Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Факультет математики, механики и компьютерных технологий
Кафедра прикладной математики и программирования

РАБОТА ПРОВЕРЕНА
Рецензент, _____
_____
_____/_____
«____»_____2018г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д.ф.-м.н.,
профессор
_____/А.А. Замышляева
«____»_____2018 г.

Защита чувствительной информации при распределенном выполнении кода на JavaScript

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ–090404.2018.194.ПЗ ВКР

Руководитель работы, к.ф.-м.н.,
Доцент кафедры ПМиП
_____/ С.М. Елсаков
«____»_____2018 г.
Автор работы
Студент группы ЕТ-223
_____/ Е.С. Кречетова
«____»_____2018 г.
Нормоконтролер, к.э.н., доцент кафедры ПМиП
_____/ Д.А. Дрозин
«____»_____2018 г.

Челябинск 2018

АННОТАЦИЯ

Кречетова Е.С. Защита чувствительной информации при распределенном выполнении кода на JavaScript.– Челябинск: ЮУрГУ, ЕТ-223, 56 с., 11 ил., библиогр. список – 29 наим., 4 прил.

В работе исследована предметная область в отношении защиты чувствительной информации. Рассмотрены существующие технические решения защиты чувствительных данных. Изучены модели защищенных систем и на их основе разработана модель защиты системы. В ходе работы было предложено решение задачи автоматизированного поиска возможных недоверенных источников данных и критических каналов вывода, создан модуль разметки кода пользователя. Проанализировано реальное приложение на Electron, и сделан вывод о найденных уязвимостях. Программная реализация осуществлена на языке JavaScript с использованием программной платформы Node.JS. В приложениях приведена программная документация на разработанный программный продукт.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

JavaScript — это язык программирования, который дает возможность реализовывать сложное поведение веб-страницы. Веб-страница не только отображает статическое содержимое, но и своевременно отображает обновление контента, выводит интерактивные карты, 2D/3D анимацию, прокручивает видео и т.д. Преимущества этого языка подтверждает высокий процент использования компаниями для разработки не только браузерных веб-приложений, в настоящее время он используется для создания десктопных приложений под различные платформы – Windows, Mac и т.п. Данная возможность используется благодаря созданию таких сред выполнения кода на JavaScript как Node.js.

Node.js представляет среду выполнения кода на JavaScript, которая построена на основе движка JavaScript Chrome V8, который позволяет транслировать вызовы на языке JavaScript в машинный код. Node.js прежде всего предназначен для создания серверных приложений на языке JavaScript. Но в рамках данной работы интересует возможность кроссплатформенной разработки десктопных приложений. Самые распространенные приложения, написанные с помощью Node.js – это мессенджер Slack, а также такие популярные редакторы кода, как GitHub Atom и Visual Studio Code. Для создания десктопных приложений на базе Node.js и веб-технологий доступны два варианта: NW.js (ранее известный, как node-webkit) и Electron (ранее – Atom Shell)[1]. Оба этих фреймворка являются проектами с открытым исходным кодом и поддерживаются такими крупными компаниями, как Intel и GitHub. Принципиальным различием между ними является парадигма выполнения приложения. В NW.js точкой входа в приложение является веб-страница. Указывается URL основной страницы в файле package.json и она открывается как главное окно приложения. В Electron точка входа — это программа на NodeJS. NW.js загружает указанную HTML-страницу и эта страница получает доступ к контексту Node.js. Поэтому большинство приходит к мнению, что NW.js использует более браузеро-ориентированный подход. Electron, с другой стороны, имеет более NodeJS-ориентированный подход. Он запускает среду выполнения Node.js, которая получает возможность открывать окна, в которые можно затем загружать веб-страницы. В остальном, различия существенны только по отношению к разрабатываемому ПО.

В рамках данной работы концепция этого фреймворка интересует как некое основание для создания приложений с распределенным выполнением кода.

В работе предлагается решение задачи автоматизированного поиска возможных недоверенного источника данных, всевозможных точек модификации чувствительной пользовательской информации, а также создание модуля разметки кода пользователя. Недоверенные источники данных – это, в первую очередь узкое место, поскольку злоумышленники могут реализовать такой достаточно распространенный тип уязвимости как Cross Site Scripting (далее – XSS) [2;29], который дает возможность передать и выполнить сторонний JavaScript-код в среде выполнения кода от имени пользователя. Специфика такого

типа уязвимостей заключается в возможности реализации ошибки в коде на клиентской стороне приложения, интерактивного интерфейса взаимодействия с пользователем.

Влияние на контекст выполнения JS-кода может произойти из-за небезопасного использования кода страницы, выполняющей визуализацию интерфейса взаимодействия с программой пользователей со своими методами и свойствами. К таким методам относятся вызовы, предназначенные для модификации содержимого страницы в интерактивном режиме (например, document.write) или для выполнения JS-кода на лету (например, eval).

Целью данной работы является разработка автоматизированной разметки исходного кода пользователя согласно клиент-серверной архитектуре на платформе Electron.

ГЛАВА 1. СРЕДСТВА И МОДЕЛИ ЗАЩИТЫ ЧУВСТВИТЕЛЬНЫХ ДАННЫХ

1.1 Средства защиты чувствительных данных

В настоящее время существует огромное множество статических и динамических анализаторов исходного кода, целью которых является выявление интересующих свойств веб-приложения. Например, время выполнения, наиболее часто исполняемые блоки (SQL-запросы, системные вызовы) программы, присутствие ошибок работы с памятью, а также наличие уязвимостей в программе. Статические анализаторы исследуют программу без ее выполнения, а динамические – во время ее выполнения.

В классификацию уязвимостей защиты включают множество программных ошибок. В данной работе интересует только контекст поиска уязвимостей веб-приложений, возникающих в случаях, когда пользователем вводятся некорректно обработанные данные. При нарушении правильной работы программы вследствие ввода некорректных данных, появляется возможность возникновения уязвимостей, которыми можно воспользоваться для обхода ограничений защиты всей системы. В этом случае пользователь может таким образом предоставить данные, что программа выполнит совсем не ту команду, которая предполагалась разработчиками.

В 1975 году ученым Дороти Дэннинг было разработано фундаментальное определение уязвимости, которое потом было развито в моделях невыводимости и невмешательства. Данную работу интересуют модель невмешательства, которая вкратце определяет уязвимость так:

«Пусть программа получает данные из high-level (доверенных) и low-level (недоверенных) каналов и генерирует вывод в high-level (критические) и low-level (обычные) каналы. Вывод low-level (недоверенных) данных в high-level (критический) канал считается нарушением принципа невмешательства.»

Для дальнейшего следования принципам данного определения существует путь, именуемый taint-анализом. В нем недоверенными каналами ввода являются методы, возвращающие данные HTTP-запросов, а критическими каналами ввода – вызовы критических функций.

Таким образом, будет уместно рассмотреть анализаторы кода и средства языков, которые исследуют веб-приложения в данном контексте.

1.1.1 Языки Ruby и Perl

Существуют различные решения для отслеживания состояния безопасности объектов в программе. Такие языки, как Ruby и Perl, осуществляют интересующую нас taint-проверку, устанавливая уровень доверия вводимым данным, которые могут повлиять на объекты. Taint-проверка, или проверка на «грязные» данные, выполняется путем пометок объектов, которые становятся ненадежными, или недоверенными, и получают метку taint [15]. Этот статус может быть передан другому объекту, на который он влияет. Чтобы использовать

его в незащищенной среде исполнения, сначала нужно проанализировать, или отфильтровать, ненадежный объект, чтобы убедиться, что он не представляет угрозы, а затем убрать с него taint-метку.

В Ruby данная проверка связана с устанавливаемым safe-режимом, в котором запущена программа. В языке есть 4 таких safe-режимов, в зависимости от которых меняется возможность использования данных из недоверенных источников [2]:

1) safe=0 – Выполняется проверка используемых внешних (испорченных) данных. Режим по умолчанию;
2) safe>=1 – Ruby запрещает использование ненадежных данных в потенциально опасных операциях. Запрещено манипулировать каталогами, имя которых взято из ненадежных данных. Запрещено чтение ненадежных данных. Запрещена загрузка файлов, имя которых получено с помощью ненадежных данных. Нельзя манипулировать и запрашивать статус файлов, имя которых получено помощью ненадежных данных. Запрещено выполнение системных команд и программ, вызываемых с помощью не доверенных введенных данных. Нельзя пропускать данные после обработки исключений;
3) safe>=2 – Запрещена загрузка файлов программ из общедоступных мест. Запрещено изменение, создание и удаления каталогов. Запрещена загрузка файлов с именем из не доверенного источника, начинающееся с ~. Нельзя использовать следующее множество методов: File#chmod , File#chown , File#lstat , File.stat , File#truncate , File.umask , File#flock , IO#ioctl , IO#stat , Kernel#fork , Kernel#syscall , Kernel#trap . Process::setpgid , Process::setsid , Process::setpriority , or Process::egid=. Не возможна обработка сигналов с помощью исключений;
4) safe>=3 – Все новые созданные объекты считаются ненадежными. Запрещено снятие метки taint с помощью метода untaint;
5) safe>=4 – Ненадежные данные не могут быть изменены. Запрещено изменение глобальных переменных. Запрещен доступ к переменным чистых объектов. Запрещено изменение переменных среды. Нельзя закрывать или заново открывать чистые файлы. Запрещено изменение видимости методов. Запрещено создание псевдонимов чистых классов или модулей. Запрещено получение метаинформации (такой как списки методов или переменных). Запрещено определение, переопределение и удаление методов в чистых классах или модулях. Запрещено изменение объектов и удаление экземпляров и констант методов. Запрещено управление потоками, их прекращение, создание переменных внутри них. Запрещены вызовы exit, exit! и abort. Невозможно преобразовать идентификаторы символов в ссылки на объекты. Нельзя использовать автозагрузку.

Текущее значение режима наследуется при создании новых потоков. Однако в каждом потоке это значение может меняться. Это средство может использоваться для реализации безопасных «песочниц», областей, где внешний код может работать без риска для остальных частей приложения или системы.

Любой объект Ruby, полученный из некоторого внешнего источника (например, введенная строка) автоматически помечается как ненадежный. Если

программа использует ненадежный объект для получения нового, этот новый также будет недоверен. Этот процесс выполняется независимо от текущего уровня безопасности.

На любом объекте можно поставить метку, вызвав метод taint. Если уровень безопасности меньше третьего, то с объекта можно удалить метку, вызвав метод untaint.

Цель taint-режима в языке Perl аналогична – не допустить влияния каких-либо внешних данных на приложение путем запрета на ввод пользователей значений в eval, которые передаются и используются в какой-то команде программы.

Когда используется taint-режим, Perl контролирует каждую переменную на предмет ненадежных данных. Ненадежные данные, согласно этому языку, также представляют собой данные, поступающие извне исходного кода. И он также не только отслеживает, откуда прибывают данные, а еще и следует за переменной, которой они присвоились, если вдруг она влияет на другие переменные.

Отличием от языка Ruby является отсутствие уровней в данном режиме – рассматриваются любые действия, которые могут повлиять на ресурсы вне кода, подлежащие принудительному исполнению. Например, манипуляции с файлом, имеющим какое-то отношение к ненадежным данным, производятся исключительно в режиме чтения. И если попытаться открыть файл для записи данных, программа прекратит работу и выдаст ошибку.

Perl автоматически включает набор специальных проверок безопасности, называемых taint mode, когда обнаруживает, что программа работает в небезопасной среде. Но также можно явно задать данный режим, используя флаг –T [3]. Этот флаг настоятельно рекомендуется разработчиками для программ, выполняющихся на сервере или так называемых CGI-скриптов. В этом режиме принимаются специальные меры предосторожности, которые называются taint-проверками.

В данном режиме запрещено использовать данные, полученные извне программы и способные повлиять на что-то еще, по крайней мере, не случайно. Все аргументы командной строки, переменные среды, результаты определенных системных вызовов (readdir(), readlink(), переменная shmread(), сообщения, возвращаемые функцией msgrcv(), поля password, gcos и shell, возвращаемые вызовом getpwxxx()) и все входные данные помечены ненадежными. Такие данные не могут использоваться прямо или косвенно в любой команде, которая модифицирует файлы, каталоги или процессы со следующими исключениями:

- Параметры print и syswrite не проверяются.

- Символьные методы $obj->$method(@args); и символьные вспомогательные ссылки &{$foo}(@args); $foo->(@args); не проверяются. Причем тут следует очень тщательно проверять влияние внешних данных на поток управления. Если не проверять и не ограничивать использование этих символьных значений, возможен непредвиденный программистом вызов функций из POSIX::system, что позволит запускать произвольный внешний код.

- Хеш-ключи никогда не помечаются меткой taint.

По соображениям эффективности Perl придерживается консервативного представления о том, ненадежны ли данные. Если выражение содержит ненадежные данные, любое подвыражение может считаться ненадежным, даже если значение подвыражения не зависит напрямую от ненадежных данных.

Исключением из принципа «из-за одного ненадежного значения ненадежным становится все выражение» являются условный оператор и тернарный условный оператор.

Чтобы проверить, содержит ли переменная скрытые данные, использование которых вызовет сообщение «Insecure dependency» («Незащищенная зависимость»), нужно использовать функции tainted() модуля Scalar::Util или is_tainted(). Данная функция как раз следует принципу «из-за одного ненадежного значения ненадежным становится все выражение», из-за неэффективности проверки каждого оператора.

Но такая проверка может зайти слишком далеко. Иногда возможно полностью довериться какому-либо оператору и снять с него taint-метку, но не напрямую. Например, значения могут быть не затронуты, если использовать их как ключи в хэше. В противном случае единственным способом обойти tainting-механизм является ссылка на подшаблон из регулярного выражения. Perl предполагает, что если вы ссылаетесь на подстроку, используя специальные переменные $1, $2 и т.п. в шаблоне, не считающимся ненадежными, вы знаете, что делаете, когда пишете этот шаблон. Но при этом рекомендуется проверить, содержит ли переменная только разрешенные символы. В документации языка описан такой пример проверки данных на содержание чего-то, кроме определенных символов (букв, цифр и подчеркиваний):

«if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;   # данные в $data теперь не считаются ненадежными
} else {
    die "Bad data in '$data'"; #в противном случае, считаем их ненадежными
}» [4]

Такая проверка считается безопасной, т.к. /\w+/ обычно не соответствует метасимволам командной строки, а также чему-либо еще причиняющему вред среде.

В языке Perl снятие taint-метки данных, введенных пользователем, с помощью регулярных выражений является единственным путем [4].

На рынке программного обеспечения также существуют пакеты программ, так называемых статических и динамических анализаторов исходного кода. Разработчики утверждают, что с помощью них можно находить недостатки, снижающие защищенность программного продукта.

Входными данными для анализатора исходного кода является массив исходных текстов программ и его зависимостей (подгружаемых модулей, используемого стороннего программного обеспечения и т. д.). В качестве результатов работы все анализаторы выдают отчет об обнаруженных уязвимостях и ошибках программирования, дополнительно некоторые анализаторы предоставляют функции по автоматическому исправлению ошибок.

1.1.2 Интерпретаторы защиты чувствительных данных

Существует большое количество коммерческих и бесплатных анализаторов кода. Список языков, для которых существуют статические анализаторы кода, также достаточно велик. Рассмотрим наиболее популярные анализаторы кода на предмет, интересующий в контексте темы работы, taint-анализа.

HP Fortify Static Code Analyzer – продукт компании Fortify, с 2010 года принадлежащей Hewlett-Packard. В линейке HP Fortify присутствуют различные продукты для анализа программных кодов: SaaS-сервис Fortify On-Demand, предполагающий загрузку исходного кода в облако HP, и полноценное приложение HP Fortify Static Code Analyzer, устанавливаемое в инфраструктуре заказчика. HP Fortify Static Code Analyzer поддерживает большое число языков программирования и платформ, включая веб-приложения, написанные на PHP, Python, Java/JSP, ASP.Net и JavaScript, и встраиваемый код на языках ABAP (SAP), Action Script и VBScript[5].

В документации к пакету программ описан модуль, называемый Анализатором потока данных анализатора исходного кода (SCA Dataflow Analyzer) [5]. Данный модуль позволяет обнаружить проблемы безопасности, которые связаны с ненадежными данными, вводимыми в одной точке и распространяющимися на другие точки программы. Например, SQL-инъекции, в которые попадают Ненадежные данные, полученные из такого источника, как HTTP-запрос, и которые в дальнейшем используются для формирования SQL-запроса. В этом случае, модуль сообщает об этой уязвимости из-за SQL-инъекции в отчете. Анализатор потока данных, выполняя межпроцедурный анализ, способен отслеживать ненадежные данные и через глобальные переменные в программе.

Также данный модуль работает с моделью программы. Модель создается анализатором из исходного кода программы и «правил». Исходный код программы обеспечивает базовый уровень модели. Этот уровень описывает поведение методов, отношения между различными методами и взаимосвязь между методами и глобальными переменными. Затем анализатор дополняет модель «правилами». Эти правила описывают точки в программе, которые действуют как источники ненадежных данных и поглотители. Они также описывают программные точки, которые манипулируют ненадежными данными и передают их.

В документации был описан такой пример простой программы, которая иллюстрирует уязвимость по отношению к вводу команд:

«function run()
{
    readFromNetwork(buffer);
    command = concatenate("/usr/bin" buffer);
    execute(command);
}» [5]

Вызов функции readFromNetwork() обеспечивает ввод ненадежных данных. Далее введенная строка объединяется со строкой для формирования команды и

передается функции execute(), которая выполняет новый процесс, указанный в командной строке.

Построив модель из исходного кода, анализатор потока данных может понять, что из run() вызывается три внешних функции, и что между этими вызовами через локальные переменные существует связь потока данных. Поскольку данный исходный код не является частью программы, модель не дополняется набором правил, описывающих характеристики этих функций. И без каких-либо знаний о внешних функциях анализатор не понимает, откуда поступают ненадежные данные и как перемещаются по программе.

В этом случае анализатор потока данных может обнаружить уязвимость со следующими правилами:

– Правило ненадежного источника для readFromNetwork().

– Правило ненадежного перехода для concatenate().

– Правило ненадежного потребителя для execute().

Анализатор потока данных включает может определить для функции следующий набор правил:

– Ненадежный источник.

– Ненадежная точка входа.

– Ненадежный потребитель.

– Ненадежный проход.

– Ненадежный флаг поведения.

– Функции проверки.

Ненадежные данные поступают в программу через программную точку, называемую ненадежным источником. Примерами таких источников являются функции, которые считывают данные из сетевых источников, таких как HTTP-запросы, и функции, которые считывают данные из ненадежных источников (чужая база данных).

Ненадежная точка входа является специальным типом источника, описывающим функцию, которая вызывается с ненадежными данными. Например, функция, вызываемая с аргументами из командной строки, или функция в структуре веб-приложения, вызываемая непосредственно фреймворком с ненадежными входными параметрами.

Ненадежные потребители – это программные точки, к которым не должны попадать ненадежные данные. Когда анализатор потока данных обнаруживает путь, через который могут поступать ненадежные данные, он сообщает о проблеме. Примером таких функций являются функции, которые выполняют SQL-запросы или выполняют другие команды, описанные данной строкой. Правило ненадежных потребителей может содержать условное выражение, которое ограничивает попадание под это правило, путем проверки флагов.

Внешне для функций, определенных в исходном коде, с помощью правила ненадежного прохода генерируется поведение сквозного перехода данных между функциями. Например, стандартный пакет правил HP Fortify Secure Coding Rulepacks содержит правило, описывающее поведение перехода данных для

функции StringBuilder.append(). Правило прохода может добавлять или удалять флаг taint из ненадежных данных [6].

Чистка метки taint (Taint Cleance) – это точка, в которой удаляется или изменяется метка taint. Обычно этой точкой является функция проверки. Документация описывает два типа чистки меток: полная чистка – правило, по которому метка taint не применяется или вовсе удаляется и прекращается слежение за распространением ненадежных данных, и частичная чистка – правило, которое указывает на необходимость установки или удаления метки. Во втором случае, слежение за распространением ненадежных данных не останавливается, а только меняется набор флагов. Правила чистки применяются в любом месте программы. Если вызов функции соответствует правилу очистки, то она применяется во всех путях выполнения программы, затрагивающих данную функцию.

Флаг taint является атрибутом ненадежных данных, который позволяет анализатору определять различные типы меток taint. Флаги помогают определить, безопасно ли использование данных в определенном контексте. Каждому пути выполнения программы соответствует свой набор флагов. Анализатор потока данных может добавлять или удалять флаги, которые были наследованы из ненадежного источника, поскольку метка совершает сквозной переход через функции (taint pass-through) и очищающие точки (Validation Functions) в программе. Ненадежный потребитель (taint sink) может проверять наличие флагов taint, которые определяют, будет ли анализатор кода сообщать о конкретном пути от источника к потребителю.

Одной из основных задач для обеспечения правильной работы анализатора является запись правил для функций проверки. Пользователю предоставляется возможность дополнить правила прохода и очистки. В большинстве случаев предпочтительно добавить определенный флаг taint в путь, по которому могут проходить определенные ненадежные данные.

Еще один продукт – анализатор исходных кодов IBM Security AppScan Source. Линейка AppScan включает множество продуктов, связанных с безопасной разработкой программного обеспечения и в первую очередь предназначен для организаций-разработчиков, при этом поддерживает меньшее число языков веб-разработки – только PHP, Perl и JavaScript.

Поиск и отметка ненадежных данных с помощью этого пакета программ происходит в несколько шагов:

1. Обзор обнаруженных источников.
2. Определение известных, но отсутствующих в предыдущем списке источников.
3. Определение потерянных потребителей.
4. Определение методов, использующих и не использующих ненадежные данные.
5. Определение распространителей ненадежных данных.
6. Идентификация других потерянных источников.
7. Повторное выполнение сканирования.

Для просмотра информации, откуда поступают данные на высоком уровне, пользователю предлагается открыть модуль Источники и Потребители. Модуль позволяет проверить типы источников данных, чтобы выяснить, нет ли среди них не предполагаемых. Например, если проверяется веб-приложение, можно увидеть источники Интернета и базы данных. Возможно, база данных разработчика создана только для записи, но это все равно предлагается проверить. На рисунке 1 показан пример представления таких источников.

Рисунок 1.1 – Просмотр источников и поглотителей, в том числе из сети и баз данных

Следующим шагом разработчик предлагает проанализировать известные, но отсутствующие источники данных (те, что приложение пользователя якобы не использует). Здесь можно получить информацию о приеме данных, источники которых не отображаются в модуле Источники и Потребители, и о том, как они работают. Если такие методы имеются, предлагается отметить их как источники и потребители ненадежных данных.

Источником здесь является метод, который возвращает ненадежные данные, а потребитель – метод, который принимает в свои параметры ненадежные данные.

Определение потерянных потребителей (шаг 3) – это поиск API, которые анализатор кода не может идентифицировать и не знает, что делает код, с которым он столкнулся. На данном шаге невозможно дальнейшее исследование переходов ненадежных данных, которые, скорее всего, в конце концов используются какими-то потребителями. Определение и устранение таких потерянных потребителей представляется полезным, потому что это действие может значительно увеличить покрытие анализа исходного кода. Анализатор определяет ключевые API-интерфейсы, к которым программа пользователя обращается наибольшее количество раз. Информация о них также хранится в модуле Источники и Потребители, как показано на рисунке 1.2.

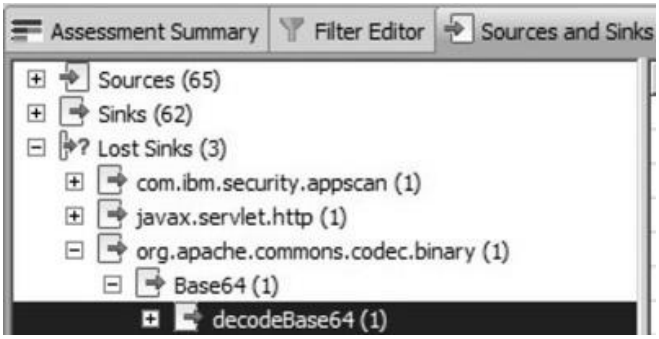

Рисунок 1.2 – Просмотр потерянных источников в модуле Источники и Потребители

Такие потерянные источники можно отметить «доверенными». Например, если это API, разработанный компанией-пользователем, нужно включить его в пакет исходных кодов для анализа. Если исходный код был доступен для сканирования, но все же отображается как потерянный потребитель (хотя разработчик анализатора утверждает, что это маловероятно), можно отметить этот потребитель как безопасный.

Следующим шагом анализа является определение методов, использующих и не использующих ненадежные данные.

Идентификация потребителей из всего списка, сформировавшегося в ходе анализа, производится пользователем программы. Если должны проводиться проверка, подтверждение или очистка данных после входа в определенный метод, то этот метод является потребителем. Например, если потерянный потребитель передает данные во внешнюю систему, стороннюю библиотеку, базу данных или пользователю, то необходимо проверять данные до того, как они покинут «диапазон контроля». Явными примерами типичных потребителей являются функции dbQuery.execute(...), netManager.send(...), httpResponse.write(...), thirdPartyLibrary.doSomething(...), backEndService.run(...) и т.п. На рисунке 1.3 показан пример потерянного потребителя – функции, которая на самом деле является потребителем ненадежных данных, logTransaction(). Она регистрирует информацию о транзакции (в том числе, конфиденциальные данные о кредитной карте) в лог-файл.

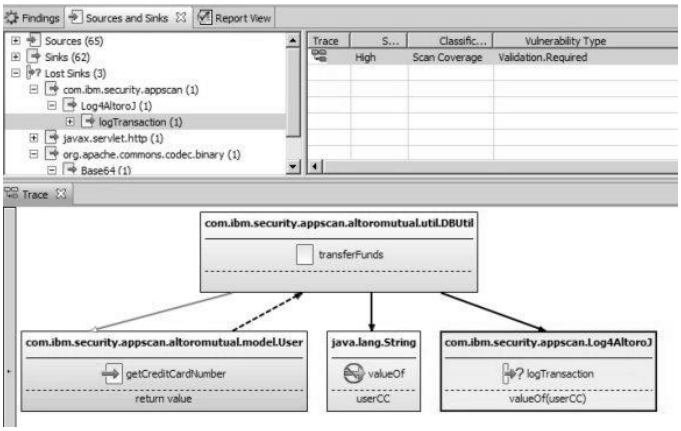

Рисунок 1.3 – Пример потерянного потребителя, который является потребителем ненадежных данных

Для определения методов, на которые не влияют никакие ненадежные данные, достаточно ответить на вопрос: «Существуют ли какие-либо сценарии выполнения программы, ведущие к данному потерянному потребителю?». Если ответ отрицательный, то данный метод не помечается как ненадежный потребитель, и сценарии, ведущие в вызову данного метода, считаются безопасными.

Анализатор предусматривает возможность определения распространителей ненадежных данных (шаг 5). Методы-распространители не «генерируют» ненадежные данные, т.е. поодиночке они не являются источниками. Однако если таким методам передаются небезопасные данные как параметры, возвращаемые ими значения также считаются небезопасными. Примерами таких методов являются string.subString(…), string.append(…) и base64.encode().

На рисунке 1.4 показан пример потерянного потребителя, который на самом деле является распространителем ненадежных или чувствительных данных. В этом примере метод decode.Base64() преобразует список учетных записей с кодировкой base64, хранящийся в файле cookie, в простой текстовый список, который может использоваться приложением.

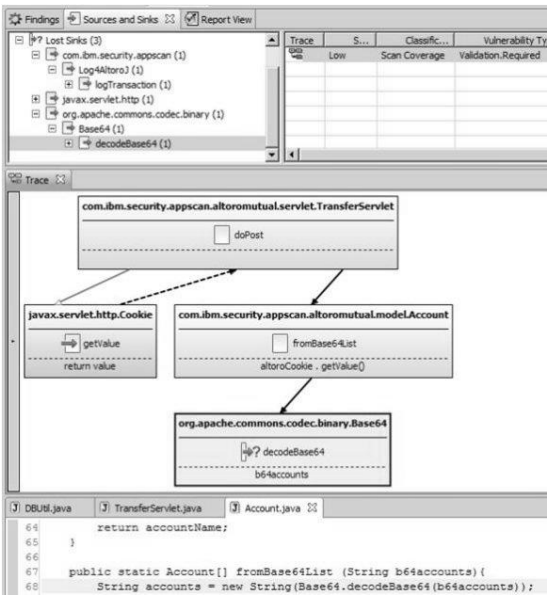

Рисунок 1.4 – Пример потерянного источника, который на самом деле является распространителем

Когда пользователь отмечает метод в качестве распространителя, анализатор считает все его входные параметры, а также возвращаемые значения и указатели ненадежными или опасными. Иными словами, теперь анализатор будет следить за данными метода, поступающими через любой параметр, и возвращаемыми им значениями.

Шагом 6 работы анализатора является обзор потерянных источников, не попавших ни под одно определение, но которые все равно нужно еще раз проверить. Их поиск и анализ может занять много времени и можно так и не обнаружить их влияние на обработку ненадежных или чувствительных данных.

На рисунке 1.5 показан пример обнаружения методов, за которыми не замечены предпосылки опасного поведения.

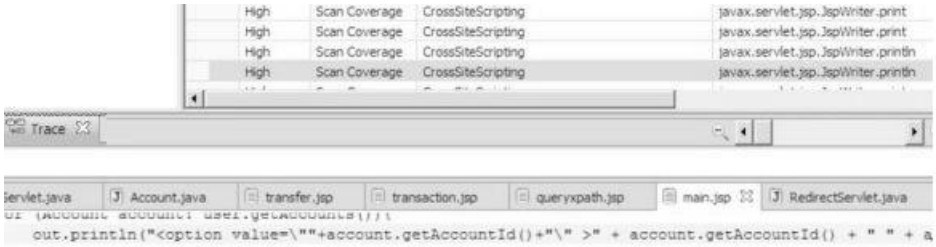

Рисунок 1.5 – Пример просмотра методов, не попадающих под влияние чувствительных или ненадежных данных

Последним шагом анализа исходного кода пользователя является повторное сканирование кода после просмотра обнаруженных уязвимостей и правок пользователя.

1.2 Модели защищенных систем

Фундаментальным понятием в сфере защиты информации компьютерных систем является политика безопасности. Под ней понимают совокупность норм и правил, регламентирующих процесс обработки информации, выполнение которых обеспечивает состояние защищенности информации в заданном пространстве угроз. Формальное выражение политики безопасности (математическое, схемотехническое, алгоритмическое и т.д.) называют моделью безопасности.

Модели безопасности играют важную роль в процессах разработки и исследования защищенных компьютерных систем, так как обеспечивают системотехнический подход, включающий решение следующих важнейших задач:

- выбор и обоснование базовых принципов архитектуры защищенных компьютерных систем, определяющих механизмы реализации средств и методов защиты информации;
- подтверждение свойств (защищенности) разрабатываемых систем путем формального доказательства соблюдения политики безопасности (требований , условий , критериев);
- составление формальной спецификации политики безопасности как важнейшей составной части организационного и документационного обеспечения разрабатываемых защищенных компьютерных систем [7].

В классификацию уязвимостей защиты включают множество программных ошибок. В данной работе интересует только контекст поиска уязвимостей веб-приложений, возникающих в случаях, когда пользователем вводятся некорректно обработанные данные. При нарушении правильной работы программы вследствие ввода некорректных данных, появляется возможность возникновения уязвимостей, которыми можно воспользоваться для обхода ограничений защиты всей системы. В этом случае пользователь может таким образом предоставить

данные, что программа выполнит совсем не ту команду, которая предполагалась разработчиками. В этой работе рассмотрим модели, защищающие от угрозы раскрытия информации, а также математические основы моделей контроля целостности информации. К данному контексту больше всего подходят модели из классификаций – модели безопасности на основе мандатной политики и информационные модели.

Рассматриваемые модели определяют ограничения на отношение ввода/вывода системы, которые достаточны для реализации системы. Данные модели накладывают ограничения на интерфейс программных модулей системы с целью достижения безопасной реализации. При этом подробности реализации определяются разработчиком системы. Данные модели являются результатом применения теории информации к проблеме безопасности систем. Классической моделью, лежащей в основе построения многих систем по принципам модели мандатного доступа, является модель Белла и Лападула. К информационным моделям относятся модели невмешательства и невыводимости и модель системы безопасности с полным перекрытием [8].

1.1.1 Модель Белла-ЛаПадулы

Модель Белла-Лападулы – модель контроля и управления доступом, основанная на мандатной модели управления доступом. В модели анализируются условия, при которых невозможно создание информационных потоков от субъектов с более высоким уровнем доступа к субъектам с более низким уровнем доступа.

Данная модель, разработанная еще в 1972–1975 г.г. американскими специалистами – сотрудниками MITRE Corporation Дэвидом Беллом и Леонардом ЛаПадулой (D.Elliott Bell, Leonard J.LaPadula), была названа по их имени и сыграла огромную методологическую роль в развитии теории компьютерной безопасности.

Основные положения модели Белла-ЛаПадулы сводятся к следующему.

Модель системы $\Sigma(v_0, Q, F_T)$ представляется совокупностью:

множества объектов O доступа;

множества субъектов S доступа;

множества прав доступа R (в т. н. "классической" модели Белла-ЛаПадулы) всего два элемента – read и write);

матрицы доступа A[s,o];

решетки Λ_L уровней безопасности L субъектов и объектов системы;

функции F_L: $S \cup O \rightarrow L$, отображающей элементы множеств S и O на множество L;

множества состояний системы V, которое определяется множеством

упорядоченных пар (F_L, A);

начального состояния $v_0 \in V$;

набора запросов Q субъектов на доступ (осуществление операций) к объектам, выполнение которых переводит систему в новое состояние;

функции переходов $F_T : (V \times Q) \rightarrow V^*$, которая переводит систему из одного состояния V в другое V* при выполнении запросов из Q.

Состояния системы разделяются на опасные и безопасные. Для анализа и формулировки условий, обеспечивающих безопасность состояний системы, вводятся следующие определения:

Определение 2.1. Состояние называется безопасным по чтению (или просто безопасным) тогда и только тогда, когда для каждого субъекта, осуществляющего в этом состоянии доступ чтения к объекту, уровень безопасности этого субъекта доминирует над уровнем безопасности этого объекта:

$\forall s \in S, \forall o \in O, read \in A[s,o] \rightarrow F_L(s) \geq F_L(o)$.

Определение 2.2. Состояние называется безопасным по записи (или безопасным[1]) тогда и только тогда, когда для каждого субъекта, осуществляющего в этом состоянии доступ записи к объекту, уровень безопасности этого объекта доминирует над уровнем безопасности этого субъекта:

Определение 2.3. Состояние системы безопасно тогда и только тогда, когда оно безопасно и по чтению, и по записи.

На основе введенных понятий, авторы модели сформулировали следующий критерий безопасности.

Определение 2.4.(Критерий безопасности в модели Белла-ЛаПадулы). Система $\Sigma(v_0, Q, F_T)$ безопасна тогда и только тогда, когда ее начальное состояние v_0 безопасно и все состояния, достижимые из v_0 путем применения конечной последовательности запросов из Q, безопасны.

На основе данного критерия Белл и ЛаПадула доказали теорему, получившую название "основной теоремы безопасности" (ОТБ).

Теорема 2.2.1.(Basic Security Theorem). Система $\Sigma(v_0, Q, F_T)$ безопасна тогда и только тогда, когда:

Состояние v_0 безопасно, и функция переходов F_T такова, что для любого состояния v, достижимого из v_0 при выполнении конечной последовательности запросов из множества Q таких, что при $F_T(v,q)=v^*$, где $v=(F_L,A)$ и $v^*=(F_L^*,A^*)$, переходы системы из состояния в состояние подчиняются следующим ограничениям для $s \in S$ и для $o \in O$

если $read \in A^*[s,o]$ и $read \in A[s,o]$, то $F_L^*(s) \geq F_L^*(o)$;

если $read \in A[s,o]$ и $F_L^*(s) < F_L^*(o)$, то $read \notin A^*[s,o]$;

если $write \in A^*[s,o]$ и $write \notin A[s,o]$, то $F_L^*(s) \geq F_L^*(o)$;

если $write \in A[s,o]$ и $F_L^*(s) < F_L^*(o)$, то $write \notin A^*[s,o]$.

Достоинствами данной модели является формулировка состава безопасной системы в определениях множеств субъектов, объектов, прав доступа и т.д. и то, что сформулированы правила записи/чтения субъектов. Но несмотря на это, при использовании данной модели в контексте практического проектирования и разработки реальных компьютерных систем возникает ряд технических вопросов. Например, применительно к теме данной работы, что будет являться субъектами системы. Данная модель определяет права доступа к записи/чтению разных типов

пользователей. А приложение данной модели к taint-анализу нужно дополнить определением субъектов системы. Это поможет сделать такой вид моделей как информационные.

1.2.2 Информационные модели

Информационные модели определяют ограничения на отношение ввода/вывода системы. Данные модели накладывают ограничения на интерфейс программных модулей системы с целью достижения безопасной реализации. При этом подробности реализации определяются разработчиком системы. Данные модели являются результатом применения теории информации к проблеме безопасности систем. К информационным моделям относятся модели невмешательства и невыводимости.

Модель невмешательства

Невмешательство – ограничение, при котором ввод высоко уровневого пользователя не может смешиваться с выводом низкоуровневого пользователя.

Модель невмешательства рассматривает систему, состоящую из четырех объектов: высокий ввод (high-in), низкий ввод (low-in), высокий вывод (high- out), низкий вывод (low-out).

Рассмотрим систему, вывод которой пользователю *u* определен функцией out(*u*, hist.read(*u*)), где hist.read(*u*) - история ввода системы (traces), чей последний ввод был read(u) (команда чтения, исполненная пользователем *u*). Для определения безопасности системы необходимо определить термин очищения (*purge*) историй ввода, где *purge* удаляет команды, исполненные пользователем, чей уровень безопасности не доминирует над уровнем безопасности *u*. Функция clearence(u) - определяет степень доверия к пользователю (см. 1.2).

Определение: purge - функция users × traces → traces такая, что:

- purge(u, <>) = <>, где <> - пустая история ввода;
- purge(u, hist.command(w)) = purge(u, hist.command(w)), если command(w) – ввод, выполненный пользователем w; clearence(u)≥clearence(w);
- purge(u, hist.command(w)) = purge(u, hist), если command(w) – ввод, выполненный пользователем w; clearence(u)<clearence(w).

Система удовлетворяет требованию невмешательства, если и только если для всех пользователей u, всех историй T и всех команд вывода c out(u,T.c(u)) = out(u,purge(u,T).c(u)).

Модель невыводимости

Рассмотрим модель невыводимости, также базирующуюся на рассмотрении информационных потоков в системе. Модель невыводимости выражается в терминах пользователей и информации, связанных с одним из двух возможных уровней секретности (высокий и низкий).

Система считается невыводимо безопасной, если пользователи с низкими уровнями безопасности не могут получить информацию с высоким уровнем безопасности в результате любых действий пользователей с высоким уровнем безопасности. Другими словами, в таких системах утечка информации не может

произойти в результате посылки высокоуровневыми пользователями высокоуровневой информации к низкоуровневым пользователям. Интуитивно это определение относится не к информационным потокам, а к разделению информации.

Чтение и запись рассматриваются в контексте этой модели как явные операции, вызываемые пользователями компьютерной системы, и выполняются определенной автоматизированной последовательностью вычислительных действий.

Допустим, что машина принимает ввод от высоко- и низкоуровневых пользователей, обрабатывает эти вводы некоторым незаданным образом и затем выдает на выходах к высоко- и низкоуровневым пользователям информацию. Возможно также, что вводят информацию и получают данные вывода одни и те же пользователи. Единственным различием пользователей является то, какой у них уровень безопасности - высокий или низкий.

Если множество вводов от пользователей в машину связано со множеством выводов, получаемых пользователями от машины каким-либо разумным образом (возможно, основываясь на времени их поступления), то тогда можно рассматривать выходную последовательность как путь поведения (traces, см. модель невмешательства) системы. Безопасность невыводимости может быть определена в соответствии со множеством всех путей поведения системы и множеством вводов и выводов, видимых пользователями.

Тогда система может быть признана невыводимо безопасной, если для каждой метки безопасности x и определенного пути есть второй путь, показывающий то же поведение, видимое пользователями с меткой безопасности меньшей или равной x, но не имеющая вводов не меньших или равных x. Другими словами, высокоуровневые вводы могут всегда быть удалены из пути и это не повлияет на то, что видят низкоуровневые пользователи. Можно заметить, что понятие невыводимости не охватывает ситуаций, основанных на концепции «интерпретации информации» в той степени, в которой этого можно было ожидать. Данный недостаток устраняется с помощью ограничения понятия составляющих вводов и выводов модели.

Каждый пользователь связан с определенным взглядом на систему (например видимые вводы и выводы) и может получить информацию, интерпретируя видимое ему поведение. Если система является невыводимо безопасной, то низкоуровневые пользователи не должны получить новой информации, если на вводе системы есть дополнительные высокоуровневые пользователи. Кроме этого, если низкоуровневые пользователи могут получить определенную информацию, основываясь на видимом ими поведении, то удаление высокоуровневых пользователей не должно изменить получаемой низкоуровневыми пользователями информации.

Согласно модели Белла-Лападулы, в системе высокороуровневый субъект имеет доступ на чтение всех объектов, когда как низкоуровневый может читать только объекты доступные для его уровня доступа. А с точки зрения моделей невмешательства и невыводимости, система безопасна только тогда, когда

выполнение некоторой последовательности команд на высоком уровне не будет менять ее состояние.

Таким образом, модели информационного невмешательства и информационной невыводимости также не могут обеспечить теоретико-методологическую базу разработки программно-технических решений в системах представления информации для обеспечения защиты чувствительной информации. Данные модели помогли подробнее, чем модель Белла-ЛаПадуллы, определить ограничения информационных потоков. А также предложили дополнительно поделить вводы/выводы системы на два типа: высокие и низкие. Данные термины помогут в формулировке модели защиты системы, но нужно определить объекты модели в терминах, которые можно применить к задаче защиты чувствительных данных.

1.3 Постановка задачи

Система, синтезированная на основании рассмотренных моделей, должна иметь, по крайней мере, одно средство для обеспечения безопасности на каждом возможном пути проникновения в систему.

Система должна точно определять каждую область, требующую защиту. С каждым объектом, требующим защиты, связывается некоторое множество действий, к которым может прибегнуть злоумышленник для получения несанкционированного доступа к объекту. Необходимо перечислить все потенциальные злоумышленные действия по отношению ко всем объектам безопасности для формирования набора угроз, направленных на нарушение безопасности.

Рассмотренные модели не представляют непосредственно механизмы защиты информации, а могут использоваться только в сочетании. И в дополнение необходимо четко переопределить объекты модели в терминах, которые можно применить к задаче защиты чувствительных данных.

Целью данной работы является разработка автоматизированной разметки исходного кода пользователя согласно клиент-серверной архитектуре на платформе Electron.

Данную цель можно достигнуть, решив следующие задачи:

1) разработка модели защищенной системы;
2) создание средства автоматизированного поиска небезопасных источников данных, позволяющих осуществлять запись/чтение чувствительных данных;
3) создание средства автоматизированной разметки кода пользователя;
4) отладка и тестирование программы на экспериментальных данных.

1.4 Выводы по разделу

Ведущие коммерческие статические анализаторы умеют строить представления веб-приложений, в которых отражены межмодульные зависимости. Соответственно, применение анализа кода к такому представлению дает хороший результат – средство добавляет возможность обнаруживать

межмодульные уязвимости. Но Taint-анализ предназначен только для обнаружения уязвимостей, возникающих из-за некорректной обработки пользовательских данных. При этом защита от этого типа уязвимостей не предлагается. К тому же разработчики предлагают мультиязычные анализаторы, и т.к. большинство из них не являются приложениями с открытым кодом, нельзя сказать с полной уверенностью, что они учитывают особенности отдельных языков из представленных.

Также были рассмотрены средства защиты чувствительных данных языков Perl и Ruby. Инструменты данных языков программирования позволяют отследить движение недоверенных данных уже на этапе разработки.

В данном разделе был произведен обзор широко применяющихся в настоящее время методов защиты информации и моделей информационной защищенности. Приведены концептуальные основы и особенности работы политики Белла-ЛаПадулы и моделей невыводимости и невмешательства.

На основании выполненного обзора можно сделать вывод о том, что рассмотрение защиты чувствительной информации при распределенном выполнении кода, построенная на основе изложенных в работе моделей в отдельности нецелесообразно. И анализ, и обеспечение ее безопасности необходимо проводить в соответствии с некоторым смешением представленных моделей.

ГЛАВА 2. МОДЕЛЬ ЗАЩИТЫ ПРИЛОЖЕНИЯ НА ELECTRON ПРИ РАСПРЕДЕЛЕННОМ ВЫПОЛНЕНИИ КОДА

В рамках данного исследования рассмотрим платформу под названием Electron, созданную в 2013 году. В данном разделе изучим создание десктопных приложений на базе Electron, фреймворка, в рамках которого будет рассмотрена работа разработанной модели защиты чувствительной информации при распределенном выполнении кода.

2.1 Построение Electron приложения

Electron – это фреймворк, который позволяет разрабатывать настольные приложения с использованием HTML, CSS и JavaScript. Это платформа с открытым исходным кодом, принцип работы которой основывается на объединении структур Chromium и Node.js. Приложение работает в браузере Chromium, наполнение которого контролируется средствами JavaScript и Node.js.

Принцип работы Электрона основан на двух типах процессов [12]:

Первый тип — **основной процесс**, который отвечает за интеграцию и взаимодействие с GUI операционной системы. Под этим понятием скрывается интеграция в док на OS X или панель задач на Windows, а также сворачивание в трей, полноэкранный режим и прочие обыденные и нативные для ОС возможности. Такой процесс запускается только один раз на всю жизнь приложения.

Второй тип — **процесс рендеринга**, отвечающий за отображение окна браузера, в котором может быть открыта страница приложения или любая другая веб-страница. Таких процессов может быть произвольное число. За создание процесса рендеринга отвечает основной процесс.

Для того, чтобы породить основной процесс используется следующая схема рождения приложения:

– Электрон читает package.json и ищет в нём секцию **main**, в которой определен основной файл приложения (далее – точка входа).

– Затем происходит обработка «точки входа» и создаётся основной процесс, который в свою очередь, при желании разработчика, открывает какую-либо страницу или страницы, то есть создаёт окно браузера. А если говорить точнее, то порождает процесс или процессы рендеринга.

Жизнь приложения, построенного на Электроне, имеет вид, изображенный на рисунке.

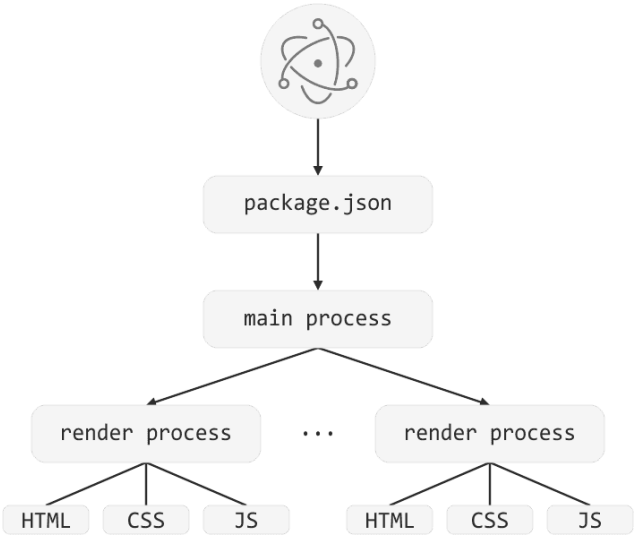

Рисунок 2.1 – Структура приложений на Electron.

В дальнейшем происходит настройка окружения, WebStorm, и после создания приложения появляется окно, которое имеет служебное меню с небольшим набором стандартных пунктов. Служебным оно называется из-за того, что после сборки проекта, как вы увидите позднее, оно исчезает, далее осуществляется распространение приложения.

2.2 Модель защиты системы

В ходе разработки модели анализа системы было решено определить области программы пользователя, требующие защиту, и участки, где необходимо создать средство защиты для обеспечения безопасности на каждом возможном пути проникновения в систему.

Предусматривается определение каждой области, к которой могут быть применены действия по несанкционированному доступу. Также необходимо перечислить все потенциальные источники недоверенных данных по отношению ко всем защищаемым объектам для формирования набора угроз (обозначим множество угроз T), направленных на нарушение безопасности [6].

После определения ключевых объектов системы (с точки зрения безопасности) с помощью абстрактного синтаксического дерева опишем пути распространения данных из источников, которые не являются доверенными.

Таким образом, мы определим пути, которые проходят данные из недоверенных источников к приемникам, позволяющим осуществлять запись/чтение чувствительных данных[8].

На основе построенных деревьев и определенных путей распространения данных, мы можем составить множество отношений источник-приемник. Оно образует двудольный граф, в котором ребро $<t_i, o_j>$ существует тогда и только тогда, когда t_i ($\forall t_i \in T$) является средством получения доступа к объекту o_j ($\forall o_j \in O$). Связь между объектами и угрозами типа «один ко многим», то есть одна угроза может распространяться на любое количество объектов, а объект может

быть уязвим с более, чем одной стороны. Цель защиты заключается в перекрытии каждого ребра графа для создания барьера по этому пути[8].

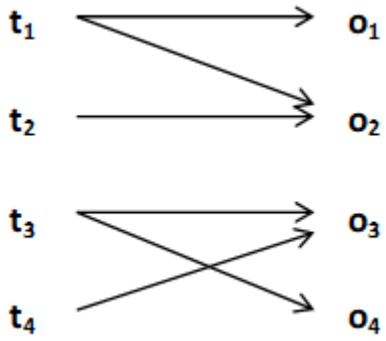

Рисунок 4.1 – Граф источник – приемник

Еще одним набором, завершающим математическую модель анализа системы, является множество C средств обеспечения безопасности. Итак, каждое c_k ($\forall c_k \in C$) должно устранять некоторое ребро $<t_i, o_j>$ из графа на рисунке 2.1. Тогда с помощью набора C двудольный граф недоверенных источников и приемников позволяющим осуществлять запись/чтение чувствительных данных преобразуется в трехдольный граф [9].

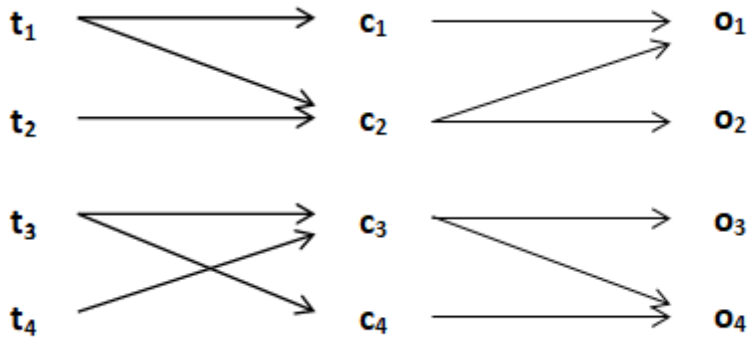

Рисунок 4.2 – Граф источник – защита – приемник

В качестве c_k выступает либо точка, в которой код должен прекратить свое существование и выполнение на устройстве пользователя (клиенте) и храниться в защищенной для выполнения среде (на сервере), либо это какая-то фильтрующая функция, позволяющая объявить используемые программой данные, поступившие извне, доверенными.

Получим таким образом систему, представляющую собой набор M={T, C, O}, где

T – набор угроз;

C – набор защищающих средств;

O – набор чувствительных данных.

Таким образом, получим систему, предлагающую пользователю изменить свой исходный код так, что на каждый путь проникновения будет иметься средство защиты.

2.3 Выводы по разделу

В данном разделе был рассмотрен принцип создания десктопных приложений на базе фреймворка Electron, в рамках которого будет рассмотрена работа разработанной модели защиты чувствительной информации при распределенном выполнении кода.

В рамках данного раздела предложена модель защиты чувствительной информации при распределенном выполнении кода.

Дальнейшее применение предложенной модели позволяет:

1) определить области программы пользователя, требующие защиту;
2) определить средство защиты на каждом участке, где существуют возможные пути проникновения в систему.

В основу модели положены: концептуальные основы и особенности работы политики Белла-ЛаПадулы и моделей невыводимости и невмешательства. Поскольку указанные модели защищенных систем разнообразны по содержанию, то анализ и обеспечение безопасности систем проводится в соответствии с некоторым смешением выше перечисленных моделей.

ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1. Поиск небезопасных источников ввода и критических каналов вывода

Программная реализация содержит в себе модуль поиска мест, содержащих источники недоверенных данных и критические каналы вывода, с помощью анализа узлов синтаксического дерева разбора JavaScript-кода. Такие источники будут помечены как потенциально небезопасные для дальнейшего анализа их распространения и влияния на чувствительные данные пользователя и на целостность самой программы.

Рассмотрим список конкретных источников и небезопасных методов для реализации их поиска и дальнейшего анализа:

1) Объекты, отвечающие за источники данных (пользовательские каналы и сторонние адреса) – такие как, например, location для работы с адресом страницы и строковые операции с ними, и каналы ввода, осуществленные с помощью DOM: eval, document.write, setTimeout, element.innerHTML, script.src. В Листинге 1 продемонстрирован небезопасный вывод URL-адреса какой-либо страницы (веб-страницы) [29]:

```
1 <html><body>
2       <h1>Подробности вы можете найти в документации к приложению
  </h1>
4       <script >document.querySelector('h1 ').innerHTML += 5
  location.href; </script>
5 </body></html>
```

Листинг 1.1 – Пример страницы, содержащей уязвимость

Злоумышленник может вынудить перейти по ссылке с адресом: http://site.com/page.html#<img src=x onerror=alert('XSS');></img>, и после выполнения операции document.querySelector('h1').innerHTML += location.href структура документы станет следующей:

```
1 <html><body>
2       <h1>Поделиться адресом страницы: http :// site.com/page
        .html#<img src=x onerror=alert('XSS ');></img ></h1>
3       <script >document.querySelector('h1 ').innerHTML +=
        location.href; </script >
4 </body></html>
```

Листинг 1.2 – Пример страницы, содержащей уязвимость, после её эксплуатации

В результате изменения структуры страницы появился новый контекст выполнения JavaScript-кода, подконтрольный злоумышленнику, который в приведённом примере содержит вызов «alert('XSS')».

Подобным образом могут использоваться следующие каналы ввода, которые в программе определяют оператор как критический, и в дальнейшем распространение данных от этого источника будет помечаться как недоверенное:

document.URL

document.URLUnencoded

document.location<>
document.referrer
window.location<>
и т.д.
Полный список таких функций находится в Приложении 3.

2) Критические каналы вывода и методы рендеров Electron, передающих контент главному процессу и использующих модули Node.js [11]:

– appendArgument и appendSwitch (использование дополнительных аргументов позволяет вставлять новые аргументы командной строки, что может изменить поведение программы), например, здесь:

«const {app} = require('electron')
app.commandLine.appendArgument(‘debug’)
app.commandLine.appendSwitch(‘proxy-server', '8080')»

при активной отладке приложения злоумышленник может подорвать подключение к порту с помощью стороннего отладчика;

– remote и сообщения IPC рендера (это инструменты для передачи контента между главным процессом и побочными, главным образом могут использоваться злоумышленниками в приложении, когда осуществляется предварительная загрузка скриптов, с помощью имитации того же механизма, только внешними инструментами) выглядят так:

«app = require(‘electron').remote.app» или
«{ipcRenderer} = require('electron')
app = ipcRenderer.sendSync('ELECTRON_BROWSER_GET_BUILTIN', 'app')»;

– insertCSS, executeJavaScript и eval (открывают доступ к передачи контента от рендеров главному процессу, риск ожидается от всех сторонних приложению файлов со скриптами);

– shell.openExternal (позволяет открывать протокол URI от Shell, который использует нативные утилиты устройства пользователя, например, на macOS эта функция работает как «открытый» терминал и может открыть конкретное приложение на основе URI или любой файл), например:

«const {shell} = require('electron')
shell.openExternal('file:///Applications/Calculator.app')»

и т.д. Полный список критических каналов вывода также находится в Приложении 3.

3.2. Разметка кода пользователя

В данной работе уязвимость можно описать как раз как нарушение принципа модели невмешательства:

Пусть программа получает данные из high-level (доверенных) и low-level (недоверенных) каналов и генерирует вывод в high-level (критические) и low-level (обычные) каналы [25].

Где, в данной работе:

– данные, полученные от пользователей, считаются ненадежными; это low-level недоверенные каналы ввода;

– данные, полученные из локального хранилища данных (файловая система, СУБД), считаются надежными; это high-level доверенные каналы ввода;

– ненадежные данные могут стать надежными вследствие специальной обработки (фильтрации);

– ненадежные данные не должны попадать в критические операции (запросы к СУБД, вывод HTML, системные вызовы, eval'ы и т.д.); это high-level критические каналы вывода [24].

Таким образом, мы считаем все места, через которые извне приходят данные, недоверенными каналами ввода. А методы, осуществляющие чтение и использование чувствительных данных, критическими каналами вывода.

Нарушением принципа невмешательства будет, к примеру: *Вывод low-level (недоверенных) данных в high-level (критический) канал считается нарушением принципа невмешательства.*

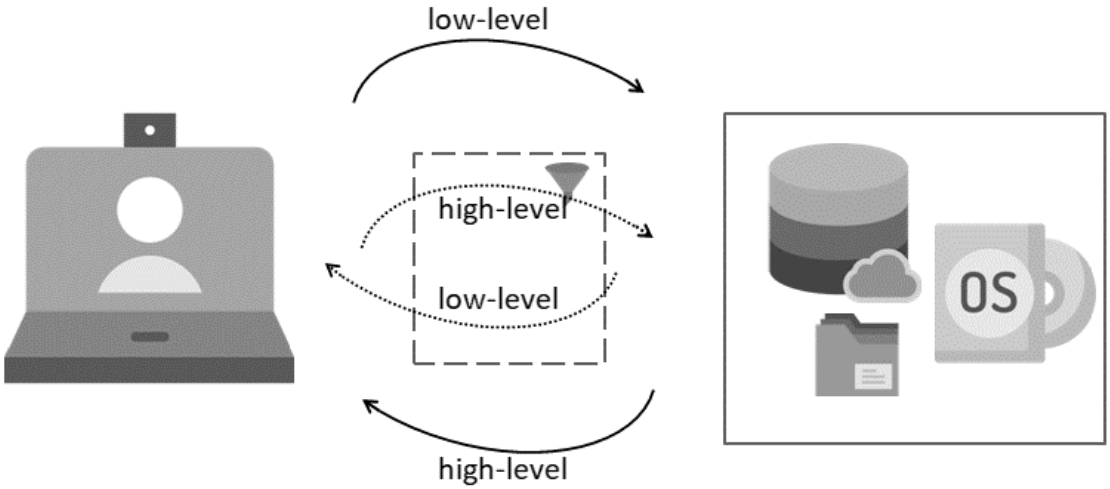

Рисунок 3.1 – Типы каналов, определенных в модели защиты

После определения ключевых объектов системы (с точки зрения безопасности) с помощью анализа абстрактного синтаксического дерева кода опишем пути распространения данных из источников, которые не являются доверенными.

Таким образом, определяются пути, которые проходят данные из недоверенных источников к приемникам, позволяющим осуществлять запись/чтение чувствительных данных.

Получим систему, представляющую собой набор M={T, C, O}, где

T – набор угроз;

C – набор защищающих средств;

O – набор чувствительных данных.

В качестве c_k выступает либо точка, в которой код должен прекратить свое существование и выполнение на устройстве пользователя (клиенте) и храниться в защищенной для выполнения среде (на сервере), либо это какая-то фильтрующая

функция, позволяющая объявить используемые программой данные, поступившие извне, доверенными.

Разметка кода пользователя предполагает разделение всех операторов и функций на две группы: те, которые обязательно должны находиться в защищенной среде выполнения – на сервере, и те, которые можно оставить для выполнения на клиенте [17].

Порядок разметки предполагает после определения источников/каналов данных определение путей, которые могут проходить доверенные данные и недоверенные данные.

Согласно разработанной модели:

– чувствительные данные не должны попасть в критические каналы вывода, если для их вывода используются данные из недоверенного источника;

– когда существует путь, по которому чувствительные данные попадают в критические каналы вывода, если для вывода используются данные из недоверенного источника, вся цепочка операторов помечается «грязной» (taint);

– когда существует путь, по которому чувствительные данные попадают в критические каналы вывода, если для вывода не используются данные из недоверенного источника, вся цепочка операторов помечается «чистой» (clean).

На рисунке 5.2 показана примерная схема распределения кода пользователя.

Исследуемый оператор может быть определен на сервер без изменений в случаях:

1) данные, обрабатываемые оператором, пришли из обращения к операционной системе/базе данных/файловой системе (os/db/fs);
2) оператор объявляет переменную в теле программы, которая в дальнейшем имеет отношение к обращению (чтению/записи) к os/db/fs;

В случае если оператор выполняет чтение/запись в os/db/fs с помощью недоверенных данных, то цепочка помечается как грязная. Это обозначает, что при обращении к чувствительным данным на стороне сервера выполняется метод encodeURI для входного потока. Данная конструкция позволит избавиться от символов (":", "/", ";", и "?"), которые могут содержаться в потенциальных атаках.

Для того, чтобы снять метку «taint» с цепочки операторов, было решено добавлять в код функцию, фильтрующую введенные пользовательские данные. После фильтрации недоверенных данных, где это возможно, цепочка (или подмножество в ней) операторов отмечается как clean и теперь обладает разметкой server, если выполняется критерий 1).

С помощью модуля, определяющей доверенные/недоверенные источники данных и критические каналы вывода, по исходному коду пользователя определяются наборы T и O.

Результат работы модуля после анализа кода пользователя представляется в виде двух массивов объектов – lowInputs (узлы, которые были определены как недоверенные источники ввода) и highOutputs (критические каналы вывода) с полями:

stSymbol – позиция, с которой начинается определение оператора;

finSymbol – позиция последнего символа оператора.

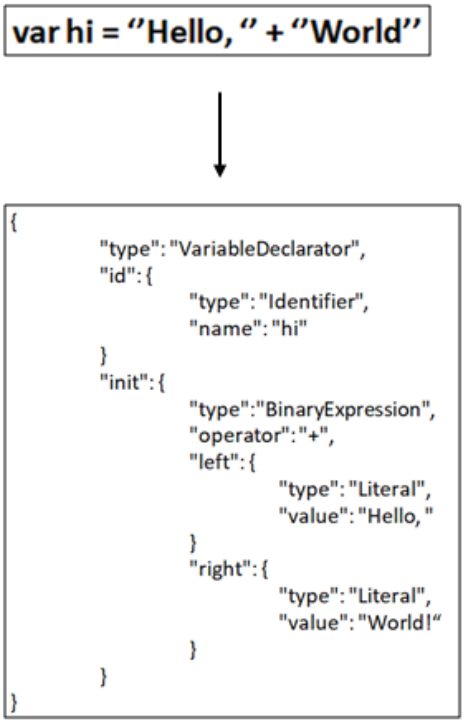

Рисунок 3.2 – Пример работы модуля esprima

Далее модуль разметки кода пользователя генерирует отчет, в котором даны рекомендации по разметке.

Для однозначной разметки кода для клиентской и серверной части введём условную директиву. Чтобы не нарушать синтаксическую корректность исходного кода, но при этом выполнить разделение в исходном файле, обозначение директивы выполним в виде комментария на языке JavaScript. Для выделения начала блока у сервера будем использовать «// #if server», у клиента «// #if client», а для обозначения окончаний блоков «// #endif». Оператор или функция принимает значение «server» если, согласно списку источников данных и критических каналов вывода данных, необходимо защитить среду их выполнения. Остальные операторы и функции помечаются как «client», т.к. возможно их выполнение не влияет на чувствительные данные пользователя и целостность системы.

Достоинством данной конструкции является ее синтаксическая корректность относительно языка JavaScript, так как выполнена в виде комментариев. Семантически данная конструкция схожа с аналогичной в языке C#, при незначительном увеличении файла с исходным кодом пользователя.

3.3 Пример работы программы

В качестве приложения на Electron для поиска небезопасных источников данных было выбрано приложение Instatron. Это неофициальное десктопное клиентское приложение для пользователей Instagram.com. Оно инициализирует

вход в аккаунт пользователя и позволяет использовать полный функционал этого сервиса.

Исходный код приложения можно найти на github.com по ссылке [29].

В ходе работы программы был сформирован файл result.log со списком найденных узких мест. Рассмотрим этот список:

1	DOM API:	1	41
2	Enabled nodeIntegration	null	null
3	Command line arguments	2	12,13
4	IPC messages	null	null
5	HTTP-connections, URL's	2	41,127
6	Allowed InsecureContent	null	null
7	Disabled webSecurity	null	null
8	ExperimentalFeatures	null	null
9	Sessions	null	null
10	External scripts	null	null
11	Enabled opening new windows	null	null
12	Shell's openExternal	2	114,118

Как видно из списка, анализируемое приложение из всего списка предлагаемых источников недоверенных данных и узких мест использует следующие:

«Command line arguments», «HTTP-connections, URL's», «Shell's openExternal».

Таблица 1. Пример работы модуля разметки

Исходный код	Пример отчета

`const {app, BrowserWindow, dialog, Menu, session} = require('electron') // http://electron.atom.io/docs/api`
 	
`require('electron-context-menu')({`
`  prepend: (params, browserWindow) => []`
`})`
 	
`let window = null`
 	
`let currentValue;`
`var currentProperty;`
 	
`... // user input currentValue and`
`    // currentProperty;`
 	
`if (process.platform === 'win32') {`
`  app.commandLine.appendSwitch('high-dpi-support', 'true')`
 	
`app.commandLine.appendSwitch(currentProperty, currentValue)`
`}`
 	
`app.once('ready', () => {`
`  window = new BrowserWindow({`
`    webPreferences: {`
`      nodeIntegration: false`
`    }`
`  })`
 	
`...`
	`// #if client`
	`const {app, BrowserWindow, dialog, Menu, session} = require('electron') // http://electron.atom.io/docs/api`
	
	`require('electron-context-menu')({`
	`  prepend: (params, browserWindow) => []`
	`})`
	
	`let window = null`
	
	`let currentValue;`
	`var currentProperty;`
	
	`... // user input currentValue and`
	`    // currentProperty;`
	
	`if (process.platform === 'win32') {`
	`  app.commandLine.appendSwitch('high-dpi-support', 'true')`
	`// #endif`
	
	`// #if server`
	
	`  filter(currentValue);`
	`  filter(currentProperty);`
	`app.commandLine.appendSwitch(currentProperty, currentValue)`
	`// #endif`
	`}`
	
	`// #if client`

```
  const url =
document.querySelector('h1').innerHTML +=
location.href; // taint

  window.loadURL(url, {
    userAgent: 'Mozilla/5.0'
  })

  window.once('ready-to-show', () => {
    window.show()
    if (process.env.NODE_ENV !== undefined
&& process.env.NODE_ENV.trim() === 'dev') {
      window.webContents.openDevTools()
    }
  })
})

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') {
    app.quit()
  }
})
```

```
app.once('ready', () => {
  window = new BrowserWindow({
    webPreferences: {
      nodeIntegration: false
    }
  })

...

  const url =
document.querySelector('h1').innerHTML +=
location.href; // taint
// #endif

// #if server
  filter(url);
// #endif

// #if client
  window.loadURL(url, {
    userAgent: 'Mozilla/5.0'
  })

  window.once('ready-to-show', () => {
    window.show()
    if (process.env.NODE_ENV !== undefined
&& process.env.NODE_ENV.trim() === 'dev') {
      window.webContents.openDevTools()
    }
  })
})

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') {
    app.quit()
  }
})
// #endif
```

3.4 Выводы по разделу

В разделе описано средство для автоматизированного поиска возможных небезопасных источников данных и критических каналов вывода, а также принцип и пример работы модуля разметки кода пользователя.

ЗАКЛЮЧЕНИЕ

В работе было реализовано средство для автоматизированного поиска возможных небезопасных источников данных и критических каналов вывода, создан модуль разметки кода пользователя.

В начале работы был проведен анализ анализаторов кода и средств языков, которые исследуют веб-приложения в контексте защиты чувствительных данных. Ведущие коммерческие статические анализаторы умеют строить представления веб-приложений, в которых отражены межмодульные зависимости. Соответственно, применение анализа кода на грязные данные к такому представлению дает хороший результат – средство добавляет возможность обнаруживать межмодульные уязвимости. Но Taint-анализ предназначен только для обнаружения уязвимостей, возникающих из-за некорректной обработки пользовательских данных. При этом данный тип уязвимостей имеет целый ряд ограничений, из-за которых мультиязычные анализаторы, реализующие её поиск, заведомо не могут иметь 100% эффективность. К тому же большинство из них не являются приложениями с открытым кодом, нельзя сказать с полной уверенностью, что они учитывают особенности отдельных языков из представленных.

Далее был произведен обзор широко применяющихся в настоящее время методов защиты информации и моделей информационной защищенности. Приведены концептуальные основы и особенности работы политики Белла-ЛаПадулы и моделей невыводимости и невмешательства. А также сделан вывод о том, что рассмотрение защиты чувствительной информации при распределенном выполнении кода, построенная на основе изложенных в работе моделей в отдельности нецелесообразно. И анализ и обеспечение ее безопасности необходимо проводить в соответствии с некоторым смешением представленных моделей.

В ходе работы был рассмотрен принцип создания десктопных приложений на базе Electron, фреймворка, в рамках которого предложена математическая модель защиты чувствительной информации при распределенном выполнении кода. Дальнейшее применение предложенной модели позволило определить области программы пользователя, требующие защиту и определить средство защиты на каждом участке, где существуют возможные пути проникновения в систему. В основу модели положены: концептуальные основы и особенности работы политики Белла-ЛаПадулы и моделей невыводимости и невмешательства.

В работе описан перечень потенциальных небезопасных методов и свойств DOM API, а также фреймфорка Electron, и приведен пример работы реализации их поиска в приложении, созданном с помощью Node.js и Electron. Также согласно разработанной модели защиты системы реализована разметка кода пользователя. Проанализировано реальное приложение на Electron, и сделан вывод о найденных уязвимостях.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Balzarotti, D. Multi-module vulnerability analysis of web-based applications / Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, Giovanni Vigna — CCS '07 Proceedings of the 14th ACM conference on Computer and communications security, 2007. — С. 25-35
2. Andrew Powell-Morse, Ruby Exception Handling: SecurityError, [Электронный ресурс]. — Режим доступа: https://airbrake.io/blog/ruby-exception-handling/securityerror, свободный — Дата обращения 28.12.2016
3. D Foy, B. Mastering Perl / Brian D Foy — O'Reilly Media, 2009. — С. 32-36.
4. Robert, K. CGI Programming with Perl / Kirrily Robert, Paul Fenwick, Jacinta Richardson — O'Reilly Media, 2012. — С. 206-209.
5. HP Fortify Static Code Analyzer Custom Rules Guide — Hewlett- Packard Development Company, 2014. — С. 19-22.
6. Девянин П.Н. Модели безопасности компьютерных систем: Учеб. пособие / П.Н. Девянин. – М.: Изд.центр «Академия», 2005. – 144 с.
7. Гайдамакин Н.А. Разграничение доступа к информации в компьютерных системах / Н.А. Гайдамакин – Екатеринбург: изд-во Урал. Ун-та, 2003. – 328 с.
8. Зегжда Д.П., Ивашко А.М. Основы безопасности информационных систем / Д.П. Зегжда, А.М. Ивашко – М.:Горячая линия – Телеком, 2000. – 452с.
9. Хоффман, Дж. Современные методы защиты информации / Дж. Хоффман – М.: Сов. радио, 1980. – 264 с.
10. Jasim, M. Building Cross-Platform Desktop Applications with Electron / Muhammed Jasim – Paperback – April 28, 2017.
11. Построение Electron приложения. Введение, [Электронный ресурс]. — Режим доступа: https://canonium.com/articles/electron-desktop-app-introduction, свободный — Дата обращения 21.11.2017.
12. Хэррон, Д. «Node.js. Разработка серверных веб-приложений в JavaScript»: Пер. с англ. Слинкина А.А. / Дэвид Хэррон – М.: ДМК Пресс, 2012. – 144 с.
13. Крейн, Д. Ajax в действии / Дейв Крейн, Эрик Паскарелло, Даррен Джеймс. – Диалектика, 2006 г. – 649 с.
14. Три полных пэ. Python, PHP или Perl? Выбираем последнюю букву в слове "LAMP" [Электронный ресурс]: журн. Хакер. - Электрон. журн. - режим доступа к журн.: https://xakep.ru/pdf/xa/127
15. Mikowski M., Powell J. Single Page Web Applications. Manning, 2014.
16. Алекс Маккоу. Веб-приложения на JavaScript. Москва: издательство “Питер”, 2012.
17. JavaScript - URL:https://developer.mozilla.org (дата обращения: 26.05.2018).
18. Mario Casciaro. Node.js Design Patterns. UK: Packt Publishing, 2014.
19. Node.js - URL:https://nodejs.org/ (дата обращения: 04.05.2015)
20. Шэлли Пауэрс. Изучаем Node.js. Москва: издательство “Питер”, 2014
21. Node Package Manager–URL:https://npmjs.com
22. Michael Sutton, Adam Greene, and Pedram Amini. Fuzzing : brute force

vulnerabilty discovery. Upper Saddle River, NJ: Addison-Wesley, 2007.
23. Takanen, Ari, Jared D. Demott, and Charles Miller. Fuzzing for software security testing and quality assurance. Boston: Artech House, 2008.
24. Code Injection–URL: https://www.owasp.org/index.php/Code_Injection (Дата обращения: 23.01.18)
25. Fuzzing : the Past, the Present and the Future – URL: http://actes.sstic.org/SSTIC09/Fuzzing-the_Past-the_Present_and_the_Future/SSTIC09-article-A-Takanen-Fuzzing-the_Past-the_Present_and_the_Future.pdf (Дата обращения: 23.01.18)
26. Mark Dowd, John McDonald, and Justin Schuh. The art of software security assessment: identifying and preventing software vulnerabilities. Indianapolis, Ind: Addison-Wesley, 2007.
27. What Are Application Security Risks – URL: https ://www.owasp.org/index.php / Top_10_2013 - Risk (Дата обращения 1.06.2018).
28. XSS уязвимость и защита от XSS – URL: http://lifeexample.ru/razrabotka-i-optimizacia-saita/xss-uyazvimos-i-zashhita-ot-xss.html (Дата обращения 1.06.2018).
29. Instatron – Instagram desktop uploader – URL: https://github.com/alexdevero/instatron (Дата обращения 25.05.2018).

ПРИЛОЖЕНИЕ 1. ОПИСАНИЕ ПРОГРАММЫ

Программа разработана на языке JavaScript. Приложение запускается в любой операционной системе с установленной Node.js.

П.1.1 ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ

Программа предназначена для преобразования программы исходного файла на языке JavaScript. В результате ее исполнения будут получен новый файл с разметкой на клиент и сервер с комментариями.

П.1.2 ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА

Для исполнения программы рекомендуется ПК со следующими параметрами:

— Операционная система, с поддержкой Node.js
— Node.js с библиотекой esprima
— Объем оперативной памяти 1 Гб
— Дисковое пространство размером 64 Мб

П.1.3 ВЫЗОВ ПРОГРАММЫ

В консоли перейти в нужную директорию и запустить класс программы. Например, командой « node mapping.js source.js newcode=mappedcode.js ». В качестве параметров командной строки задается путь к файлу и название файла, куда сохраняется результат работы.

П.1.4 ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

Входными данными является путь к файлу исходного кода на языке JavaScript, имена файлов для сохранения размеченного кода.

Выходными данными является файл на языке JavaScript: код с выделенными блоками для исполнения на сервере и клиенте.

ПРИЛОЖЕНИЕ 2. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

П.2.1 ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММЕ

Программа написана на языке программирования JavaScript.
Требования к окружению, в котором запускается программа:
— Операционная система, с поддержкой Node.js
— Node.js версии не ниже 6.11.3. с библиотекой esprima
— Объем оперативной памяти 1 Гб
— Дисковое пространство размером 64 Мб

П.2.2 НАСТРОЙКА ПРОГРАММЫ

Необходимо проверить, что имеющееся программное и аппаратное обеспечение соответствует требованиям, описанным в разделе П.2.1. Дополнительных действий по настройке программы не требуется

П.2.3 ВЫПОЛНЕНИЕ ПРОГРАММЫ

Для запуска программы требуется запустить файл «mapping.js» на исполнение в среде Node.js. В коммандной строке можно передать следующие аргументы:

— mapping.js – название файла программы генератора кода
— filename – имя файла с исходным кодом, преобразование которого необходимо выполнить
— newcode=строка – имя файла, куда необходимо сохранить код с выделенными блоками для исполнения на сервере и клиенте. Например, newcode=mappedcode.js.

Пример команды: node mapping.js source.js newcode=mappedcode.js

П.2.4 СТРУКТУРА ПРОГРАММЫ

Программа состоит из двух модулей: sinkfinder, mapping.

П.2.5 ПРОВЕРКА ПРОГРАММЫ

```
PC@DESKTOP-GTEF6MH MINGW64 /d/projects/MagicMapping
$  node mapping.js source.js newcode=mappedcode.js
Input file 'source.js'
================================================================================
Source's and sink's searching [success], Total time: 0.12 sec
Total operands: 41
--------------------------------------------------------------------------------
Mapping [success], Total time: 0.88 sec
Server's operands: 13
Client's operands: 28
Filter's add: 5
================================================================================
Output file: mappedcode.js, lines: 58
```

Рис. П.2.1. Пример исполнения программы

П.2.6. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

Входными данными является путь к файлу исходного кода на языке JavaScript, имена файлов для сохранения размеченного кода.

Выходными данными является файл на языке JavaScript: код с выделенными блоками для исполнения на сервере и клиенте.

ПРИЛОЖЕНИЕ 3. СПИСОК ИСТОЧНИКОВ И КАНАЛОВ ВЫВОДА

Sources:

```
document.URL
document.documentURI
document.URLUnencoded
document.baseURI
location
location.href
location.search
location.hash
location.pathname
document.cookie
document.referrer
window.name
```

Sinks:

```
app.commandLine.appendArgument
app.commandLine.appendSwitch
remote
ipcRenderer.sendSync
insertCSS
executeJavaScript
eval
shell.openExternal
document.write
document.writeln
document.body.innerHTML=
document.forms[...].action=
document.attachEvent
document.create...
document.execCommand
window.attachEvent
document.cookie=
document.domain=
document.location=
document.location.hostname=
document.location.replace
document.location.assign
document.location.URL=
window.navigate
document.open
window.open
window.location.href=
window.execScript
window.setInterval
window.setTimeout
```

ПРИЛОЖЕНИЕ 4. ИСХОДНЫЙ ТЕКСТ ПРОГРАММЫ

Модуль поиска небезопасных источников данных

```
function VariableFinder() {
    var esprima = require('esprima');

    var variables = [];

    function extraction(node){
        switch(node.type){
            case 'VariableDeclarator':
                variables.push(node.id.name);
                break;
            case 'AssignmentExpression':
                if (node.left.name){
                    variables.push(node.left.name);
                } else if (node.left.property.name){
                    variables.push(node.left.property.name)
                }

                break;
        }
    }

    this.GetVariables = function(code){
        variables = [];
        var codeTree = esprima.parseScript(code, {}, extraction);
        return variables;
    }

    function traverse(node, func) {
        func(node);//1
        for (var key in node) { //2
            if (node.hasOwnProperty(key)) { //3
                var child = node[key];
                if (typeof child === 'object' && child !== null) { //4

                    if (Array.isArray(child)) {
                        child.forEach(function(node) { //5
                            traverse(node, func);
                        });
                    } else {
                        traverse(child, func); //6
                    }
                }
            }
        }
    }

    this.GetAssignmentsFrom = function(tree){
        variables = [];
        traverse(tree, extraction);
        return variables;
    }
};
```

```
    module.exports = VariableHunter;


    function ConvertForSatementToFunction(statement, functionName){
        let newNode = [];
        if (statement.init!==null){
            newNode.push(statement.init)
        }
        let body = {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": functionName
            },
            "params": [],
            "body": {
                "type": "BlockStatement",
                "body": []
            },
            "generator": false,
            "expression": false,
            "async": false
        };
        if (statement.body!==null) {
            body.body = statement.body;
        }

        return newNode;
    }

    function GetNewContext(){
        var p = this;
        var c = function(){
            this.parent = p;
        };
        return c();
    }

    function ExecuteFunctionByName(functionName, context) {
        var args = Array.prototype.slice.call(arguments, 2);
        var namespaces = functionName.split(".");
        var func = namespaces.pop();
        for(var i = 0; i < namespaces.length; i++) {
            context = context[namespaces[i]];
        }
        return context[func].apply(context, args);
    }

    var contextStack = [];

    function DefaultListener(evt){
        let respJson = evt.data;
        let resp = JSON.parse(respJson);
        ExecuteFunctionByName(resp.FunctionName,   contextStack[contextStack.length   -
1])
    }
```

```
var newContext = {};
var child;

function addName(){
    this.name = 'Hello';
    newContext =  this;
    this.ololo = function(){
        console.log('ololo');
        child = this;
    }
}

function addLastName(){
    this.lastname = 'ivanov';
}

function print(){
    console.log(lastname + ' ' + name);
}

addName();
newContext.ololo();

addLastName.apply(child);
print.apply(child);
```

Модуль разметки кода пользователя:

```
var static = require('node-static');
var esprima = require('esprima');
var fs = require('fs');
var escodegen = require('escodegen');

var arr = [];
arr.back = function() { return this[this.length - 1]}


var VariableHunter = require('./VariableHunter');
var hunter = new VariableHunter();

var NodeNumber = 0;
var PromiseNumber = 0;

var clientVariables =  [[]];
var serverVariables = [[]];

var isLastNode = [];

var clientStack = [{
    "type": "Program",
    "body": [],
    "sourceType": "script"
}];

var serverStack = [{
    "type": "Program",
    "body": [],
```

```
        "sourceType": "script"
    }];

    var currentFunctionOwnerStack = [];

    const NodeObjects = ['require', 'global', 'process',
        'Buffer', 'module', 'exports','__dirname', '__filename'];

    const    BrowserObjects    =    ['$',    'Chart',    'window',    'document',    '$',
'lastMeasureTimes'];

    /**
     * @return {string}
     */
    function GetFunctionNodeName(number){
        return 'FunctionNode' + number;
    }

    function GetPromiseName(number){
        return 'promise' + number;
    }

    function AddVariable(type, name){
        if (type === 'Server'){
            serverVariables[serverVariables.length - 1].push(name);
        } else {
            clientVariables[clientVariables.length - 1].push(name);
        }
    }

    function IsNodeObject(name){
        for (var i = 0; i < NodeObjects.length; i++){
            if (NodeObjects[i] == name){
                return true;
            }
        }
        for (let i = serverVariables.length - 1; i >= 0; i--){
            for (let j = 0; j < serverVariables[i].length; j++) {
                if (serverVariables[i][j] === name) {
                    return true;
                }
            }
        }
        return false;
    }

    function IsBrowserObject(name){
        for (var i = 0; i < BrowserObjects.length; i++){
            if (BrowserObjects[i] == name){
                return true;
            }
        }
        for (let i = clientVariables.length - 1; i >= 0; i--){
            for (let j = 0; j < clientVariables[i].length; j++) {
                if (clientVariables[i][j] === name) {
                    return true;
                }
```

```
            }
        }
        return false;
    }

    function GetNodeExecutionWrapper(node) {
        let nodefunctionJS = `this.FunctionNodeName = function () {};`;
        var newNode = (esprima.parseScript(nodefunctionJS, {})).body[0];

        newNode.expression.left.property.name = GetFunctionNodeName(NodeNumber++);
        newNode.expression.right.body.body.push(node);

        if (!isLastNode[isLastNode.length - 1]){
            let nextNodeJS = GetFunctionNodeName(NodeNumber) + ".apply(context);";
            newNode.expression.right.body.body.push((esprima.parseScript(nextNodeJS,
{})).body[0]);
        } else {
            let  nextNodeJS = `contextStack.pop();`;
            newNode.expression.right.body.body.push((esprima.parseScript(nextNodeJS,
{})).body[0]);
        }
        return newNode;
    }

    function GetCallNode(node){
        let assignments = hunter.GetAssignmentsFrom(node);
        let params = [];
        for (let i = 0; i < assignments.length; i++){
            let obj = {};
            obj.type = "Literal";
            obj.value = assignments[i];
            obj.raw = "\""+assignments[i]+"\"";
            params.push(obj);
        }
        let callNodeJS = `this.FunctionNodeName = function(){
        websocket.send(JSON.stringify(CreateGetPackage("CallFunctionNodeName",
["123"])));}`;

        let callNode = (esprima.parseScript(callNodeJS, {})).body[0];
        callNode.expression.left.property.name = GetFunctionNodeName(NodeNumber++);

callNode.expression.right.body.body[0].expression.arguments[0].arguments[0].arguments
[0].value = GetFunctionNodeName(NodeNumber);

callNode.expression.right.body.body[0].expression.arguments[0].arguments[0].arguments
[0].raw = "\"" + GetFunctionNodeName(NodeNumber) + "\"";

callNode.expression.right.body.body[0].expression.arguments[0].arguments[0].arguments
[1].elements = params;
        return callNode;
    }

    function GetResponseNode(node){
        let assignments = hunter.GetAssignmentsFrom(node);
        let params = [];
        for (let i = 0; i < assignments.length; i++){
            let obj = {};
```

```
            obj.type = "Literal";
            obj.value = assignments[i];
            obj.raw = "\""+assignments[i]+"\"";
            params.push(obj);
        }
        let respJS = `this.FunctionNodeName = function(){
            client.send(
                CreateSetPackage("NextFunctionNodeName",      GetVariables.apply(this,
[["cpus"]]))
        );}`;
        let resp = (esprima.parseScript(respJS, {})).body[0];
        resp.expression.left.property.name = GetFunctionNodeName(NodeNumber++);
        resp.expression.right.body.body[0].expression.arguments[0].arguments[0].value
= GetFunctionNodeName(NodeNumber);
        resp.expression.right.body.body[0].expression.arguments[0].arguments[0].raw   =
"\""+GetFunctionNodeName(NodeNumber)+"\"";

resp.expression.right.body.body[0].expression.arguments[0].arguments[1].arguments[1].
elements[0].elements = params;
        resp.expression.right.body.body.unshift(node);
        return resp;
    }


    function AddNode(who, node){
        let  currentScope  =  currentFunctionOwnerStack[currentFunctionOwnerStack.length
- 1];

        let servernode = null;
        let clientnode = null;

        if (currentScope!=null && currentScope!== who){

            let callNode = GetCallNode(node);
            let execNode = GetResponseNode(node, true);

            if (currentScope === 'Server'){
                servernode = callNode;
                clientnode = execNode;
            } else {
                servernode = execNode;
                clientnode = callNode;
            }

        } else if (currentScope!=null){
            if (who === 'Server') {
                servernode = GetNodeExecutionWrapper(node, true);
            } else {
                clientnode = GetNodeExecutionWrapper(node, true);
            }
        } else {
            if (who !== 'Server'){
                clientnode = node;
            } else {
                servernode = node;
            }
        }
```

```
    if (servernode!==null){
        if (serverStack.length - 1 > 0) {
            serverStack[serverStack.length - 1].body.body.push(servernode);
        } else {
            serverStack[serverStack.length - 1].body.push(servernode);
        }
    }
    if (clientnode!==null){
        if (clientStack.length - 1 > 0) {
            clientStack[clientStack.length - 1].body.body.push(clientnode);
        } else {
            clientStack[clientStack.length - 1].body.push(clientnode);
        }
    }
}

function AddPromiseNode(who, functionName, variableName){
    let promiseName = GetPromiseName(PromiseNumber++);

    let codeJS = `let ` +promiseName+` = new Promise((resolve, reject) => {
        `+functionName+`(resolve);
    });

    this.`+GetFunctionNodeName(NodeNumber++)+`=function(){
        `+promiseName+`.then(result => {
            context.`+variableName+` = result;
            `+GetFunctionNodeName(NodeNumber)+`.apply(context);
        });
    }`;

    let body = (esprima.parseScript(codeJS, {})).body;

    if (who === 'Server'){
        if (serverStack.length - 1 > 0) {
            serverStack[serverStack.length - 1].body.body.push(body[0]);
            serverStack[serverStack.length - 1].body.body.push(body[1]);
        } else {
            serverStack[serverStack.length - 1].body.push(body[0]);
            serverStack[serverStack.length - 1].body.push(body[1]);
        }
    } else
    {
        if (clientStack.length - 1 > 0) {
            clientStack[clientStack.length - 1].body.body.push(body[0]);
            clientStack[clientStack.length - 1].body.body.push(body[1]);
        } else {
            clientStack[clientStack.length - 1].body.push(body[0]);
            clientStack[clientStack.length - 1].body.push(body[1]);
        }
    }
}

//return true is serverMajor, otherwise false, null if undefined nodes win
function GetAssignmentMajor(node){
    if (!node){
        return 'None';
    }
```

```
    if (node.type === 'ArrayExpression'){
        if (node.elements === []){
            return 'None';
        }
        let s = 0, c = 0, ud = 0;
        for (let i = 0; i < node.elements.length; i++){
            if (node.elements[i].type === 'Literal'){
                ud++;
            } else if (node.elements[i].type === 'Identifier'){
                if (IsBrowserObject(node.elements[i].name)){
                    c++;
                } else if (IsNodeObject(node.elements[i].name)){
                    s++;
                } else {
                    ud++;
                }
            }
        }
        return ud > s && ud > c ? null : s > c;
    }
    if (node.type === 'CallExpression'){
        if (node.callee.type === 'MemberExpression'){
            let name = node.callee.object.name;
            if (IsNodeObject(name)){
                return true;
            } else if (IsBrowserObject(name)){
                return true;
            } else {
                return null;
            }
        } else if (node.callee.type === 'Identifier'){
            let name = node.callee.name;
            if (IsNodeObject(name)){
                return true;
            } else if (IsBrowserObject(name)){
                return true;
            } else {
                return null;
            }
        }
    }
    return null;
}

function VariableDeclarationToThisAssignment(declaration, kind){
    let result = {};
    if (currentFunctionOwnerStack[currentFunctionOwnerStack.length - 1] === null){
        result.type = 'VariableDeclaration';
        result.declarations = [];
        result.kind = kind;
        result.declarations.push(declaration);
    } else {
        let thisJS = 'this.' + declaration.id.name + '= null;';
        result = (esprima.parseScript(thisJS, {})).body[0];
        if (declaration.init !== null) {
            result.expression.right = declaration.init;
        }
```

```
        }
        return result;
    }

    function UpdateNewNodeArgs(node){
        if (node.expression.arguments===undefined || node.expression.arguments.length
== 0){
            return;
        }
        for (let i = 0; i < node.expression.arguments.length; i++){
            if (node.expression.arguments[i].type === 'ArrowFunctionExpression'){
                let                                newNode                                =
TraverseArrowFunctionReturnNode(node.expression.arguments[i]);
                node.expression.arguments[i].body = newNode.body;
            }
        }
        return node;
    }

    function ClassifyVaribleDeclaration(node){
        for (let i = 0; i < node.declarations.length; i++){
            let  newNode  =  VariableDeclarationToThisAssignment(node.declarations[i],
node.kind);
            let name = node.declarations[i].id.name;
            if (node.declarations[i].init === null){
                //AddVariable('Client', name);
                AddNode('Client', newNode);
                continue;
            }
            let initType = node.declarations[i].init.type;
            if (initType === 'Literal' || initType === 'ArrayExpression'){
                AddNode('Client', newNode);
                AddVariable('Client', name)
                continue;
            }
            if (initType === 'Identifier'){
                if (IsNodeObject(node.declarations[i].init.name)) {
                    AddNode('Server', newNode);
                } else {
                    AddNode('Client', newNode);
                }
                continue;
            }
            if (initType === 'CallExpression'){
                let classification = GetAssignmentMajor(node.declarations[i].init);
                let needPromise = false;
                if (currentFunctionOwnerStack[currentFunctionOwnerStack.length - 1] &&
                    node.declarations[i].init.callee.type === 'Identifier'){
                    needPromise = true;
                }
                if (classification === true){
                    AddVariable('Server', name);
                    if (needPromise){
                        AddPromiseNode('Server',
node.declarations[i].init.callee.name, name);
                    } else {
                        AddNode('Server', newNode);
```

```
                    }
                } else if (classification === false){
                    if (needPromise){
                        AddPromiseNode('Client',
node.declarations[i].init.callee.name, name);
                    } else {
                        AddNode('Client', newNode);
                    }
                } else {
                    if (needPromise){

AddPromiseNode(currentFunctionOwnerStack[currentFunctionOwnerStack.length    -    1],
node.declarations[i].init.callee.name, name);
                    } else {

AddNode(currentFunctionOwnerStack[currentFunctionOwnerStack.length - 1], newNode);
                    }
                }
                continue;
            }
            if (initType === 'MemberExpression'){
                if (IsNodeObject(node.declarations[i].init.object.name)){
                    AddNode('Server', newNode);
                } else {
                    AddNode('Client', newNode);
                }
                continue;
            }
        }
    }

    /**
     * @return bool
     */
    function ClassifyExpressionStatement(node){
        if (node.expression.type === 'CallExpression'){
            if (!node.expression.callee){
                return null;
            }
            if (node.expression.callee.type === 'MemberExpression'){
                let name = node.expression.callee.object.name;
                if (IsNodeObject(name)){
                    return true;
                } else if (IsBrowserObject(name)){
                    return false;
                } else {
                    return null;
                }
            } else if (node.expression.callee.type === 'Identifier'){
                let name = node.expression.callee.name;
                if (IsNodeObject(name)){
                    return true;
                } else if (IsBrowserObject(name)){
                    return false;
                } else {
                    return null;
                }
```

```
        }
    }
}

function TraverseFunctionDeclaration(node){
    let isServer = false;
    let  funtionJS = `function ` + node.id.name + `() {
        this.context = this;
        contextStack.push(context);
    }`;
    let body = (esprima.parseScript(funtionJS, {})).body[0];
    body.params = node.params;
    if (isServer){
        serverVariables.push([]);
        serverStack.push(body);
        currentFunctionOwnerStack.push('Server');

    } else {
        clientVariables.push([]);
        clientStack.push(body);
        currentFunctionOwnerStack.push('Client');
    }

    let firstNode = NodeNumber;
    GenerateCode(node.body);

    let lastNodeJs = GetFunctionNodeName(firstNode)+'.apply(context);';
    let lastNode = (esprima.parseScript(lastNodeJs, {})).body[0];

    if (isServer){
        serverVariables.pop();
        let f = serverStack.pop();
        f.body.body.push(lastNode);
        serverStack[serverStack.length - 1].body.push(f);
    } else {
        clientVariables.pop();
        let f = clientStack.pop();
        f.body.body.push(lastNode);
        clientStack[clientStack.length - 1].body.push(f);
    }
    currentFunctionOwnerStack.pop();
}

function TraverseArrowFunctionReturnNode(node, isServer){
    let  funtionJS = `function f() {
        this.context = this;
        contextStack.push(context);
    }`;
    let body = (esprima.parseScript(funtionJS, {})).body[0];
    body.params = node.params;
    if (isServer){
        serverVariables.push([]);
        serverStack.push(body);
        currentFunctionOwnerStack.push('Server');

    } else {
        clientVariables.push([]);
```

```
            clientStack.push(body);
            currentFunctionOwnerStack.push('Client');
        }

        let firstNode = NodeNumber;
        GenerateCode(node.body);

        let lastNodeJs = GetFunctionNodeName(firstNode)+'.apply(context);';
        let lastNode = (esprima.parseScript(lastNodeJs, {})).body[0];

        currentFunctionOwnerStack.pop();

        if (isServer){
            serverVariables.pop();
            let f = serverStack.pop();
            f.body.body.push(lastNode);
            return f;

        } else {
            clientVariables.pop();
            let f = clientStack.pop();
            f.body.body.push(lastNode);
            return f;
        }
    }
    function AddReturnOperations(node){
        if (node.argument === null){
            return;
        }
       // let lastOperationsJS = `{returnValue.push(`+node.argument.name+`);
       //     resolve();
       //     contextStack.pop();}`;
        let lastOperationsJS = `resolve(`+node.argument.name+`);
            `
        let lastOperation = (esprima.parseScript(lastOperationsJS, {})).body[0];
        //lastOperation.body[0].expression.arguments[0] = node.argument;
        AddNode(currentFunctionOwnerStack[currentFunctionOwnerStack.length    -    1],
lastOperation);
        if   (currentFunctionOwnerStack[currentFunctionOwnerStack.length   -   1]   ===
'Server'){
            if (serverStack[serverStack.length - 1].params === null){
                serverStack[serverStack.length - 1].params = [];
            }
            serverStack[serverStack.length - 1].params.push(
                {
                    "type": "Identifier",
                    "name": "resolve"
                }
            );
        } else {
            clientStack[clientStack.length - 1].params = [
                {
                    "type": "Identifier",
                    "name": "resolve"
                }
            ];
        }
```

```
    }

    function Traverse( node){
        if (node.type === 'ReturnStatement'){
            AddReturnOperations(node);
            return;
        }
        if (node.type === 'ForStatement'){
            AddNode('Client', node);
            return;
        }
        if (node.type === 'VariableDeclaration'){
            ClassifyVaribleDeclaration(node);
            return;
        }
        if (node.type === 'AssignmentExpression'){
            return;
        }
        if (node.type == 'ExpressionStatement'){
            let type = ClassifyExpressionStatement(node);
            if (node.expression.type === 'CallExpression') {
                UpdateNewNodeArgs(node);
            }
            if (type === null){
                AddNode(currentFunctionOwnerStack[currentFunctionOwnerStack.length    -
1], node);
            } else {
                AddNode(type ? 'Server' : 'Client', node);
            }
            return;
        }
        if (node.type == 'FunctionDeclaration'){
            TraverseFunctionDeclaration(node);
            return;
        }
        for (var key in node) {
            if (node.hasOwnProperty(key)) {
                var child = node[key];
                if (typeof child === 'object' && child !== null) {
                    if (Array.isArray(child)) {
                        child.forEach(function(node) {
                            Traverse(node, Traverse);
                        });
                    } else {
                        Traverse(child, Traverse);
                    }
                }
            }
        }
    }

    function GenerateCode(tree){
        isLastNode.push(false);
        if (tree.body){
            for (let i = 0; i < tree.body.length; i++){
                isLastNode[isLastNode.length - 1] = false;
                if (i == tree.body.length - 1){
```

```
                isLastNode[isLastNode.length - 1] = true;
            }
            Traverse(tree.body[i]);
        }
    }
    isLastNode.pop();
}


var program = fs.readFileSync('input.js', 'utf8');
currentFunctionOwnerStack.push(null);
GenerateCode(esprima.parseScript(program, {}));


fs.writeFile(userNewFile, escodegen.generate(codeStack[codeStack.length - 1]),
function (err) {
    if (err) {
        console.log('Cannot save user code');
        return;
    };
    console.log('Output file: ' + userNewFileName.toString(userNewFile) + ',
lines: ' + userFileLength.toString(length-1));
});
```