ares	113	113	75	18.0	2.89	2.62	44.2	7.1	9.7	739	**779**	-	-
liblcms	113	113	75	3.94	0.7	0.78	9.7	1.7	2.9	212	**605**	-	+
libproj4	113	113	75	15.9	2.77	2.57	39	6.8	9.5	1629	**1817**	-	-

В результате эксперимента был получен следующий результат – при бо́льшем числе запусков (скорость работы QEMU с fork-сервером приблизительно в 6 раз выше, чем у QEMU с механизмом снимков ВМ) ELF-версия с fork-сервером (от TriforceAFL) показывает значительно худший результат по покрытию. Заметим, что в столбце «Покрытие (AFL, total paths)» приведены результаты учёта работы всего пользовательского кода в системе, включая все фоновые процессы. Исследование показало, что при порождении дочернего процесса QEMU 2.3 (TriforceAFL) с использованием вызова fork теряется связь с виртуальным жестким диском. Такое поведение существенно влияет на логику работы процессов, использующих механизмы файлового ввода/вывода. В ряде случаев это приводит к тому, что версия ELF с QEMU 2.3 (TriforceAFL) пропускает аварийные завершения, которые были зафиксированы версией ELF с QEMU 5.2 (механизм снимков), поскольку часть кода становится недостижима. Таким образом, при тестировании библиотек в ELF с QEMU 5.2 (механизм снимков) покрытие увеличилось в случае libjpeg на 94%, libc-ares – на 5%, liblcms – на 185%, libproj4 – на 12 %.

Кроме того, версия ELF с механизмом снимков ВМ была адаптирована к работе с RAM-дисками, что увеличило скорость работы в среднем в 1.5 раза (см. столбцы v2-ram в табл. 1) и позволило получить те же результаты за 75 ч.

Третий набор тестов – это сетевой сервер vulnserver, в котором реализуется HTTP-подобный протокол. В обработчиках запросов сервера намеренно заложены ошибки. Запуск теста позволяет проверить работу сетевого стека протокола IoT-устройства. Регистрация аварийных завершений обеспечивается через интерфейс gdb QEMU.

5. Заключение

Приведенная схема использования инструмента ELF позволяет упростить процесс подготовки программного кода IoT-устройства к динамическому анализу, в том числе с применением различных существующих фаззеров.

Полученные замеры скорости работы и покрытия уже в текущей версии позволяют проводить фаззинг за приемлемое время, при этом в процессе фаззинга обеспечивается стабильная работа IoT-устройства и всего комплекса в целом. Также в инструменте ELF, в отличие от TriforceAFL, решена задача синхронизации данных по жесткому диску во время фаззинга.

С целью усовершенствования инструмента ELF планируется рассмотреть возможность добавления следующего функционала:

- расширение номенклатуры поддерживаемых фаззеров;
- добавление возможности масштабирования;
- использование декомпилятора для представления покрытия в формате lcov, даже если исходные коды отсутствуют;
- инжектирование средств бинарной динамической инструментации.

В первую очередь, планируется добавить фаззер libFuzzer для тестирования кода библиотек, входящих в состав IoT-устройства, а также один из фаззеров ядра ОС Linux.

Текущая версия позволяет запускать несколько копий инструмента ELF для проведения фаззинга. Однако, настройка каждого экземпляра тестируемой ВМ, их клонирования, указание заданий для фаззинга (цели, начальные данные) — всё это выполняется вручную. Для удобства использования и быстрого масштабирования указанные задачи в следующей версии ELF будут выполняться автоматизировано.

Несмотря на то, что настоящая версия ELF поддерживает интеграцию с базой данных Ida Pro, очевидно, что для анализа более удобен формат lcov, в котором покрытие предоставляется в виде исходных кодов с указанием блоков языка C, выполненных в ходе фаззинга. В следующие версии ELF планируется добавить разметку кода на уровне языка C. Поскольку ядро ОС Linux, многие сервисы и приложения пользовательского режима в IoT-устройствах написаны на языке C, то можно предполагать, что декомпилированный код будет функционально близок к оригинальному.

Инжектирование современных средств бинарной динамической инструментации позволит расширить класс фиксируемых ошибок на IoT-устройстве. Основная сложность заключается в том, что запуск таких средств анализа в среде IoT-устройств часто затруднителен в силу ограниченности функционала среды выполнения.

Список литературы / References

[1]. Binwalk. URL: https://github.com/ReFirmLabs/binwalk, accessed 17.10.2021.
[2]. C. Simmonds. Mastering Embedded Linux Programming. Second Edition. Packt Publishing, 2017, 983 p.
[3]. Pin - A Dynamic Binary Instrumentation Tool. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, accessed 17.10.2021.
[4]. DynamoRIO. https://dynamorio.org/, accessed 17.10.2021.

[5]. M. Sharma, N. Agarwal, and S.R.N. Reddy. Design and development of daughter board for USB-UART communication between Raspberry Pi and PC. In Proc. of the International Conference on Computing, Communication & Automation, 2015, pp. 944-948
[6]. F. Bellard. QEMU, a fast and portable dynamic translator. In Proc. of the USENIX Annual Technical Conference, 2005, pp. 41-46.
[7]. D. D. Chen, M. Egele et al. Towards automated dynamic analysis for Linux-based embedded firmware. In Proc. of the Network and Distributed System Security Symposium *(*NDSS*)*, 2016, pp. 1-16.
[8]. Y. Zheng, A. Davanian et al. FIRMAFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In Proc. of the 28th USENIX Security Symposium (USENIX Sec), 2019, pp. 1099-1114.
[9]. LuaQEMU. URL: http://github.com/Comsecuris/luaqemu, accessed 17.10.2021.
[10]. J. Zaddach, L. Bruno et al. Avatar: A framework to support dynamic security analysis of embedded systems firmwares. In Proc. of the Network and Distributed System Security Symposium *(*NDSS*)*, 2014, pp. 1-16.
[11]. M.Muench, D. Nisi et al. Avatar2: A Multi-Target Orchestration Platform. In Proc. of the Workshop on Binary Analysis Research, 2018, pp. 1-11.
[12]. TriforceAFL. URL: https://github.com/nccgroup/TriforceAFL, accessed 17.10.2021.
[13]. J. Chen, W. Diao et al. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2018, pp. 1-15.
[14]. Metasploit. URL: https://www.metasploit.com/, accessed 17.10.2021.
[15]. A. Henderson, LK. Yan et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. IEEE Transactions on Software Engineering, vol. 43, issue 2, 2017, pp. 164-184.
[16]. M. Zalewski. American Fuzzy Lop. URL: http://lcamtuf.coredump.cx/afl/, accessed 17.10.2021.
[17]. J. Pereyda. Boofuzz Documentation, Release 0.4.0, 2021. URL: https://buildmedia.readthedocs.org/media/pdf/boofuzz/latest/boofuzz.pdf, accessed 17.10.2021.
[18]. Sulley. URL: https://github.com/OpenRCE/sulley, accessed 17.10.2021.
[19]. P. Dovgalyuk. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proc. of the 16th European Conference on Software Maintenance and Reengineerin, 2012, pp. 553-556.
[20]. fuzzer-test-suite. https://github.com/google/fuzzer-test-suite, accessed 17.10.2021.
[21]. NIST Test Suites. https://samate.nist.gov/SRD/testsuite.php, accessed 17.10.2021.
[22]. CTP/OSCE Prep – Boofuzzing Vulnserver for EIP Overwrite. https://h0mbre.github.io/Boofuzz_to_EIP_Overwrite/, accessed 17.10.2021.

Информация об авторах / Information about authors

Роман Дмитриевич КОВАЛЕНКО – научный сотрудник отдела компиляторных технологий. Сфера научных интересов: информационные технологии, безопасность ПО, анализ бинарного кода, системное программирование, фаззинг, исследования программ.

Roman Dmitrievich KOVALENKO – Researcher, Compiler Technologies Department. Research interests: information technology, software security, binary code analysis, system programming, fuzzing, program exploration.

Алексей Николаевич МАКАРОВ – научный сотрудник отдела компиляторных технологий. Сфера научных интересов: информационные технологии, безопасность ПО, анализ бинарного кода, системное программирование, фаззинг, исследования программ.

Alexey Nikolaevich MAKAROV – Researcher, Compiler Technologies Department. Research interests: information technology, software security, binary code analysis, system programming, fuzzing, program exploration.