ПОДХОДЫ К ОЦЕНКЕ ПОВЕРХНОСТИ АТАКИ И ФАЗЗИНГУ ВЕБ-БРАУЗЕРОВ

Козачок А.В.[1], Николаев Д.А.[2], Ерохина Н.С.[3]

Цель работы: *разработка подхода к определению поверхности атаки на основе анализа разности покрытий кода и его применение для анализа веб-браузеров.*

Метод исследования: *использование инструментирующего компилятора для анализа покрытия кода в зависимости от входных данных. Предложенный подход позволяет оценить взаимосвязь входных данных, обрабатываемых анализируемым приложением, с кодом программы за счет вычисления разности покрытий и исключения из анализа модулей, вызываемых независимо от входных данных.*

Результаты исследования: *рассмотрены существующие общие подходы к фаззингу программного обеспечения, а также особенности подходов к фаззингу веб-браузеров. В общем случае фаззинг программного обеспечения выполняется одним из трех методов: черного, серого, белого ящика. Базовым критерием различия указанных методов является полнота информации об исходном коде тестируемого программного обеспечения. Фаззинг веб-браузеров можно разделить на статический и динамический. Подходы к фаззингу сложного программного обеспечения можно разделить на две группы: анализ монолитного приложения, фаззинг отдельных модулей приложения (интерфейсов библиотек). Различие указанных групп определяется полнотой вовлечения функциональных компонент исследуемого программного обеспечения в процесс тестирования. Каждый из уровней имеет свои достоинства и недостатки. Зачастую эти недостатки могут компенсироваться за счет комбинации фаззинга различных фаззинг-целей. Для корректного определения фаззинг-целей следует выделить поверхность атаки исследуемого программного обеспечения. Авторами предложен подход к оценке поверхности атаки за счет вычисления разности покрытий, он позволяет исключить из анализа модули, вызываемые независимо от входных данных.*

Научная и практическая значимость: *результатов статьи состоит в разработке нового подхода к определению поверхности атаки на основе анализа разности покрытий кода анализируемого приложения в зависимости от подаваемых на вход данных, и позволяющего исключить из анализа модули, вызываемые независимо от входных данных.*

Ключевые слова: *веб-браузер, JavaScript движок, покрытие кода, программные дефекты, уязвимости программного обеспечения, фаззинг-тестирование.*

DOI: 10.21681/2311-3456-2022-3-32-43

Введение

Всеобщая цифровизация привела к тому, что современное общество невозможно без глобальной сети Интернет. Интернет включает в себя тысячи сетей, такие как: сети индустриальных предприятий и корпораций всех видов, коммерческих предприятий, а также сети вооруженных сил и правительственных организаций. Безопасность доступа в сеть пользователей зачастую базируется на безопасности браузера и применяемых технологий для хранения, обработки и передачи конфиденциальной информации. По статистике веб-браузер является одним из основных инструментов доставки вредоносных программ на компьютеры пользователей.

Наиболее распространенными веб-браузерами являются:

— Chrome от Google;
— Firefox от Mozilla;
— Safari от Apple;
— Edge от Microsoft.

Основное предназначение веб-браузера – предоставлять контент веб-приложений пользователям. Для этого на сервер отправляется запрос, а результат вы-

1 Козачок Александр Васильевич, доктор технических наук, доцент, Академия ФСО России, г. Орел, Россия, E-mail: a.kozachok@academ.msk.rsnet.ru, https://orcid.org/0000-0002-6501-2008

2 Николаев Дмитрий Александрович, Академия ФСО России, г. Орел, Россия, E-mail: mriddi@bk.ru, https://orcid.org/0000-0001-9334-6948

3 Ерохина Наталья Сергеевна, Академия ФСО России, г. Орел, Россия, E-mail: osipova_nc@mail.ru, https://orcid.org/0000-0002-4878-0865

водится в окне веб-браузера. Каждый раз, когда веб-браузер связывается с веб-сервером, последний собирает некоторую информацию о веб-браузере для того, чтобы корректно обработать формирование запрашиваемой страницы. Если вредоносный код был внесен в содержимое веб-приложения, то уязвимости, конкретного веб-браузера, могут позволить этому вредоносному коду выполнить некоторые нелегитимные действия на компьютерах пользователей. Одним из вариантов последствий таких действия является возможность запуска злоумышленником произвольного кода, позволяющего ему выполнять любые действия не только на зараженном компьютере, но и, возможно, на других компьютерах в сети. Одними из наиболее эффективных методов обнаружения уязвимостей в программном обеспечении (ПО) являются статический и динамический анализ кода. В рамках проведения динамического анализа особое место занимает проведение фаззинг-тестирования.

1. Общие подходы к фаззингу программного обеспечения

Дефекты и ошибки в логике программы являются основной причиной возникновения уязвимостей в ПО. В наши дни большинство обнаруживаемых ошибок в программном обеспечении, связанных с удаленным выполнением кода и повышением привилегий, находятся при помощи фаззинга. Фаззинг – это методика тестирования, при которой на вход программы подаются невалидные, непредусмотренные или случайные данные, которые могут привести её к аварийному завершению или неопределённому поведению.

Целью фаззинг-тестирования является решение следующих частных задач:

- поиск программных дефектов;
- исследование уязвимостей методом изучения и эксплуатации ошибок;
- получение информации о внутренней структуре, так как ошибки или иная реакция на фаззинг может раскрывать системную информацию;
- поиск ошибок в логике ПО [1].

С момента своего появления в начале 1990-х годов фаззинг оставался одним из наиболее широко применяемых методов обнаружения уязвимостей безопасности ПО. В 1988 году был создан первый простейший фаззер, предназначенный для командной строки, с целью тестирования надежности приложений под Unix. Он генерировал случайные данные, которые передавались как параметры для других программ до тех пор, пока они не останавливались с ошибкой. Однако современный фаззинг мало похож на свою первоначальную версию. Множеством исследований подтверждено, что подача на вход программы случайных данных малоэффективна [2]. Из-за того, что программы часто проверяют вводимые данные перед синтаксическим анализом и обработкой. Данная проверка выполняется с целью защиты программы от сбоев, вызванных некорректными входными данными и большая часть сгенерированных случайных данных отбрасывается при валидации и не приносит результатов фаззинга.

Выделяют три метода фаззинг-тестирования: метод «черного ящика» (англ. BlackBox fuzzing), метод «серого ящика» (англ. GrayBox fuzzing), а также метод «белого ящика» (англ. WhiteBox fuzzing) [3].

Тестирование методом «черного ящика» – это метод тестирования, основанный на работе исключительно с внешними интерфейсами тестируемой системы. Данный метод используются, когда исходный код тестируемой программы недоступен и нет возможности получения обратной связи по коду.

Фаззерам, работающим по данному методу, зачастую не удается сгенерировать входные данные без обратной связи по коду, обеспечивающие прирост покрытия. Вследствие чего, эффективность такого тестирования во многих случаях является низкой.

Тестирование методом «белого ящика» – метод тестирования программного обеспечения, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Входные значения выбираются, основываясь на знании кода, который будет их обрабатывать. Точно так же известно, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование методом белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов [4]. Фаззер Sage [5] — канонический пример данного метода.

Данный метод подразделяется на три подвида:

- символьное исполнение (англ. Symbolic Execution);
- конколическое исполнение (англ. Concolic Execution)[6];
- отслеживание искажения данных (англ. Data Taint Tracking).

Метод «белого ящика» показал свою неэффективность для сложных программ [7]. Также одним из недостатков является его требовательность к ресурсам.

Тестирование методом «серого ящика» – метод тестирования программного обеспечения, который предполагает, что глубокий анализ программы может быть невозможен или запрещен для выполнения [8]. Вместо этого предполагается, что программа может быть оснащена инструментами, а обратная связь может использоваться для управления генерацией входных данных программы. AFL [9-10] – это канонический пример фаззера, реализующего метод «серого ящика» с обратной связью.

Главное преимущество метода в том, что фаззер получает информацию не только о выводе и аварийном завершении программы, но и о ходе исполнения программы. В общем случае информация может быть любой, но обычно используют такую метрику, как покрытие кода программы. Повышение покрытия кода означает повышение охвата состояний выполнения программы и повышение качества тестирования.

Эффективность этого подхода может быть повышена за счет исходного корпуса, который реализует более крупные части целевой программы, предоставляя более широкие границы для поиска входных данных. Фаззеры, которые полагаются на мутацию начального корпуса для исследования входного пространства, называются фаззерами мутаций. AFL также является каноническим примером данного типа фаззеров. При этом мутации можно комбинировать с генерационными процедурами (AFLSmart [11]).

2. Подходы к фаззингу веб-браузеров

В последние годы наблюдается тенденция внедрения фаззинга в процессы разработки и становится стандартом де-факто для такого большого и сложного программного обеспечения, как веб-браузеры и их ключевых компонент – графических и JavaScript движков. Современные веб-браузеры подвергаются обширному фаззингу для обнаружения уязвимостей. Код Chromium в Google Chrome постоянно обрабатывается командой безопасности Chrome 15 000 вычислительных ядер. Для Microsoft Edge и Internet Explorer, Microsoft проводит глобальное тестирование в процессе разработки продуктов, создавая более 400 миллиардов манипуляций для DOM из 1 миллиарда файлов HTML.

Подходы к фаззингу веб-браузеров определяется следующими особенностями:

- веб-браузер имеет большую кодовую базу и сложную внутреннюю структуру, а также множеством дополнительных модулей;
- входными данными веб-браузеров в основном являются сложно структурированные текстовые данные;
- существует множество различных веб-браузеров имеющие значительные архитектурные особенности;
- не существует универсального инструментария фаззинга совместимого со всеми веб-браузерами.

Рассмотрим известные подходы к фаззингу веб-браузеров. В работе [12] выделяют два типа фаззинга веб-браузеров:

1) Статический.

При статическом фаззинге веб-браузера входные данные генерируются случайным образом перед запуском фаззера, а затем входные данные подаются в веб-браузер, чтобы проверить, произойдет ли сбой при их обработке.

Примером такого фаззера является bf3 – это статический фаззер, который добавляет один элемент и присваивает случайную последовательность в качестве значения в начальном файле. Следовательно, он может генерировать множество входных данных, которые могут передаваться в веб-браузер. Однако этот способ поиска аварийных завершений малоэффективен, поскольку входные данные могут получаться аналогичной или простой структуры.

2) Динамический.

В динамическом фаззинге веб-браузеров обычно используются JavaScript скрипты для постоянного изменения структуры начального входного файла, который представляет собой html-документ. Существует случайное число, определяющее выполняемые процедуры каждый раз, поэтому результат не известен заранее. Алгоритм не остановится, пока веб-браузер не завершит работу или не завершится аварийно. Дополнительно используется объектная модель документа (DOM) для динамического изменения структуры начального входного файла.

Обобщая [12, 13, 14], можно выделить три уровня фаззинга веб-браузеров, определяемых полнотой вовлечения его функциональных элементов:

- фаззинг монолитного приложения;
- фаззинг отдельных модулей приложения (фаззинг интерфейсов библиотек).

Наиболее очевидным подходом является фаззинг монолитного приложения веб-браузера. Существуют специальные инструменты для его реализации, например, для FireFox он может осуществляется с помощью фреймворка Grizzly, а одним из наиболее эффек-

тивных фаззеров является инструмент Domino. Преимущество этого метода заключается в том, что нам доступны все возможности веб-браузера, и тестирование происходит близко к тому, как работают в веб-браузере пользователи. Однако в силу наличия большой и непрерывно развивающейся на протяжении многих лет кодовой базы, фаззинг монолитного приложения является малоэффективным с точки зрения прироста покрытия кода и является самым медленным из рассматриваемых подходов, остановимся на более детальном рассмотрении двух других подходов.

Фаззинг отдельных модулей веб-браузера связан с выделением в его структуре ключевых функциональных компонент и последующим их тестированием по модулям. На рисунке 1 представлена обобщенная структура веб-браузера. Наибольший интерес в качестве целей фаззинга в первую очередь представляют графический и JavaScript движки.

Графический движок (англ. rendering engine или layout engine или browser engine) – это один из базовых элементов веб-браузера. Графический движок отображает на экране содержимое запрашиваемого ресурса. Именно эта часть веб-браузера анализирует полученный HTML или XML, при этом учитывает влияние CSS и JavaScript, а также других объектов, расположенных на веб странице (например, изображения или flash). На основе всех этих данных, движок создает разметку страницы, которую видит пользователь на экране.

Ключевыми компонентами графического движка являются HTML и CSS парсеры, являющиеся сложными программными комплексами и позволяющие графическому движку отобразить документ даже при наличии ошибок в обрабатываемых данных.

Наиболее распространенными движками веб-браузеров являются:

— Trident (веб-браузер Internet Explorer);
— Gecko (веб-браузер FireFox);
— Webkit (веб-браузеры Chrome, Safari);
— Presto (веб-браузеры Opera).

Существует ряд инструментов, основанных на динамическом фаззинге графического движка и его элементов, например, crossfuzz [15], который генерирует чрезвычайно длинные последовательности операций DOM, проверяя возвращенные объекты, выполняя рекурсию и создавая циклические ссылки на узлы. Инструмент ndujafuzz демонстрирует эволюционный подход к фаззингу веб-браузера, основанный на некоторых интерфейсах DOM, представленных спецификациями W3C DOM Level 2 и Level 3. Также широко применяются специализированные фаззеры Domato, Dharma [16] и FreeDom [17], используемые для обнаружения связанных с памятью уязвимостей и дефектов в DOM-реализациях веб-браузеров. Они генерируют структурные данные, которые содержат элементы HTML, CSS и JavaScript, связанный с DOM [18].

JavaScript наряду с HTML и CSS является одним из трех основных технологических языков создания контента в World Wide Web. Большинство веб-приложений используют данный язык, который поддерживают все современные веб-браузеры без использования плагинов с помощью встроенного механизма интерпретации и выполнения JavaScript.

JavaScript-движок – это программа, или, другими словами, интерпретатор, выполняющий код, написанный на JavaScript. Движок может быть реализован с использованием различных подходов: в виде обычного

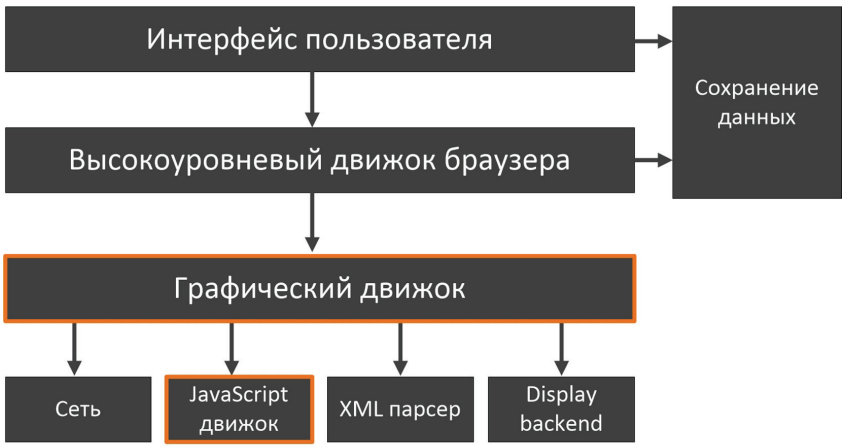

Рис. 1. Обобщенная структура веб-браузера

интерпретатора, или в виде динамического компилятора (JIT-компилятора (англ. Just-In-Time compiler)), который, перед выполнением программы, преобразует исходный код на JavaScript в байт-код [19].

Как видно из таблицы 1, JavaScript-движки чаще всего используются в веб-браузерах. Каждый движок подобен языковому модулю, который позволяет приложению поддерживать определенное подмножество стандартов языка JavaScript.

Таблица 1

Наиболее распространенные JavaScript движки

Название JavaScript движка	Применение
SpiderMonkey	Firefox, the Gecko layout engine, Adobe Acrobat
V8	Google Chrome, Node.js, Opera, MarkLogic
JavaScriptCore (JSC)	WebKit, Safari, Qt5
Chakra	Microsoft Edge

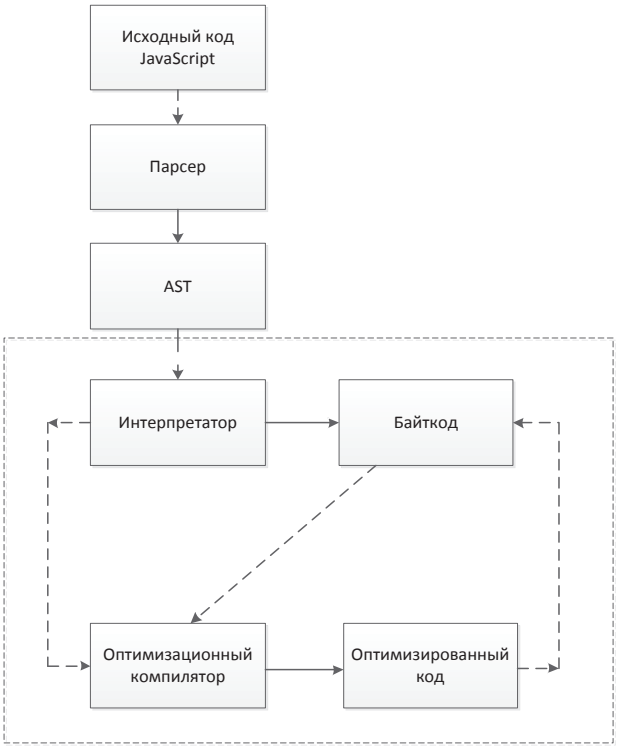

Рис. 2. Обобщенная схема работы JavaScript-движка

Схема работы JavaScript-движка (рис. 2) имеет общий принцип работы, однако в каждом из них есть свои особенности реализации. Рассмотрим различия в структуре различных распространенных движках.

Интерпретатор в движке V8 называется Ignition, он отвечает за генерацию и выполнение байткода. Он собирает данные профилирования, которые могут быть использованы для ускорения выполнения программы на следующем этапе, пока обрабатывается байткод. Сгенерированный байткод и данные профилирования передаются в оптимизирующий компилятор TurboFan для генерации высокооптимизированного машинного кода, основанного на данных профилирования.

JavaScript движок SpiderMonkey от Mozilla, который используется в Firefox и SpiderNode, работает немного иначе. В нем не один, а два оптимизирующих компилятора. Интерпретатор оптимизируется в базовый компилятор (Baseline compiler), который производит первично оптимизированный код. Вместе с данными профилирования, собранными во время исполнения кода, компилятор IonMonkey может генерировать высоко оптимизированный код (англ. heavily-optimized code). Если спекулятивная оптимизация не удается, IonMonkey возвращается к базовому коду (Baseline code) [20].

Chakra – JavaScript движок от Microsoft, используется в Edge и Node-ChakraCore, имеет похожую структуру, в нем используется два оптимизирующих компилятора. Интерпретатор оптимизируется в SimpleJIT, который производит первично оптимизированный код. Вместе с профилирующими данными FullJIT может создавать высоко оптимизированный код.

JavaScriptCore (JSC), JavaScript движок от Apple, который используется в Safari и React Native, имеет три разных оптимизирующих компилятора. LLInt — низкоуровневый интерпретатор, оптимизируется в базовый компилятор, который в свою очередь оптимизируется в DFG (англ. Data Flow Graph) компилятор, а он уже оптимизируется в FTL (англ. Faster Than Light) компилятор.

Указанные различия в структуре движков обоснованы существующим компромиссом. Интерпретатор может быстро обрабатывать байткод, но сам по себе байткод неэффективен. Оптимизирующий компилятор, с другой стороны, работает немного дольше, но производит более эффективный машинный код. Это компромисс между быстрым получением кода (интерпретатор) или же некоторым ожиданием и запуском кода с максимальной производительностью (оптимизирующий компилятор). Некоторые движки выбирают

добавление нескольких оптимизирующих компиляторов с разными характеристиками времени и эффективности, что позволяет обеспечивать наилучший контроль над этим процессом и понимать накладные расходы дополнительного усложнения внутреннего устройства.

Поиск дефектов в движках JavaScript является важной задачей, учитывая широкий перечень приложений, которые могут быть затронуты этими ошибками. Также это является сложной задачей для тестировщика, так как спецификации изначально неполны для обеспечения гибкости разработки.

Основная трудность при фаззинге движков JavaScript состоит в том, чтобы генерировать синтаксически и семантически корректные входные данные, чтобы можно было исследовать различные функциональные возможности. Однако из-за динамической природы JavaScript и особенностей различных движков это сделать довольно сложно. Еще одна проблема заключается в том, что существующие фаззеры не могут генерировать вызовы новых методов, которые не включены в первоначальный исходный корпус или заранее определенные правила, что ограничивает возможности поиска ошибок [21].

Fuzzil[22] и Jsfunfuzz – это распространенные фаззеры для обнаружения уязвимостей в движках JavaScript, которые генерируют структурные данные, содержащие корректный HTML, CSS и JavaScript код, связанный с DOM.

Также примерами фаззеров JavaScript движков являются:

Favocado – это фаззер, использующий подход, ориентированный на фаззинг уровней связывания систем выполнения JavaScript. Favocado может генерировать синтаксически и семантически правильные тестовые примеры JavaScript за счет использования извлеченной семантической информации и тщательного поддержания состояний выполнения [23].

SoFi – фаззер, реализующий новый семантический метод фаззинга. Чтобы гарантировать достоверность сгенерированных тестовых данных, SoFi применяет детальный программный анализ для определения доступных переменных и определения типов этих переменных для мутации [24].

Фаззинг интерфейсов библиотек базируется на декомпозиции программы и фаззинге интересующих изолированных элементов программы по-отдельности. В качестве таких элементов обычно выступают функции (интерфейсы) бибилиотек, которые и будут являться целями фаззинга. Очевидно, что необходимым требованием для такого подхода является наличие исходного кода программы, однако на практике наибольшее распространение получили инструменты семейства Greybox фаззеров: AFL и LibFuzzer[25].

Также важной задачей является определение поверхности атаки для проведения исследований веб-браузера. Так как кодовая база очень обширна и в документации на браузеры зачастую не выделены интерфейсы функций безопасности, данная задача является нетривиальной.

3. Оценка поверхности атаки для фаззинга веб-браузеров

На подготовительном этапе аналитиком должна быть решена задача по определению поверхности атаки (ПА). Под ПА понимается совокупность интерфейсов и реализующих их модулей ПО, через которые могут реализовываться угрозы безопасному функционированию ПО.

Для решения указанной задачи на практике применяются следующие подходы:

- анализ помеченных данных для выделения модулей, участвующих в обработке данных, поступающих от пользователя. Таким образом, с учетом обеспечения полноты возможных вариантов получения и обработки данных от пользователя можно обеспечить определение ПА. Подход на основе отслеживания помеченных данных позволяет определять функции, обращающиеся к помеченным данным, тем самым формируя поверхность атаки. Примером такого инструмента является Natch[4], разработанный Институтом системного программирования им. В.П. Иванникова РАН.
- экспертный анализ ПО на основе анализа исходного кода и документации с целью выделения интерфейсов и модулей ПА.

Указанные подходы обладают достаточно высокой трудоемкостью и предъявляют высокие требования к эксперту. В отличии от рассмотренных выше подходов, для выбора целей можно применить следующее эвристическое правило – количество вызовов функции может косвенно обозначать ее значимость при условии ее определенной цикломатической сложности [26]. Таким образом, множество вызываемых функций при заданных входных данных формирует поверхность атаки, то есть указывает на наличие связи

4 https://www.ispras.ru/technologies/natch/

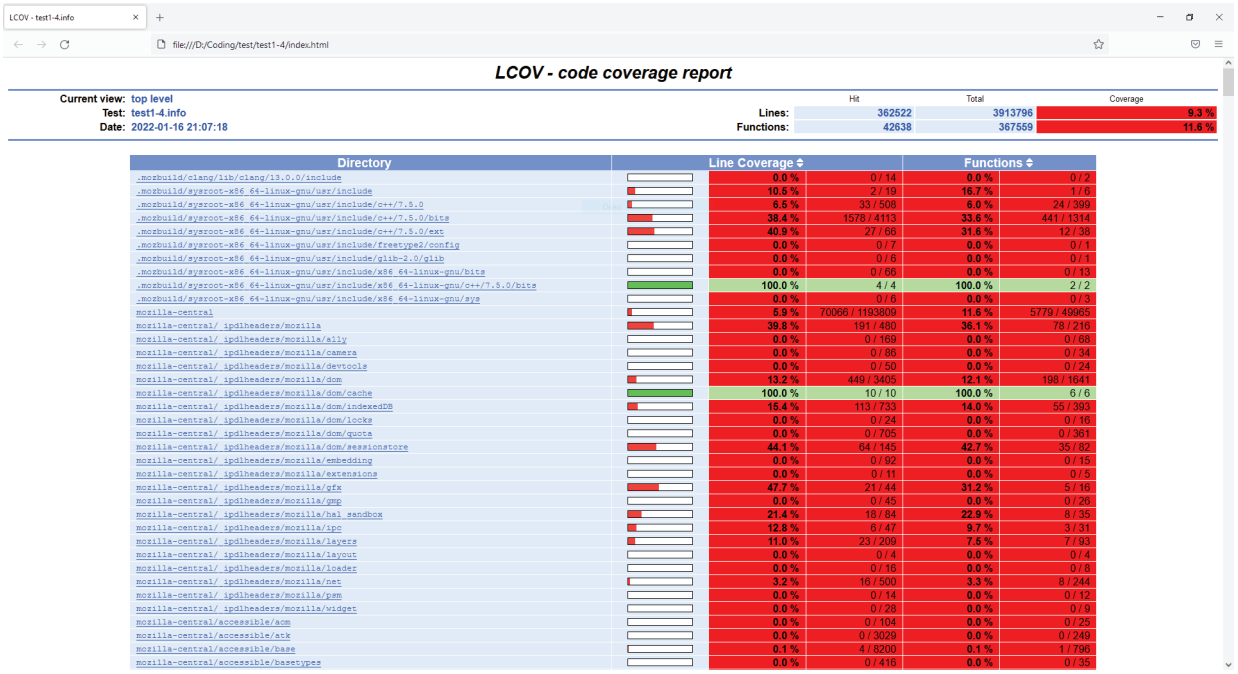

Рис. 3. Пример отчета по покрытию кода, сформированный утилитой lcov для веб-браузера

ее вызова со входными данными, аналогично подходу на основе анализа помеченных данных.

Известны и широко применяются на практике инструменты, позволяющие, в том числе получать информацию по количеству вызовов функций в ходе выполнения исследуемого программного обеспечения (например, для поиска не выполняющегося и недостижимого кода), результат такого анализа называется покрытием кода. Так, соответствующий инструмент включен в состав набора компиляторов GCC – утилита gcov [27]. Для ее использования достаточно скомпилировать программное обеспечение с отключенной оптимизацией и набором необходимых флагов, после чего будут автоматически сгенерированы необходимые файлы, а последующее выполнение программного обеспечения приведет к формированию данных по покрытию кода. Также доступна утилита lcov [28], являющаяся клиентской частью для gcov и конвертирующая результаты покрытия в нужный формат. Она собирает данные от gcov и генерирует HTML страницы, содержащие аннотированный исходный код с данными по покрытию (рис. 3).

Следует учитывать, что покрытие формируется для конкретного набора входных данных ПО, а также может зависеть от состояния среды функционирования [29].

Для сложного приложения, принимающего различные входные данные, как веб-браузер, целесообразно выделить отдельные функциональные элементы, зависящие от конкретного типа входных данных, такую процедуру назовем расчетом разности покрытия кода. С этой целью авторами был разработан Python модуль[5], на вход которому подаются два файла содержащие результаты анализа покрытия для разных условий запуска программного обеспечения, а на выходе формируется файл, содержащий разность покрытий. Заложенная в скрипт логика вычитания представлена в таблице 2. Указанный модуль позволяет исключить из анализа модули, вызываемые независимо от входных данных.

Отметим, что зачастую исследователями используется механизм сложения покрытий, для вычислений суммарного, например, по результатам обработки корпуса данных, такой функционал реализован в lcov. Однако реализации логики вычитания покрытия кода для исключения участков кода, вызываемых независимо от поданных на вход данных, авторами обнаружено не было.

На рисунке 4 представлен пример применения разработанного скрипта для расчета разности покрытия: содержание трех фалов a.json, b.json, c.json с указанием названия и количеством вызовов функций и строк кода. Файл c.json является результатом расчета разности покрытия для a.json и b.json.

5 https://github.com/mriddi/json_diff/blob/main/json_diff.py

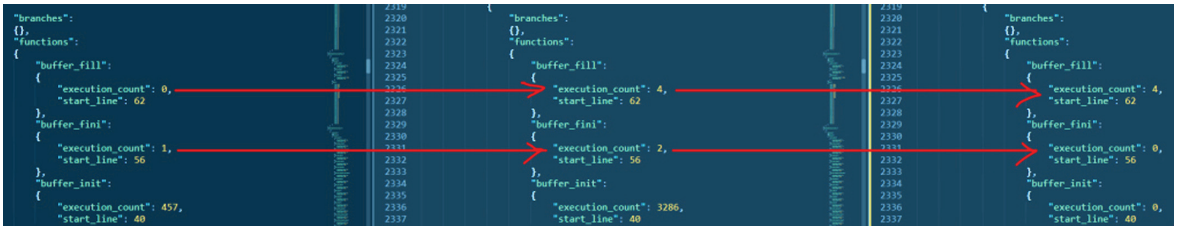

Рис. 4. Пример расчет разности покрытия кода

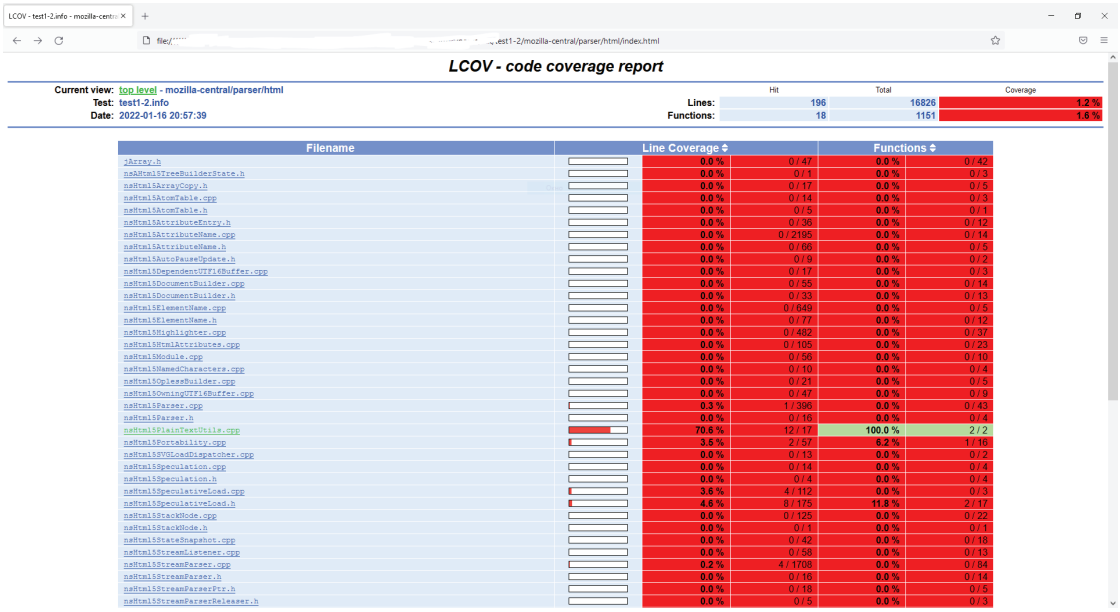

Рис. 5. Отчет по результатам расчета разности покрытий кода

Рис. 6. Внедрение функции abort() для выявления аварийного завершения

Таблица 2

Логика вычитания при расчете разности покрытия

Количество вызовов функции в файле покрытия кода в условиях №1	Количество вызовов вызова функции в файле покрытия кода в условиях №2	Количество вызовов функции в файле разности покрытия кода
0	0	0
0	y	y
x	0	0
x	y	0

В качестве тестируемого ПО был выбран веб-браузер FireFox. Были рассмотрены следующие тесты:

1) запуск FireFox и его завершение без осуществления каких-либо действий;

2) запуск FireFox, открытие веб-страницы http://ifconfig.me/ip (без активного контента), завершение FireFox.

Целью являлось определение кода, вызываемого при рендеринге веб-страницы и последующая модификация кода для выявления аварийного завершения и подтверждения его достижимости.

На первом этапе была осуществлена компиляция приложения с возможностью получения покрытия кода, были выполнены тесты, для сформированных отчетов рассчитана разность покрытия кода и сгенерированы HTML-страницы с отчетом (рис. 5).

При этом под поверхностью атаки будем понимать модули и интерфейсы, покрытие по которым оказалось ненулевым по результатам расчета разности покрытий.

На втором этапе из сформированной поверхности атаки была выбрана функция *NewLinkAttributes()* файла *nsHtml5PlainTextUtils.cpp*, вызываемая при открытии страницы с *plaintext.css*. Далее в код был интегрирован вызов функции *abort()* (рис. 6) и осуществлена компиляции приложения.

На третьем этапе выполнен второй тест, в результате чего произошло аварийное завершение вкладки веб-браузера.

Преимуществом предложенного подхода к определению поверхности атаки с использованием разности покрытий кода состоит в простоте его реализации при наличии исходного анализируемого ПО. Он также позволяет сформировать перечень потенциальных целей для фаззинга, однако у такого подхода есть недостатки:

— он не позволяет анализировать связь параметров функций с входными параметрами программы;

— он требует подготовки набора входных данных программы, обеспечивающих выполнение специфичных блоков кода.

Указанные недостатки частично устраняются подходом на основе отслеживания помеченных данных, позволяющим определять функции, обращающиеся к помеченным данным, тем самым формируя поверхность атаки [30].

Выводы

Фаззинг-тестирование сложного программного обеспечения с большой кодовой базой, такого как веб-браузеры, является актуальной и трудоемкой задачей. В работе рассмотрены существующие общие подходы к фаззингу программного обеспечения, а также особенности подходов к фаззингу веб-браузеров. Фаззинг программного обеспечения может выполняться одним из трех методов: черного, серого, белого ящика. Наиболее эффективным является подход на основе метода серого ящика (семейство фаззеров AFL и libfuzzer). Подходы к фаззингу сложного программного обеспечения можно разделить на две группы: анализ монолитного приложения, фаззинг отдельных модулей приложения (интерфейсов библиотек). Различие указанных групп определяется полнотой вовлечения функциональных компонент исследуемого программного обеспечения в процесс тестирования. Каждый из уровней имеет свои достоинства и недостатки. Зачастую эти недостатки могут компенсироваться за счет комбинации фаззинга различных фаззинг-целей. Для корректного определения фаззинг-целей следует выделить поверхность атаки исследуемого программного обеспечения. Авторами предложен подход к оценке поверхности атаки за счет вычисления разности покрытий, который позволяет исключить из анализа модули, вызываемые независимо от входных данных.

Литература

1. Козачок, А. В., Козачок, В. И., Осипова, Н. С., Пономарев, Д. В. Обзор исследований по применению методов машинного обучения для повышения эффективности фаззинг-тестирования // Вестник ВГУ. Серия: Системный анализ и информационные технологии, 2021 (4), С. 83-106, DOI: 10.17308/sait.2021.4/3800.
2. Li J., Li J., Zhao B., Zhang C. Fuzzing: a survey // Cybersecurity, 2018, Vol. 1, No 1, p. 6, DOI: 10.1186/s42400-018-0002-y.

3. Gopinath R., Görz P., Groce A. Mutation Analysis: Answering the Fuzzing Challenge // arXiv preprint arXiv:2201.11303, 2022, DOI: /10.48550/arXiv.2201.11303.
4. Bounimova E., Godefroid P., Molnar D. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production // In Proceedings of the International Conference on Software Engineering, pp. 122–131, 2013, DOI: 10.1109/ICSE.2013.6606558.
5. Godefroid, Michael Y. Levin, David Molnar. SAGE: Whitebox Fuzzing for Security Testing // Communications of the acm, 2012 (3), pp. 40-44 doi:10.1145/2093548.2093564.
6. Debnath Mukta, Basak Chowdhury Animesh, Saha Debasri, Sur Kolay Susmita. FuCE: Fuzzing+Concolic Execution guided Trojan Detection in Synthesizable Hardware Designs, 2021, DOI: 10.48550/arXiv.2111.00805.
7. Muhammad Noman Khalid, Muhammad iqbal, Kamran Rasheed, Malik Muneeb Abid. Web Vulnerability Finder (WVF): Automated Black-Box Web Vulnerability Scanner // I.J. Information Technology and Computer Science, 2020 (4), pp. 38-46, DOI: 10.5815/ijitcs.2020.04.05.
8. Danyang Zhao. Fuzzing Technique in Web Applications and Beyond // Journal of Physics: Conference Series 1678(1), 2020, doi: 10.1088/1742-6596/1678/1/012109.
9. Зимин Е.Е. Методика фаззинг-тестирования кода с помощью AFL // Безопасные информационные технологии. Сборник трудов Одиннадцатой международной научно-технической конференции, Издательство: МГТУ имени Н.Э. Баумана, 2021, С. 124-129.
10. Тронов К. А., Белов Ю.С. Оптимизация инструментария AFL для лучшего покрытия кода при работе со специфическими данными // E-Scio, ФГБОУ ВО «МГТУ имени Н.Э. Баумана, 2021, № 5 (56), С. 566-571.
11. Pham V. T. et al. Smart greybox fuzzing // IEEE Transactions on Software Engineering, 2019, Т. 47, №. 9, pp. 1980-1997, DOI: 10.1109/TSE.2019.2941681.
12. Ying-Dar Lin, Feng-Ze Liao, Shih-Kun Huang, Yuan-Cheng Lai. Browser Fuzzing by Scheduled Mutation and Generation of Document Object Models // The 49th IEEE International Carnahan Conference on Security Technology, 2020, DOI: 10.1109/CCST.2015.7389677.
13. Sablotny M., Jensen B. S., Johnson C. W. Recurrent neural networks for fuzz testing web browsers // International Conference on Information Security and Cryptology, Springer, Cham, 2018, pp. 354-370, DOI: 10.1007/978-3-030-12146-4_22.
14. Eberlein M. et al. Evolutionary grammar-based fuzzing // International Symposium on Search Based Software Engineering, Springer, Cham, 2020, pp. 105-120, DOI: 10.1007/978-3-030-59762-7_8.
15. Manes V. J. M., Han H., Cha S. K. [et al.] The Art, Science, and Engineering of Fuzzing: A Survey // IEEE Transactions on Software Engineering, 2021, Vol. 47, No 11, pp. 2312-2331, DOI: 10.1109/TSE.2019.2946563.
16. Костандян В., Маргаров Г., Педроса Т., Жуан Родригес П.. Фаззинг в кибербезопасности (обзор) // Новое в российской электроэнергетике, 2020, № 9, С. 17-30.
17. Xu W., Park S., Kim T. FREEDOM: Engineering a State-ofthe-Art DOM Fuzzer // Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event USA: ACM, 2020, pp. 971–986. DOI: 10.1145/3372297.3423340.
18. Chaofan Shou, ˙Ismet Burak Kadron, Qi Su, and Tevfik Bultan. CorbFuzz: Checking Browser Security Policies with Fuzzing // University of California, Santa Barbara, 2021, DOI: 10.48550/arXiv.2109.00398.
19. Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing // In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), 2021, Virtual, Canada. ACM, New York, NY, USA, DOI: 10.1145/3453483.3454054.
20. Agrawal V., Rastogi R., Tiwari D. C. Spider monkey optimization: a survey // International Journal of System Assurance Engineering and Management, 2018, Т. 9, №. 4, pp. 929-941, DOI: 10.1007/s13198-017-0685-6.
21. Tian Y., Qin X., Gan S. Research on Fuzzing Technology for JavaScript Engines // The 5th International Conference on Computer Science and Application Engineering, 2021, pp. 1-7, DOI: 10.1145/3487075.3487107.
22. Groß S. Fuzzil: Coverage guided fuzzing for javascript engines // Department of Informatics, Karlsruhe Institute of Technology, 2018.
23. Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos , Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupe, and Yan Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases // Network and Distributed Systems Security (NDSS) Symposium, 2021, DOI: 10.14722/ndss.2021.24224.
24. Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, Wei Huo. SoFi: ReflectionAugmented Fuzzing for JavaScript Engines // In Pro-ceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), 2021, Virtual Event, Republic of Korea, ACM, New York, NY, USA, 14 pages,. DOI: 10.1145/3460120.3484823.
25. Porkolab Z. Fuzzing C++ class interfaces for generating and running tests with libFuzzer // IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2020, pp. I-Iii, DOI: 10.1109/ISSREW51248.2020.00027.
26. Hsu Chin-Chia, Wu Che-Yu, Hsiao Hsu-Chun, Huang Shih-Kun. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing, 2018, DOI: 10.14722/bar.2018.23014.
27. Hu Jr Z. A Software Package for Generating Code Coverage Reports With Gcov, 2021.
28. Beyer D., Lemberger T. TestCov: Robust test-suite execution and coverage measurement // 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1074-1077, DOI: 10.1109/ASE.2019.00105.
29. Васильев И.А., Довгалюк П.М., Климушенкова М.А. Использование идентификации потоков выполнения при решении задач полносистемного анализа бинарного кода // Труды Института системного программирования РАН, 2021, 33(6), С. 51-66, https://doi.org/10.15514/ISPRAS-2021-33(6)-4.
30. Климушенкова М.А., Бакулин М.Г., Падарян В.А., Довгалюк П.М., Фурсова Н.И., Васильев И.А. О некоторых ограничениях полносистемного анализа помеченных данных // Труды Института системного программирования РАН, 2016, 28(6), 11-26, https://doi.org/10.15514/ISPRAS-2016-28(6)-1.

APPROACHES OF ATTACK SURFACE ESTIMATION AND WEB BROWSER FUZZING

Kozachok A.V.[6], Nikolaev D. A.[7], Erokhina N.S.[8]

Abstract

Purpose of the work *is to develop of an approach to determining the attack surface based on the analysis of the difference in code coverage and its application to the analysis of web browsers.*

Research method *is to use an instrumentation compiler to analyze code coverage depending on the input data. The proposed approach makes it possible to evaluate the relationship between the input data processed by the analyzed application and the program code by calculating the coverage difference and excluding from the analysis the modules called regardless of the input data.*

Results of the research: *the existing general approaches to software fuzzing, and the features of approaches to fuzzing web browsers, are considered. In general, software fuzzing is performed using one of three methods: black-box, gray-box, and white-box. The basic criterion for distinguishing these methods is the completeness of information about the source code of the software under test. Web browser fuzzing can be divided into static and dynamic. Approaches to fuzzing complex software can be divided into two groups: analysis of a monolithic application, fuzzing of individual application modules (library interfaces). The difference between these groups is determined by the completeness of the involvement of the functional components of the software under study in the testing process. Each of the levels has its own advantages and disadvantages. Often these shortcomings can be compensated by a combination of fuzzing different fuzzing targets. To correctly determine fuzzing targets, it is necessary to identify the attack surface of the software under study. The authors proposed an approach to assessing the attack surface by calculating the coverage difference; it allows excluding from the analysis modules that are called regardless of the input data.*

Scientific and practical significance: *the results of the article consist in the development of a new approach to determining the attack surface based on the analysis of the difference in coverage of the code of the analyzed application, depending on the data supplied to the input, and allowing to exclude modules from the analysis that are called regardless of the input data.*

Keywords: *web browser, JavaScript engine, code coverage, software defects, software vulnerabilities, fuzzing testing.*

References

1. Kozachok, A. V., Kozachok, V. I., Osipova, N. S., Ponomarev, D. V. A review of studies on the use of machine learning methods to improve the efficiency of fuzzing testing. Vestnik VGU. Series: System Analysis and Information Technologies, 2021 (4), pp. 83-106. DOI: 10.17308/sait.2021.4/3800.
2. Li J., Li J., Zhao B., Zhang C. Fuzzing: a survey // Cybersecurity, 2018, Vol. 1, No 1, p. 6, DOI: 10.1186/s42400-018-0002-y.
3. Gopinath R., Görz P., Groce A. Mutation Analysis: Answering the Fuzzing Challenge // arXiv preprint arXiv:2201.11303, 2022, DOI: /10.48550/arXiv.2201.11303.
4. Bounimova E., Godefroid P., Molnar D. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production // In Proceedings of the International Conference on Software Engineering, pp. 122–131, 2013, DOI: 10.1109/ICSE.2013.6606558.
5. Godefroid, Michael Y. Levin, David Molnar. SAGE: Whitebox Fuzzing for Security Testing // Communications of the acm, 2012 (3), pp. 40-44 doi:10.1145/2093548.2093564.
6. Debnath Mukta, Basak Chowdhury Animesh, Saha Debasri, Sur Kolay Susmita. FuCE: Fuzzing+Concolic Execution guided Trojan Detection in Synthesizable Hardware Designs, 2021, DOI: 10.48550/arXiv.2111.00805.
7. Muhammad Noman Khalid, Muhammad iqbal, Kamran Rasheed, Malik Muneeb Abid. Web Vulnerability Finder (WVF): Automated Black-Box Web Vulnerability Scanner // I.J. Information Technology and Computer Science, 2020 (4), pp. 38-46, DOI: 10.5815/ijitcs.2020.04.05.

6 Kozachok Alexander V., doctor of technical science, Associate Professor, Academy of the Federal Guard Service of the Russian Federation, Orel, Russia, E-mail: a.kozachok@academ.msk.rsnet.ru, https://orcid.org/0000-0002-6501-2008

7 Nikolaev Dmitry A., Academy of the Federal Guard Service of the Russian Federation, Orel, Russia, E-mail: mriddi@bk.ru, https://orcid.org/0000-0001-9334-6948

8 Erokhina Natalya S., Academy of the Federal Guard Service of the Russian Federation, Orel, Russia, E-mail: osipova_nc@mail.ru, https://orcid.org/0000-0002-4878-0865

8. Danyang Zhao. Fuzzing Technique in Web Applications and Beyond // Journal of Physics: Conference Series 1678(1), 2020, doi: 10.1088/1742-6596/1678/1/012109.
9. Zimin E.E. Code fuzzing testing technique using AFL // Secure Information Technologies. Proceedings of the Eleventh International Scientific and Technical Conference. 2021 Publisher: MSTU named after N.E. Bauman, 2021, pp. 124-129.
10. Tronov K. A., Belov Yu.S. Optimization of the AFL toolkit for better code coverage when working with specific data // E-Scio. FGBOU VO "MSTU named after N.E. Bauman, 2021, No. 5 (56), pp. 566-571.
11. Pham V. T. et al. Smart greybox fuzzing // IEEE Transactions on Software Engineering, 2019, T. 47, №. 9, pp. 1980-1997, DOI: 10.1109/TSE.2019.2941681.
12. Ying-Dar Lin, Feng-Ze Liao, Shih-Kun Huang, Yuan-Cheng Lai. Browser Fuzzing by Scheduled Mutation and Generation of Document Object Models // The 49th IEEE International Carnahan Conference on Security Technology, 2020, DOI: 10.1109/CCST.2015.7389677.
13. Sablotny M., Jensen B. S., Johnson C. W. Recurrent neural networks for fuzz testing web browsers // International Conference on Information Security and Cryptology, Springer, Cham, 2018, pp. 354-370, DOI: 10.1007/978-3-030-12146-4_22.
14. Eberlein M. et al. Evolutionary grammar-based fuzzing // International Symposium on Search Based Software Engineering, Springer, Cham, 2020, pp. 105-120, DOI: 10.1007/978-3-030-59762-7_8.
15. Manes V. J. M., Han H., Cha S. K. [et al.] The Art, Science, and Engineering of Fuzzing: A Survey // IEEE Transactions on Software Engineering, 2021, Vol. 47, No 11, pp. 2312-2331, DOI: 10.1109/TSE.2019.2946563.
16. Kostandyan V., Margarov G., Pedrosa T., Juan Rodriguez P.. Fuzzing in cybersecurity (review) // New in the Russian electric power industry, 2020, no. 9, pp. 17-30.
17. Xu W., Park S., Kim T. FREEDOM: Engineering a State-ofthe-Art DOM Fuzzer // Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event USA: ACM, 2020, pp. 971–986. DOI: 10.1145/3372297.3423340.
18. Chaofan Shou, ˙Ismet Burak Kadron, Qi Su, and Tevfik Bultan. CorbFuzz: Checking Browser Security Policies with Fuzzing // University of California, Santa Barbara, 2021, DOI: 10.48550/arXiv.2109.00398.
19. Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing // In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), 2021, Virtual, Canada. ACM, New York, NY, USA, DOI: 10.1145/3453483.3454054.
20. Agrawal V., Rastogi R., Tiwari D. C. Spider monkey optimization: a survey // International Journal of System Assurance Engineering and Management, 2018, T. 9, №. 4, pp. 929-941, DOI: 10.1007/s13198-017-0685-6.
21. Tian Y., Qin X., Gan S. Research on Fuzzing Technology for JavaScript Engines // The 5th International Conference on Computer Science and Application Engineering, 2021, pp. 1-7, DOI: 10.1145/3487075.3487107.
22. Groß S. Fuzzil: Coverage guided fuzzing for javascript engines // Department of Informatics, Karlsruhe Institute of Technology, 2018.
23. Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos , Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupe, and Yan Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases // Network and Distributed Systems Security (NDSS) Symposium, 2021, DOI: 10.14722/ndss.2021.24224.
24. Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, Wei Huo. SoFi: ReflectionAugmented Fuzzing for JavaScript Engines // In Pro-ceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), 2021, Virtual Event, Republic of Korea, ACM, New York, NY, USA, 14 pages,. DOI: 10.1145/3460120.3484823.
25. Porkolab Z. Fuzzing C++ class interfaces for generating and running tests with libFuzzer // IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2020, pp. I-Iii, DOI: 10.1109/ISSREW51248.2020.00027.
26. Hsu Chin-Chia, Wu Che-Yu, Hsiao Hsu-Chun, Huang Shih-Kun. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing, 2018, DOI: 10.14722/bar.2018.23014.
27. Hu Jr Z. A Software Package for Generating Code Coverage Reports With Gcov, 2021.
28. Beyer D., Lemberger T. TestCov: Robust test-suite execution and coverage measurement // 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1074-1077, DOI: 10.1109/ASE.2019.00105.
29. Vasiliev I.A., Dovgalyuk P.M., Klimushenkova M.A. Using the identification of threads of execution when solving problems of full-system analysis of binary code // Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), 2021, 33(6), pp. 51-66, (In Russ.), DOI: 10.15514/ISPRAS-2021-33(6)-4.
30. Klimushenkova M.A., Bakulin M.G., Padaryan V.A., Dovgalyuk P.M., Fursova N.I., Vasiliev I.A. On Some Limitations of Information Flow Tracking in Full-system Emulators // Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), 2016, 28(6), pp. 11-26, (In Russ.), DOI: 10.15514/ISPRAS-2016-28(6)-1.