guardant

конференция РусКрипто

Автоматическая защита исполняемых модулей для ARM-архитектуры

Михаил Бакаляров,
Руководитель департамента разработки Guardant

Обзор рынка готовых решений на начало 2023 года

Протекторы с поддержкой ARM:

- UPX https://upx.github.io/
- Virbox Protector https://lm-global.virbox.com
- Thales Sentinel https://cpl.thalesgroup.com/software-monetization/

Открытые LLVM-обфускаторы

Obfuscator-LLVM

- https://github.com/obfuscator-llvm/obfuscator/wiki/Installation
- https://github.com/heroims/obfuscator/wiki/Installation

HikariObfuscator

- https://github.com/HikariObfuscator/Hikari/tree/release_80
- https://github.com/61bcdefg/Hikari-LLVM15/tree/llvm-15.0.2rel

Различные подходы к решению задачи запутывания кода

1

Протектор

- Защита исполняемого модуля
- Уровень инструкций процессора
- Декомпиляция для каждой отдельной платформы

2

LLVM-обфускатор

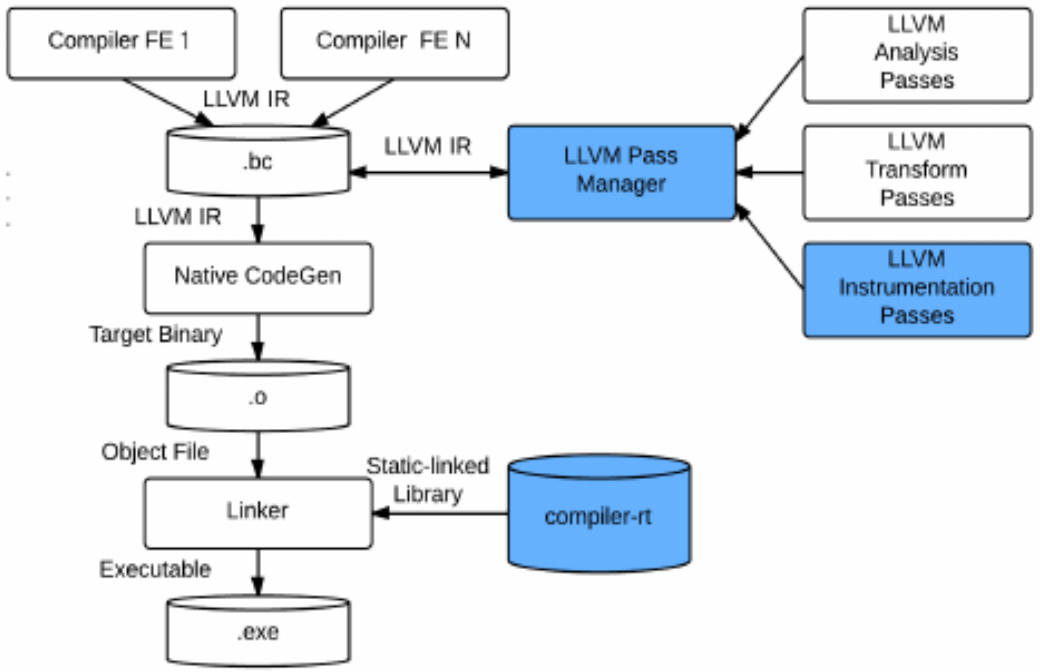
- Защита исполняемого кода на этапе компиляции
- Уровень низкоуровневого промежуточного представления LLVM IR
- Поддержка разных целевых платформ

Архитектура LLVM-обфускатора

- Обфускация не зависящая от языка программирования.
- Независимая от бэкенда обфускация.
- Преобразования в основном выполняются на внутреннем представлении LLVM IR.
- Доступ к реальным переменным и функциям приложениям.

- Некоторые трюки, специфичные для процессора, не могут быть реализованы в общем виде.
- Некоторая информация недоступна на уровне IR (размер функции,...).

Процесс компиляции в LLVM



- LLVM-обфускатор является частью LLVM-фреймворка.
- LLVM предоставляет API с обширной документацией.
- Проход в LLVM — это метод структурирования, используемый компилятором для преобразования и оптимизации исходного кода.
- Преобразование исходного кода может использоваться также для его запутывания.

Оптимизация и обфускация

Замена инструкций (Instructions substitution)

- Addition
  - `a = -(-b + (-c))`
  - `r = rand (); a = b + r; a = a + c; a = a - r`
  - `r = rand (); a = b - r; a = a + b; a = a + r`
- AND
  - `a = b & c` => `a = (b ^ ~c) & b`
- OR
  - `a = b | c` => `a = (b & c) | (b ^ c)`
- XOR
  - `a = a ^ b` => `a = (~a & b) | (a & ~b)`

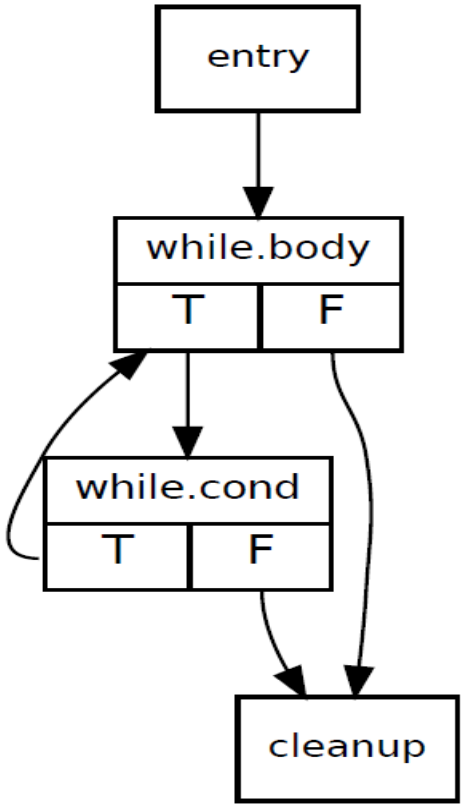
- Заменяет простые арифметические и логические операции более сложными, но эквивалентными.
- Выражение логических операций через базисы.
- В коде приложения создаются избыточные переменные.
- Случайное значения переменных вычисляются во время компиляции.
- Можно применять замены несколько раз.
- Начальное случайное число из командной строки придает некоторую дополнительную уникальность результирующему двоичному файлу.

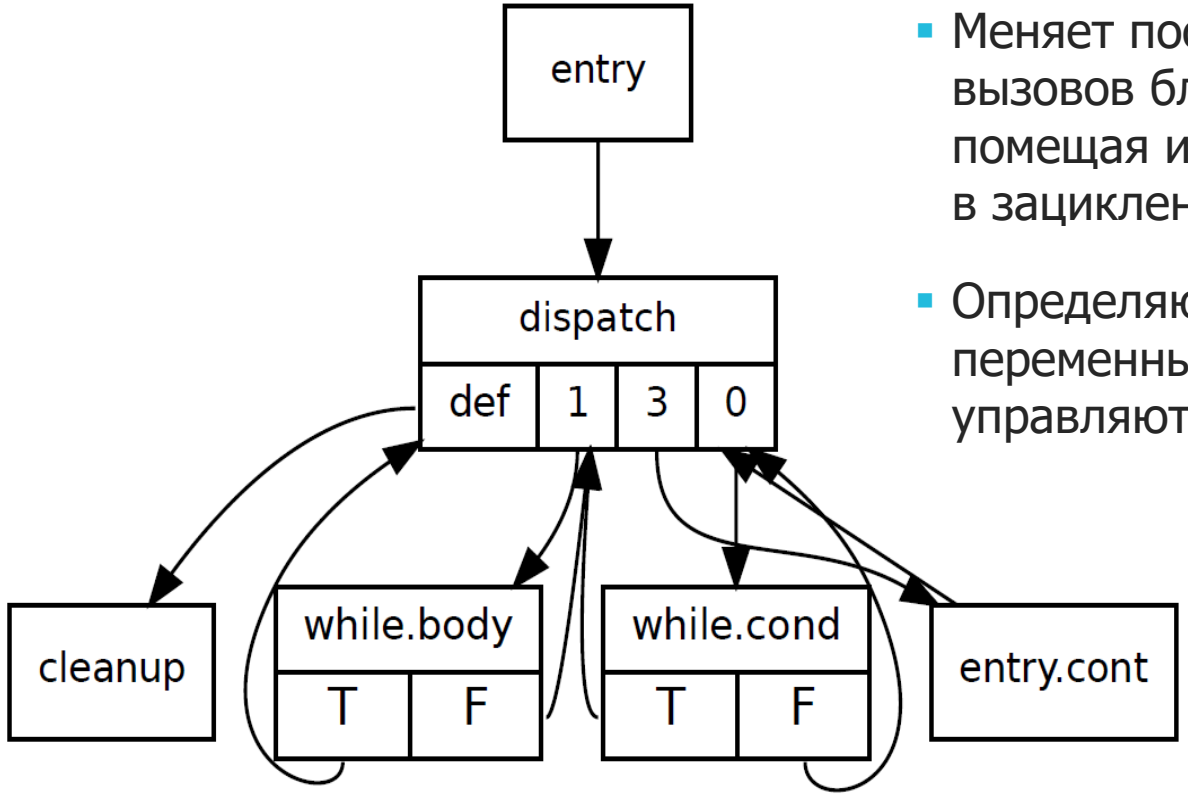
Ложные потоки управления
(Bogus control flow)

$7y^2 - 1 \neq x^2$
$2 \mid x(x+1)$
$3 \mid x(x+1)(x+2)$
$x^2 > 0$
$7x^2 + 1 \not\equiv 0 \mod 7$
$x^2 + x + 7 \not\equiv 0 \mod 81$
$x > 0$ for $x \in I$ random where $I \subset \mathbb{N} = \mathbb{Z}_{>0}$ is a random interval

- Добавляет непрозрачные предикаты перед блоками инструкций.
- Непрозрачный предикат — это выражение, результат выполнения которого заранее определен логическим значением (истина или ложь).
- За непрозрачным предикатом следует условный переход на исходный блок инструкций.
- Обфускация может применяться несколько раз и может быть нацелена на случайные блоки кода.

Сглаживание потока управления
(Control flow flattening)





- Меняет последовательность вызовов блоков инструкций, помещая их на один уровень в зацикленном switch-операторе.
- Определяются дополнительные переменные, которые фактически управляют порядком выполнения.

HikariObfuscator

- BogusControlFlow — улучшенная версия алгоритма создания ложных потоков управления.
- FunctionCallObfuscate — обфускация вызова внешних функций.
- FunctionWrapper — фиктивные функции, которые обертывают фактический вызов функции.
- IndirectBranching — инструкции ветвления заменены косвенным ветвлением на основе регистров на поддерживаемом бэкэнде. Из-за этого дизассемблеры не могут предсказать полный поток управления из статического анализа.
- StringEncryption — шифрование строк.

Контроль целостности

Бинарный уровень:

- Один раз при старте — легко снимается.
- Случайный контроль в разных местах — потеря производительности.

Уровень функций:

- Подсчёт контрольной суммы каждой функции индивидуально — наиболее оптимальный вариант.

Проблемы:

- В LLVM-IR нет объекта который бы указывал на конец функции.
- Нет гарантии что код функции является непрерывным.
- В коде функции присутствуют релокации.
- Арифметика указателей и разыменование указателя на функцию должны работать.

JIT-компилятор

- Использовать LLVM-jitter для генерации кода во время выполнения.
- Функция превращается в массив данных.
- Легко контролировать целостность.
- Возможность расшифровывать на лету.

- Фреймы исключений C++ нужно будет регистрировать вручную.
- NX-бит защиты может быть выставлен на страницу.
- Адрес функции будет известен только в момент выполнения.

Выводы

- Протектор и LLVM-обфускатор решают одну задачу, но разными способами.
- В теории LLVM-обфускация должна быть сильнее.
- На практике в LLVM-обфускаторе присутствует зависимость от бэкенда.
- UPX и O-LLWM обфускатор работают под ARM и являются открытыми.
- Лучшим вариантом будет модифицировать открытые решения и сделать кастомную обфускацию.

Вопросы

guardant

guardant

bma@guardant.ru
+7 903 198-23-39
www.guardant.ru