Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б10-мм

Обфускация глобальных переменных

Зекашева Анастасия Темирбековна

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. преподаватель кафедры ИАС, К.К.Смирнов

Консультант:
Старший разработчик ПО I категории ООО «Софтком» П.А.Бабанов

Санкт-Петербург
2024

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор **5**

2.1. Методы обфускации данных 5

2.2. LLVM Passes 5

2.3. Аналоги 6

2.4. Выводы 7

3. Описание решения **9**

3.1. Алгоритм 9

3.2. Реализация 10

4. Эксперимент **12**

4.1. Декомпиляция обфусцированного кода в IDA 12

4.2. Замер времени исполнения программы 13

4.3. Результаты 14

Заключение **15**

Список литературы **16**

Введение

В современном промышленном программировании защита программных продуктов от несанкционированного доступа является одной из важнейших задач. Особое внимание уделяется защите от реверс-инжиниринга, поскольку он предоставляет злоумышленникам возможность восстановить исходный код и получить доступ к конфиденциальной информации. Для минимизации подобных рисков широко применяются различные методы обфускации такие, как скрытие логики программы, усложнение потока управления, добавление мертвого кода и другие техники.

Однако наряду с обфускацией структуры программы важно также уделять внимание защите данных, хранящихся в глобальных переменных. В них могут содержаться такие важные данные, как криптографические ключи, параметры шифрования, пользовательские данные, сертификаты или параметры алгоритмов. Открытый доступ к подобным данным может представлять угрозу безопасности системы.

В компании «Софтком» ведётся проект по созданию обфускатора для компилятора Clang. В рамках проекта уже разработаны несколько плагинов, включая плагины для модификации графа потока управления и арифметических преобразований. Эти плагины взаимодействуют с промежуточным представлением LLVM. Благодаря работе на уровне IR, обфускация не зависит от целевой архитектуры и может применяться к коду для различных платформ. Проект нуждается в расширении, одним из возможных его продолжений является создание плагина для обфускации глобальных переменных.

Таким образом, в данной работе будет создан обфускатор для компилятора Clang, который модифицирует промежуточное представление LLVM с целью усложнения доступа к глобальным и статическим переменным. Эффективность обфускации будет протестирована на устойчивость к средствам реверс-инжиниринга, после чего решение будет интегрировано в проект компании «Софтком».

1. Постановка задачи

Цель данной работы — разработка обфускатора для компилятора Clang, который будет усложнять доступ к глобальным переменным модуля и локальным статическим переменным функций. Для ее выполнения потребуется выполнить следующие задачи:

- провести обзор существующих методов и инструментов для обфускации;
- спроектировать алгоритм обфускации для данных и реализовать плагин для Clang в инфраструктуре LLVM-17, который будет применять предложенный алгоритм на этапе компиляции для автоматической обфускации переменных;
- провести валидацию результата работы, используя средства реверс-инжиниринга для анализа защищенности обфусцированного файла;
- замерить время исполнения обфусцированного и необфусцированного файла.

2. Обзор

2.1. Методы обфускации данных

Обфускация данных — это процесс скрытия конфиденциальных данных от несанкционированного доступа. Наиболее часто используемые методы обфускации данных [1]:

- маскировка, или анонимизация, данных — обфускация данных посредством создания поддельных, реалистичных значений данных. Часто используется в сборе статистики и анализе данных, при котором необходимо сохранять реалистичность данных и при этом конфиденциальность;
- шифрование данных — преобразование данных в шифротекст, использует ключи (симметричное, асимметричное шифрование);
- токенизация — процесс замены данных неконфиденциальным эквивалентом — токеном, который не имеет смысла в контексте искомого кода.

Существует также множество других техник обфускации данных, таких как перемешивание, размытие, нулирование, повторяющаяся маскировка, недетерминированная рандомизация (замена реального значения на случайное допустимое).

2.2. LLVM Passes

Реализовать преобразование констант планируется в инфраструктуре LLVM. В этой системе с помощью фреймворка проходов (Passes) выполняются анализ и модификация кода на уровне промежуточного представления (IR LLVM) [6].

Существует несколько типов проходов, каждый из которых работает на разном уровне:

1. Проходы на уровне модуля (Module Passes). Модуль в LLVM представляет собой набор функций и глобальных переменных, поэтому такие проходы анализируют или модифицируют все функции сразу. Примером может быть инлайнинг функций или устранение мёртвого кода (DCE) на уровне модуля.

2. Проходы на уровне графа вызовов (Call Graph SCC Pass). Эти проходы работают с графом вызовов (Call Graph) — структурой, отображающей взаимосвязи между функциями программы. Здесь проходы могут анализировать циклические зависимости в графе (Strongly Connected Components, SCC) или выполнять трансформации, связанные с функциями, которые вызывают друг друга.

3. Проходы на уровне функций (Function Passes). Эти проходы анализируют или модифицируют код внутри одной функции. Примеры таких проходов включают подсчет конкретных инструкций в функции, а также сворачивание констант.

4. Проходы на уровне циклов (Loop Pass Manager). Используются для анализа и оптимизации циклов, таких как разворачивание циклов или удаление инвариантов из тела цикла.

Для поставленной задачи потребуется регистрация прохода на уровне модуля, так как глобальные переменные определяются для всего модуля разом.

2.3. Аналоги

Из уже существующих инструментов для обфускации можно выделить следующие:

- OLLVM (Obfuscator-LLVM) [4] — система обфускации на уровне LLVM IR, предоставляющая такие функции, как изменение потока управления (-bcf), подмена инструкций на уровне команд (-sub) и упрощение структуры управляющих потоков (-fla). Однако у OLLVM есть несколько недостатков. Во-первых, OLLVM

основан на LLVM версии 4.0, что может привести к отсутствию поддержки некоторых новых функций API более поздних версий LLVM. Во-вторых, так как OLLVM — это проект с открытым исходным кодом, для него легче создать инструменты для обратной разработки. В-третьих, OLLVM не предлагает плагинов для обфускации данных [3];

- Tigress [7] — обфускатор исходного кода на языке C. Tigress предлагает множество методов обфускации: изменение потока управления, разделение функций, шифрование констант, а также механизмы для усложнения использования отладчиков и инструментов динамического анализа. В контексте обфускации данных у Tigress есть плагин, шифрующий целочисленные переменные при хранении и обращении к ним и дешифрующий при выводе, например, в функции print. Также в Tigress можно задать, какое именно арифметическое преобразование будет использовано для кодировки: линейное преобразование, XOR, RNC (Residue Number Coding — представление чисел на основе остатков от деления на несколько взамно простых модулей) или комбинация вышеперечисленных преобразований.

- VMProtect [5] — коммерческий проект, который использует виртуализацию кода для усложнения реверс-инжиниринга. VMProtect преобразует критические участки программы в байткод для собственной виртуальной машины. Также можно настроить работу с разными виртуальными машинами для многослойной защиты. VMProtect поддерживает такие типы обфускации, как дублирование кода, добавление «мертвого» кода и неявных предикатов.

2.4. Выводы

Таким образом, существующие обфускаторы имеют ряд недостатков, которые не позволяют использовать их для данной работы. OLLVM, хотя и предоставляет мощные инструменты для обфускации

потока управления и содержит базовые механизмы защиты, подвержен реверс-инжинирингу, не поддерживает современные версии LLVM и не предоставляет функциональности для обфускации данных. Tigress, с другой стороны, обладает широким набором полезных функций для шифрования переменных и генерации защищённого кода, но ограничен в возможности настраивать собственные арифметические преобразования, что снижает его гибкость.

3. Описание решения

3.1. Алгоритм

Суть реализуемого алгоритма заключается в том, чтобы при обнаружении глобальной или локальной статической переменной создать структуру, содержащую коэффициенты квадратного уравнения. Эти коэффициенты, для усложнения анализа, умножаются на случайное число. Важно, что единственным корнем этого уравнения будет исходная переменная.

На уровне IR (Intermediate Representation) LLVM создаётся функция для решения квадратного уравнения, которая по заданным коэффициентам вычисляет единственный корень. При этом значение переменной в исходном коде заменяется на ноль. Во время исполнения программы, сразу после обнуления переменной, вызывается функция-«решатель». Её результат присваивается обрабатываемой переменной.

Листинг 1: Пример кода до обфускации

```
@GLOBAL_ARG1 = dso_local global i32 42, align 4
```

Листинг 2: Пример кода после обфускации

```
%QuadraticCoefficients = type { i32, i32, i32 }
@GLOBAL_ARG1 = dso_local global i32 0, align 4
@GLOBAL_ARG1_coeffs =
    private constant %QuadraticCoefficients
        { i32 88, i32 -7392, i32 155232 }

define private i32 @solveQuadratic(ptr %0) {
entry:
    .......
    ret i32 %result
}

call i32 @solveQuadratic(%QuadraticCoefficients)
```

```
store i32 %result, ptr @GLOBAL_ARG1
```

Такой подход позволяет скрыть исходное значение переменной и добавить в программу дополнительную логику, не предусмотренную исходным кодом, что должно усложнить реверс-инжиниринг.

3.2. Реализация

Логика решения поставленной задачи начинается с непосредственного поиска глобальных переменных модуля и локальных статических переменных функций. В IR LLVM эти переменные используют идентификатор *global* и всегда имеют тип указателя. Будем проходить только по тем глобальным переменным, у которых есть инициализатор и которые имеют тип Int32. Затем создадим структуру, которая хранит три целочисленных значения: {randomInt; -2 * constValue * randomInt; constValue * constValue * randomInt}. Таким образом, у уравнения, составленного с этими коэффициентами будет ровно один корень — искомая константа. Также создадим функцию, решающую уравнение по заданным коэффициентам:

Листинг 3: Функция, решающая уравнение по заданным коэффициентам в IR LLVM

```
Function *EncodeGlobalConst::createSolverFunction
(Module &M, Type *returnType, StructType *coeffStructType,
   StringRef funcName) {
LLVMContext &Context = M.getContext();

FunctionType *funcType = FunctionType
::get(returnType, PointerType::getUnqual(coeffStructType),
   false);
Function *solverFunc = Function
::Create(funcType, Function::PrivateLinkage, funcName, &M);

BasicBlock *BB = BasicBlock::Create(Context, "entry",
```

```
        solverFunc);
    IRBuilder<> builder(BB);

    Value *coeffPtr = solverFunc->getArg(0);
    Value *a = builder.CreateLoad(Type::getInt32Ty(Context),
        builder.CreateStructGEP(coeffStructType, coeffPtr, 0));
    Value *b = builder.CreateLoad(Type::getInt32Ty(Context),
        builder.CreateStructGEP(coeffStructType, coeffPtr, 1));

    Value *neg_b = builder.CreateNeg(b, "neg_b");
    Value *two_a = builder.CreateMul(
    ConstantInt::get(Type::getInt32Ty(Context), 2),
    a, "two_a");

    Value *result = builder.CreateSDiv(neg_b, two_a, "result");

    builder.CreateRet(result);

    return solverFunc;
}
```

Теперь осталось только обнулить значение обрабатываемой переменной и сохранить результат вызова функции-«решателя» в некую локальную переменную и сохранить ее значение в глобальную. В коде LLVM IR это выглядит следующим образом:

Листинг 4: Присваивание результата функции глобальной переменной

```
    BasicBlock &entryBB = F.getEntryBlock();
    IRBuilder<> builder(&entryBB, entryBB.begin());
    Value *result = builder.CreateCall(solverFunc, {globalCoeffs
        });
    builder.CreateStore(result, G);
```

4. Эксперимент

В этой главе будет рассмотрено, как дизассемблер справляется с задачей реверс инжиниринга обфусцированного кода, а также будет замерено время исполнения программы до и после обфускации.

4.1. Декомпиляция обфусцированного кода в IDA

Для проверки результатов работы по обфускации переменных использовался дизассемблер IDA [2]. В качестве минимальной тестовой программы рассмотрим такой код:

Листинг 5: Исходный код тестовой программы

```
#include <stdio.h>
int GLOBAL_ARG1 = 42;
int main() {
    static int LOCAL_ARG1 = 2;
    printf("%d\n", GLOBAL_ARG1);
    printf("%d", LOCAL_ARG1);
    return 0;
}
```

После декомпиляции был получен следующий код:

Листинг 6: Результат декомпиляции обфусцированного кода

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    __int64 v3; // rdx

    main_LOCAL_ARG1 = sub_4011CC(&unk_402020, argv, envp);
    GLOBAL_ARG1 = sub_4011CC(&unk_402020, argv, v3);
    printf("%d\n", (unsigned int)GLOBAL_ARG1);
    printf("%d", (unsigned int)main_LOCAL_ARG1);
    return 0;
}
```

Листинг 7: Результат декомпиляции оригинального кода

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    printf("%d\n", (unsigned int)GLOBAL_ARG1);
    printf("%d", (unsigned int)main_LOCAL_ARG1);
    return 0;
}
```

В обфусцированном коде можно заметить вызовы функций `sub_4011CC`, которые заменяют прямое обращение к глобальным переменным:

Листинг 8: Дополнительная функция sub_4011CC

```
int64 __fastcall sub_4011C0(_DWORD *a1)
{
    return (unsigned int)(-a1[1] / (2 * *a1));
}
```

Таким образом, дизассемблер справился со своей задачей и правильно восстановил исходный код с учетом внесенных модификаций.

4.2. Замер времени исполнения программы

Проверим теперь, как изменяется время исполнения программы после обфускации. Для замеров используем linpack, предложенный для бенчмаркинга в «Софткоме».

Оригинальный и обфусцированный linpack запускается 50 раз. По результатам замеров оказалось, что данные не обладают нормальным распределением, значит считать среднее арифметическое непоказательно. Рассмотрим боксплоты, иллюстрирующие разброс значений времени исполнения и медианы для обоих случаев. Медиана увеличилась всего на 2.55%, что вполне сравнимо с допустимой погрешностью измерений. Можно предположить, что подобный результат вызван тем, что процессор ожидает данные из памяти, что нивелирует влияние дополнительных вычислительных издержек, которые могли бы возникнуть вследствие обфускации. Этот эффект хорошо согласуется с малым уве-

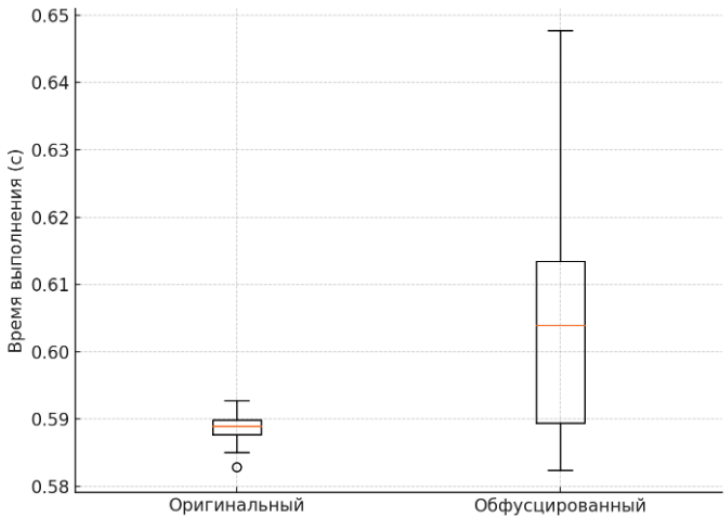

Рис. 1: Сравнение времени исполнения оригинального и обфусцированного кода

личением времени выполнения, поскольку время простоя процессора, вызванное задержками при доступе к данным, становится основным ограничивающим фактором производительности. Однако статистика по промахам кэша, полученная с помощью perf, показывает, что количество промахов кэша после обфускации почти не изменилось. Это говорит о том, что гипотеза о том, что процессор занят ожиданием данных из памяти неверна. Малое увеличение времени исполнения может быть вызвано другими факторами.

4.3. Результаты

В ходе эксперимента было определено, что средства обратной разработки справляются с декомпиляцией обфусцированного кода и вместо прямого доступа к переменным создают структуру коэффициентов и вызывают дополнительную функцию для подсчета переменных, что не скрывает их значения, но усложняет логику доступа к ним. Измерение времени исполнения обфусцированного и оригинального файла не позволило сделать статистически значимых выводов о влиянии обфускации на производительность.

Заключение

Код решенной задачи находится в закрытом репозитории «Софткома».

В процессе выполнения работы был выбран алгоритм обфускации данных, основанный на создании структуры коэффициентов квадратного уравнения так, чтобы у данного уравнения был единственный корень, соответствующий искомой переменной. Затем вызывается функция-решатель, и результат её выполнения присваивается предварительно обнулённой переменной.

Был реализовал плагин, работающий с глобальными переменными по описанному алгоритму.

Эксперимент с использованием инструмента обратной разработки IDA показал, что доступ к переменной действительно затруднен вызовом дополнительной функции.

Список литературы

[1] Data Obfuscation Explained. — 2023. — URL: https://www.crowdstrike.com/cybersecurity-101/data-obfuscation/ (дата обращения: 16 сентября 2024 г.).

[2] IDA Free. — URL: https://hex-rays.com/ida-free/ (дата обращения: 16 сентября 2024 г.).

[3] Iollvm: enhance version of ollvm / Chengyang Li, Tianbo Huang, Xiarun Chen et al. // arXiv preprint arXiv:2203.03169. — 2022. — URL: https://arxiv.org/pdf/2203.03169 (дата обращения: 15 сентября 2024 г.).

[4] OLLVM Features. — 2019. — URL: https://github.com/obfuscator-llvm/obfuscator/wiki/Features (дата обращения: 15 сентября 2024 г.).

[5] VMProtect 3: Virtualization-Based Software Obfuscation Pt. 2. — 2021. — URL: https://www.mitchellzakocs.com/blog/vmprotect3#strengths-and-weaknesses (дата обращения: 15 сентября 2024 г.).

[6] Writing an LLVM Pass. — URL: https://llvm.org/docs/WritingAnLLVMNewPMPass.html (дата обращения: 15 сентября 2024 г.).

[7] The tigress c obfuscator. — URL: https://tigress.wtf/index.html (дата обращения: 15 сентября 2024 г.).