Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИЧЕРСИТЕТ»**

**(МОСКОВСКИЙ ПОЛИТЕХ)**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА «ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ»

Отчет выполнения по Проектной деятельности

**«Оценка уязвимостей смарт-контрактов»**

Выполнила: Николаец Д.И

Студент 3-го курса

Группы 171-361

Москва 2020

**Цели задачи 2** проекта «Оценка уязвимостей смарт-контрактов»:

* Изучение принцип работы методов анализа уязвимостей смарт-контрактов (статический и динамический), анализ их преимуществ и недостатков
* Выбор популярных существующих методик и инструментов анализа уязвимостей смарт-контрактов. Анализ и сравнение методик и инструментов путем выделения их особенностей
* Выбор наиболее сбалансированную комбинацию методики и инструмента, которую можно использовать для поиска и анализа уязвимостей в коде смарт-контракта.

**Результат работы:**

* Рассмотрен сравнительный анализ методик и инструментов. Приведены примеры методик и инструментов анализа уязвимостей смарт-контрактов, их характеристики
* На основе анализа преимуществ и недостатков, сформирован вывод какую наиболее сбалансированную комбинацию методики и инструмента, можно использовать для поиска и анализа уязвимостей в коде смарт-контракта.

Рассмотрим два направления в анализе безопасности уязвимостей статический и динамический анализ.

**DAST**   
  
Динамический анализ приложений методом черного ящика самый простой и распространённый способ поиска уязвимостей. Собственно, каждый раз «вставляя кавычку в URL» или вводя '>  мы осуществляем эту мудреную процедуру. По сути это [fault injection](http://en.wikipedia.org/wiki/Fault_injection) (aka fuzzing) приложения, путем эмуляции клиентской части и попытки отправки на вход «хорошо известных плохих данных ».  
Простота метода приводит к большому количеству реализаций, и в «магическом квадранте» Gartner просто тесно от конкурентов. Более того, «движки» DAST присутствуют в большинстве систем контроля уязвимостей и соответствия стандартов, таких как MaxPatrol, за исключением разве что Symantec Control Compliance Suite. Распространены и некоммерческие решения, так базовый (очень-очень базовый) модуль DAST присутствует в Nessus, более продвинутые механизмы существуют в w3af или sqlmap.  
К преимуществам DAST относится простота использования и отсутствие необходимости доступа к серверной части приложения. Также серьезным плюсом является относительная независимость от платформы, фреймворков и языков, на которых разработано приложение. Учет этих нюансов может повысить эффективность анализа, но это скорее оптимизация, добавляющая проценты к общей эффективности.  
  
Однако у DAST есть и обратная сторона. Перечислим списком:

* Невысокая степень покрытия. Далеко не все вызовы API и точки входа можно легко обнаружить методом черного ящика.
* Драматическое падение эффективности при усложнении клиента/протокола. Приложения Web 2.0, JSON, Flash, HTML 5.0 и JavaScript требует либо динамического (например, эмуляцией выполнения JavaScript) либо статического разбора («отгрепать» Flash или taint-анализ того же JavaScript) клиентской части, что значительно усложняет клиентскую часть фазера, приближая его к «полноценному» браузеру.
* Ненулевая вероятность нарушения целостности и доступности (например при попадании конструкций типа or 1=1 в UPDATE через SQL Injection).
* Долгое время работы. Вашему покорному слуге практически ни разу не удавалось наблюдать завершения работы большинства из утилит класса DAST на достаточно «разлапистом» сайте, раньше заканчивалось окно на тестирование. Тут хорошо подходит правило Парето – за 20% времени найти 80% уязвимостей.
* Сложность выявления многих типов. Например, ошибки использования криптографии, такие как слабые механизмы генерации cookie или session ID (кроме самых примитивных случаев) DAST обнаруживает крайне плохо.

**SAST: Quick and dirty**  
  
Многие из методов статического анализа генерируют большое количество «подозрений на уязвимость», которые на проверку оказываются ложными срабатываниями, что существенно увеличивает трудозатраты. Невозможность выявления некоторых классов уязвимостей. Собственно «статичность». Примеров тут множество, и генерируемый «на лету» код, и хранение кода/данных в СУБД, файловой системе и т.д. Отдельных трудозатрат разработчикам добавляет зависимость от языков (и даже версий),

фреймворков. Так, чтобы понять точки входа и точки выхода, фильтрующие функции недостаточно понимания «голого» языка, необходимо распознавать те библиотеки и фреймворки, которые используют разработчики в реальном мире.

И у SAST и DAST есть свои преимущества и недостатки и достаточно долгое время муссируется идея об объединении результатов работы этих методов или о «гибридном анализе» (hybrid analysis), что позволит взять лучшее из этих двух подходов. Не смотря, на то, что данная гипотеза, что называется «лежит на поверхности» и относится к интуитивно понятным, реализация данной концепции на практике не дает ожидаемых результатов.

*Далее рассмотрим частые методики.*

**Инструментирование(code instrumentation)** - это отслеживание параметров уровня производительности кода, возможность диагностировать ошибки и записывать информацию на всём протяжении работы для отслеживания причин их возникновения. Таким образом можно проверить смарт-контракт в «боевых» условиях.

**Символьное выполнение(symbolic execution)** – это средство анализа программы для определения того, какие входные данные вызывают выполнение какой части программы. Интерпретатор следует за программой, принимая символические значения для входных данных, а не получая фактические входные данные, как это было бы при обычном выполнении программы. Другими словами, оно позволяет находить недочеты в коде, даже не зная, какое значение переменных будет в строке с ошибкой.

**Устранение ограничений(Constraint solving)** означает определение разрешимости ограничений и возможность вычисления конкретного решения. Ограничение - это набор условий, которым должны удовлетворять переменные. В нашем контексте ограничения в основном возникают из условий ветвления в коде.

**Абстрактная интерпретация(Abstract interpretation)** игнорирует определенные инструкции или определенные эффекты инструкций при выполнении байт-кода. Это можно сделать, переведя инструкции в другой формализм, например DataLog, а затем изучив все возможные варианты выполнения.

**Логика Хорна(Horn Logic)**-это ограниченная форма логики первого порядка, где все формулы (предложения) являются правилами if-then. Хотя логика Хорна и ограничена, она все же является вычислительно универсальной, поэтому может выполнять те же вычисления, что и любой компьютер.

**Проверка моделей(Model checking)** - это метод автоматической проверки правильности свойств конечных систем. Для этого требуется модель системы, которая затем проверяется на соответствие заданной спецификации.

*Рассмотрим инструменты:*

**Scompile** берет байт-код контракта, строит CFG, определяет все вычислительные пути, включающие любой поток эфира, выбирает те, которые соответствуют паттернам, характерным для определенных уязвимостей, ранжирует их эвристически в соответствии с релевантностью и, наконец, применяет символическое исполнение, прежде чем представить результат пользователю для ручной проверки.

**Mythril** - это инструмент командной строки в Python для интерактивного анализа смарт-контрактов. Он выполняет байт-код виртуальной машины символически и визуализирует CFG, причем узлы, содержащие разобранный код, и ребра помечаются формулами пути. Проверенные уязвимости подробно описаны в интерактивной документации. Mythril разрабатывается и поддерживается компанией Consensus, а также доступен на Github под лицензией MIT с сентября 2017 года.

**Securify** принимает байт-код EVM и свойства безопасности в качестве входных данных. Инструмент декомпилирует байт-код, ориентированный на стек, в форму, основанную на присвоении, и представляет код в виде фактов журнала данных. Затем он выводит дополнительные факты, которые описывают управление и поток данных в абстрактной форме. Свойство безопасности состоит из паттернов соответствия и нарушения, чрезмерно приближенных как к удовлетворению, так и к неудовлетворению этого свойства. Эти паттерны кодируются как правила ведения журнала данных, которые можно сверить с фактами, используя Soufflé. Этот подход гарантирует, что при обнаружении паттерна код определенно обладает/нарушает соответствующее свойство безопасности. Этот инструмент написан на Java и доступен на Github под лицензией Apache 2.0 с сентября 2018 года. Кроме того, доступ к закрытой версии исходного кода можно получить через веб-сайт компании Chain Security.

**Manticore** использует символическое выполнение, чтобы найти уникальные пути вычислений в ЭВМ и двоичные ELF-файлы. Он записывает соответствующие следы выполнения. Что касается ЭВМ, то Мантикора компилирует код солидности в байт-код для его анализа, проверяет трассировки на наличие уязвимостей, таких как повторное проникновение и достижимые операции саморазрушения, и сообщает о них в контексте исходного кода. Информация о методах и их ограничениях весьма скудна. Этот инструмент разработан и поддерживается компанией Trail of Bits и доступен на Github под лицензией AGPL-3.0 с февраля 2017 года. Его можно использовать из командной строки или через API Python.

**MAIAN** рассматривает атаки, требующие нескольких транзакций. Он выполняет байт-код VM символически и проверяет наличие следов исполнения, указывающих на то, что контракт может быть самоуничтожен или слит эфира с произвольных адресов, или что он принимает эфир без выплаты. Чтобы отбросить ложные срабатывания, контракты динамически анализируются, развертывая их на частном блокчейне и атакуя их с помощью вычисленных транзакций. MARIAN написана на Python и доступна на Github под лицензией MIT с марта 2018 года. Он использует компилятор солидности для компиляции исходного кода в байт-код и Go-Ethereum для запуска частного блокчейна. Mayan -это в основном инструмент командной строки, но также предоставляет простой графический пользовательский интерфейс, который требует графической библиотеки Qt.

Есть много академических проектов, которые оказали большое влияние на реальный мир, например, на обнаружение важных ошибок в программном обеспечении с открытым исходным кодом, с помощью символьного выполнения. Я считаю, что красота **символьного выполнения** как техники заключается в том, что по сравнению с тестированием, это дает вам возможность представить, как ваша программа будет себя вести с потенциально бесконечным набором возможных входных данных. Это позволяет исследовать массивы входных данных, которые было бы совершенно нецелесообразно и непрактично исследовать, скажем, случайным тестированием даже при наличии очень большого количества тестеров. С другой стороны, по сравнению с более традиционными методами статического анализа оно имеет следующее преимущество. При исследовании проблемы символьное выполнение может создать входные данные и трассировку, путь выполнения, которые можно запустить в реальной программе и выполнить эту программу на основе этих входных данных. И после этого мы можете выявить реальный баг и приступить к его исправлению, используя традиционные механизмы отладки. И это особенно ценно, когда вы находитесь в промышленной среде разработки, где у вас, вероятно, нет времени, чтобы заботиться о каждой маленькой проблеме в вашем коде.

Для методики «символьное выполнение» лучшим инструментом является **MAIAN**.

**Maian** анализирует смарт-контракты, определенные в файле с помощью:

* Солидность исходного кода
* Источник байт-кода
* Байт-код скомпилирован (т. е. код, сидящий на блокчейне Maian проверяет наличие трех типов контрактов на баги:

**Maian** проверяет наличие трех типов контрактов на баги:

* Самоубийственные контракты (могут быть убиты кем угодно, например, контракт с библиотекой Parity Wallet)
* Блудные контракты (может отправить эфир кому угодно)
* Жадные контракты (никто не может выйти из эфира)

Рассмотрим **сравнение** инструментов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TOOLS** | **SCompile** | **Manticore** | **Mythril** | **Securify** | **MAIAN** |
| Security issues | Галочка | Галочка | Галочка | Галочка | Галочка |
| Exploits |  | Галочка | Галочка |  | Галочка+ |
| Formal guarantees |  |  |  | Галочка |  |
| Bulkanalysis | Галочка |  |  | Галочка | Галочка |
| Bytecode |  | Галочка+ | Галочка+ | Галочка+ | Галочка+ |
| Solidity code | Галочка |  |  |  |  |
| Static analysis | Галочка | Галочка | Галочка | Галочка | Галочка |
| Dynamic analysis |  |  |  |  | Галочка |
| Contextualization | Галочка | Галочка | Галочка | Галочка |  |
| Disassembly | Галочка | Галочка | Галочка | Галочка | Галочка |
| Control ﬂow graph | Галочка |  | Галочка |  | Галочка |
| Call graph |  |  |  |  |  |
| ASTanalysis |  |  |  |  |  |
| Decompilation |  |  |  | Галочка |  |
| **Code instrumentation** |  |  |  |  |  |
| **Symbolic execution** | Галочка | Галочка | Галочка |  | Галочка |
| **Constraint solving** | Галочка | Галочка | Галочка |  | Галочка |
| **Abstract interpretation** |  |  |  | Галочка |  |
| **Horn logic** |  |  |  | Галочка |  |
| **Model checking** |  |  |  |  |  |

Проблемы безопасности(Security issues): обнаружение уязвимостей и потенциальных проблем безопасности.

Эксплойт(Exploit): генерирует эксплойты, причем **+** указывает на мультитранзакционные эксплойты.

Формальные гарантии(Formal guarantees): доказывает, что договор имеет определенное свойство. Массовый анализ(Bulk analysis): подходит для анализа больших наборов контрактов

Статический анализ(Dynamic analysis)

Динамический анализ (Dynamic analysis)

Символьное выполнение(Symbolic execution)

Решение ограничений(Constraint solving)

Абстрактная интерпретация(Abstract interpretation)

Логика хорна (Horn logic)

Проверка модели(Model checking)