# Статья Разработка вредоносного ПО. Часть 6 - расширенная обфускация с помощью LLVM и метапрограммирование шаблонов



**Введение**

Это шестой пост в серии, посвященной разработке вредоносного ПО. В этой серии статей мы исследуем и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, укрытия от защиты и персистентности.

Сегодня мы рассмотрим анти-дизассемблерную обфускацию с использованием LLVM и метапрограммирование шаблонов.

**Обфускация LLVM**

LLVM - это инфраструктура компилятора. Чтобы понять, что это такое, нам нужно погрузиться в процесс компиляции (это наиболее точно для неуправляемого кода, такого как C/C++).

Мы можем выделить три этапа создания сборки из исходного кода:

**1. Фронт-энд, в который входят:**

-сканер, который выполняет лексический анализ кода и производит токены (строки с определенным значением)

-парсер, который создает абстрактное синтаксическое дерево (токены, сгруппированные в дерево, которое представляет фактический алгоритм, реализованный в исходном коде)

-семантический анализ (в основном проверка типов), во время которого AST проверяется на наличие ошибок, таких как неправильное использование типов или использование переменных перед инициализацией

- генерация промежуточного представления, обычно на основе AST

**2. Оптимизация, которая направлена на снижение сложности кода, например, путем предварительного расчета.**

Оптимизация не должна изменять сам алгоритм/программу.

**3. Серверная часть, которая переводит промежуточное представление в ожидаемый результат (ассемблер или байт-код).**

Ядром LLVM является оптимизатор, но проект также включает интерфейс компилятора - clang - который предназначен для использования с цепочкой инструментов LLVM.

**Обфускатор-LLVM**

Мы будем использовать проект Obfuscator-LLVM, который является форком LLVM с открытым исходным кодом.

Обфускация работает на упомянутом промежуточном уровне представления (IR). Другими словами, это своего рода «антиоптимизация». Clang используется для генерации IR из исходного кода, затем IR обрабатывается, чтобы скрыть поток кода, и, наконец, создается ассемблерный код.

**Настройка**

Пройдя теоретическое введение, давайте подготовим среду для обфускации кода C++. Необходимо загрузить и скомпилировать Obfuscator-LLVM. Последняя ветка - llvm-4.0 (с 2017 года последняя версия LLVM в настоящее время - 11.0), и код необходимо компилировать с помощью Visual Studio 2017, а не 2019 (так как при компиляции возникают некоторые ошибки). Нам нужно использовать CMake для создания проекта VS2017, а затем его скомпилировать (учитывая целевую архитектуру). Мы можем использовать командную строку разработчика для VS 2017, которая является частью Visual Studio 2017:

Code:

```
git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator
cd obfuscator
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

Примечание: мне пришлось вручную определить ENDIAN_LITTLE, чтобы избавиться от некоторых ошибок компиляции.

Есть разные способы использования компилятора Obfuscator-LLVM:

***- использовать вручную через командную строку***

***- добавить компилятор в качестве настраиваемого инструмента ассемблера для .cpp и других файлов в Visual Studio (в соответствующем файле Property Pages)***

***- используйте VS Installer для установки набора инструментов платформы clang-cl и вручную замените версию clang Visual Studio на скомпилированный компилятор (это вроде как проблема курица-яйцо  ) https://en.wikipedia.org/wiki/Bootstrapping_(compilers)***

**Использование и особенности**

Напишем простую программу, которая выполняет довольно простые вычисления на основе псевдослучайного значения:

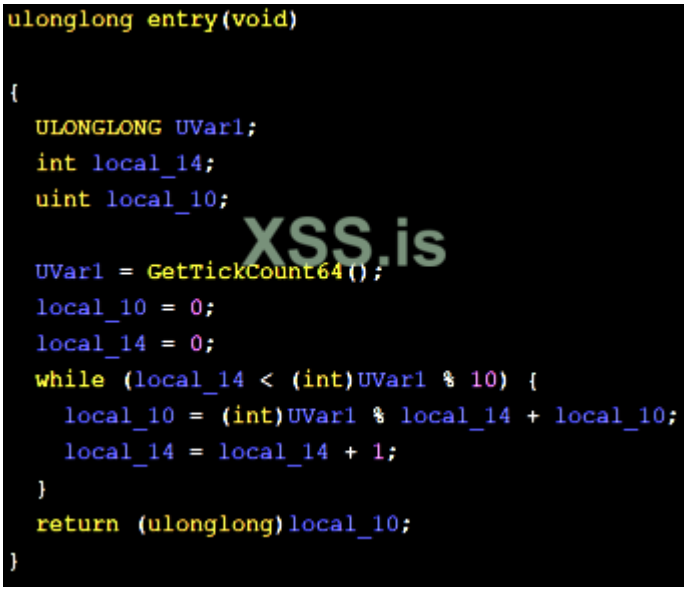
C++:

```
int main()
{
    int a = GetTickCount64();
    int b = a % 10;
    int c = 0;
    for (int i = 0; i < b; i++)
    {
        c += a % i;
    }
    return c;
}
```

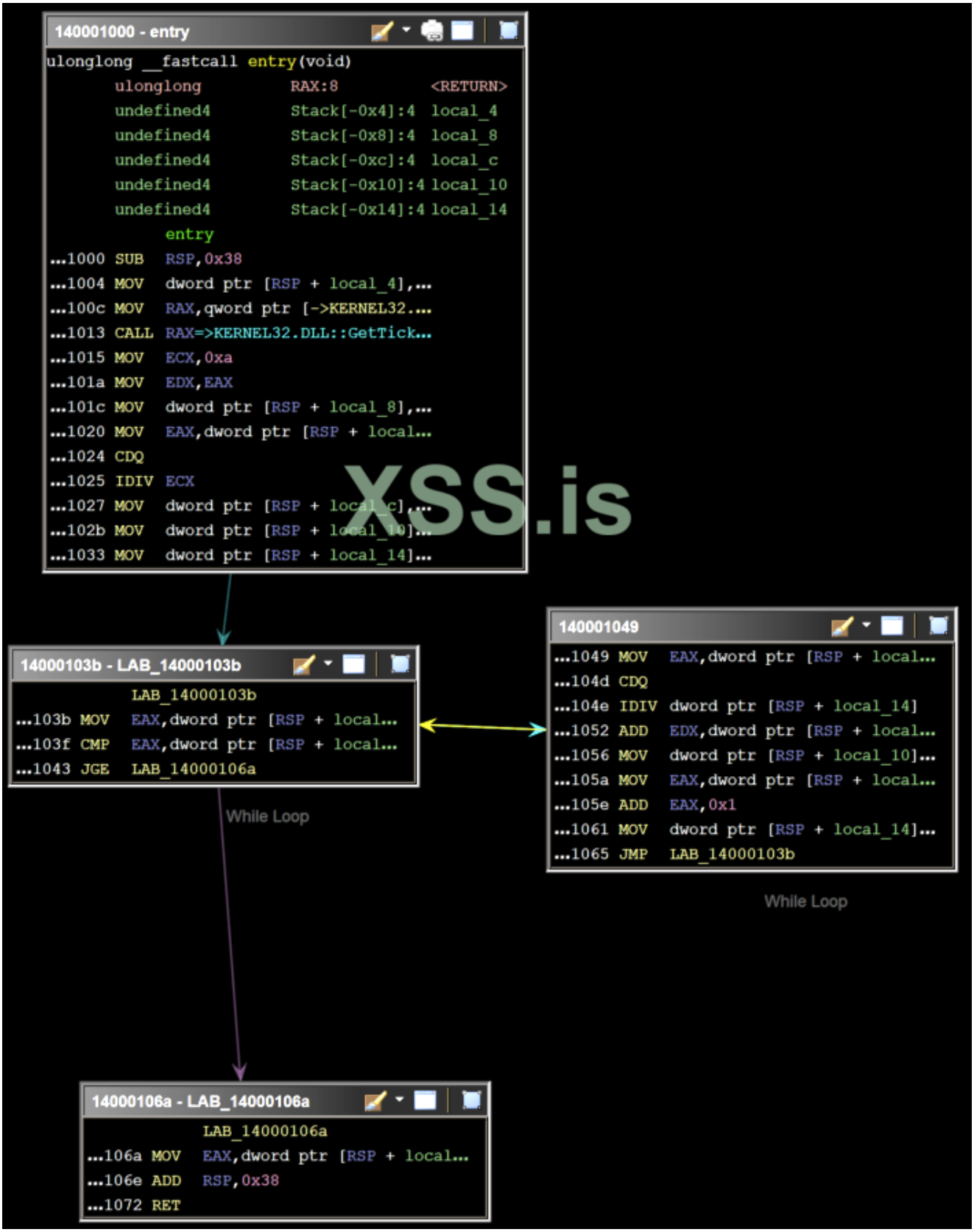
Примечание. Я скомпилировал этот код без зависимости от CRT, поэтому двоичный файл имеет небольшой размер и не требует дополнительного кода (например, mainCRTStartup и т.д.)

Вот как выглядит код после декомпиляции с помощью Ghidra:

И граф программы:



```
ulonglong entry(void)

{
  ULONGLONG UVar1;
  int local_14;
  uint local_10;

  UVar1 = GetTickCount64();
  local_10 = 0;
  local_14 = 0;
  while (local_14 < (int)UVar1 % 10) {
    local_10 = (int)UVar1 % local_14 + local_10;
    local_14 = local_14 + 1;
  }
  return (ulonglong)local_10;
}
```




Obfuscator-LLVM имеет 3 функции обфускации кода: подстановка инструкций, фиктивный поток управления и сглаживание потока управления. Давайте изучим их. Подробности можно найти в репозитории проекта. https://github.com/obfuscator-

llvm/obfuscator/wiki/Features

Эти функции используют случайное значение, которое должно быть указано в качестве параметра командной строки (-mllvm -aesSeed = 1234567890ABCDEF1234567890ABCDEF) в системах Windows (в Linux используется / dev / random).

**Замена инструкций**

Это заменяет простые арифметические операции более сложными, но эквивалентными. Например: a = b + c можно заменить на r = rand(); a = Ь + r; a = a + c; a = a - r ;. Случайное значение рассчитывается во время компиляции.

Замены можно применять несколько раз. Случайное начальное число из командной строки используется для случайного выбора замещающей последовательности инструкций, что придает дополнительную уникальность результирующему двоичному файлу.

Давайте добавим следующие ключи в командную строку компиляции: -mllvm -sub -mllvm -sub_loop = 5 -mllvm -aesSeed = 1234567890ABCDEF1234567890ABCDEF

Полученный ассемблерный код (декомпилированный):

```
ulonglong entry(void)

{
  ULONGLONG UVar1;
  int local_14;
  uint local_10;

  UVar1 = GetTickCount64();
  local_10 = 0;
  local_14 = 0;
  while (local_14 < (int)UVar1 % 10) {
    local_10 = -((-(local_10 + 0x1ff53be2) - (int)UVar1 % local_14) + 0x1ff53be2);
    local_14 = -0x2eece4c7 - (-1 - (0x662d2b91 - (0x374046ca - local_14)));
  }
  return (ulonglong)local_10;
```



И его граф:

Обратите внимание, что декомпилятор Ghidra довольно хорошо справился с "деоптимизацией" обфускатора.

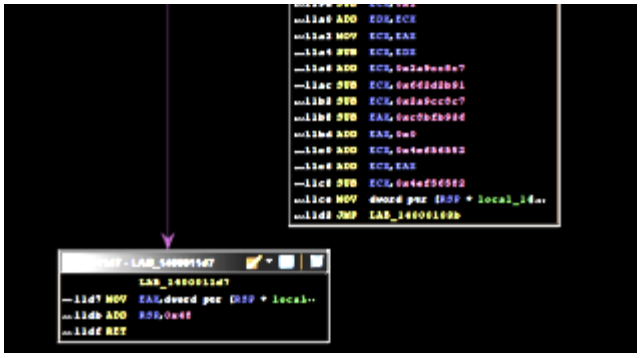
**Поддельный поток управления**

Это добавляет непрозрачные предикаты перед блоками инструкций. Непрозрачный предикат - это в основном часть (предпочтительно случайного) кода, которая оценивается во время выполнения до заранее определенного логического значения (истина или ложь). За ним следует условный переход, который указывает на исходный блок инструкций.

Эта обфускация также может применяться несколько раз и может нацеливаться на случайные блоки кода.

Пример использования: -mllvm -bcf -mllvm -bcf_prob = 100 -mllvm -bcf_loop = 1 -mllvm -aesSeed = 1234567890ABCDEF1234567890ABCDEF

Полученный ассемблерный код:

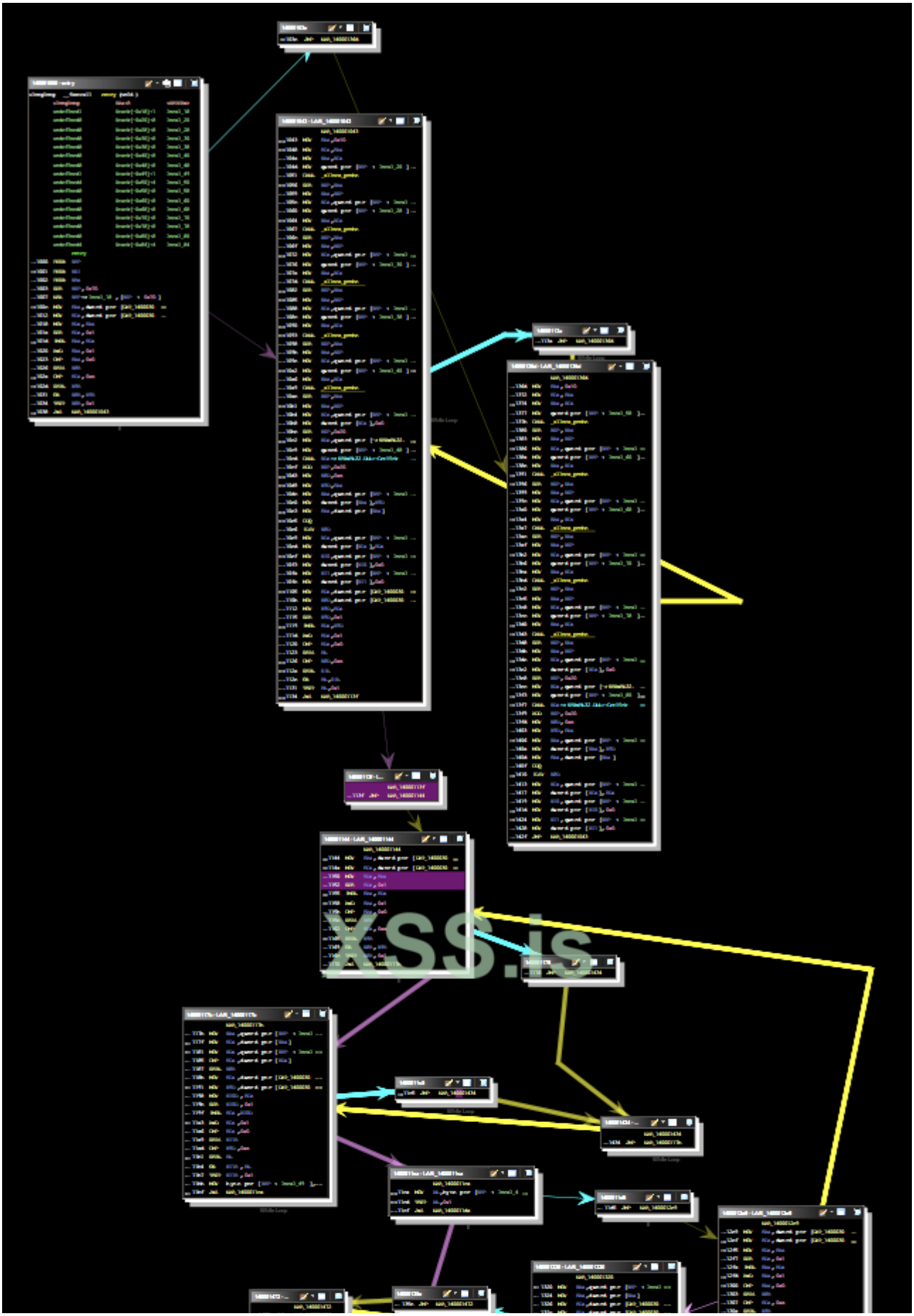


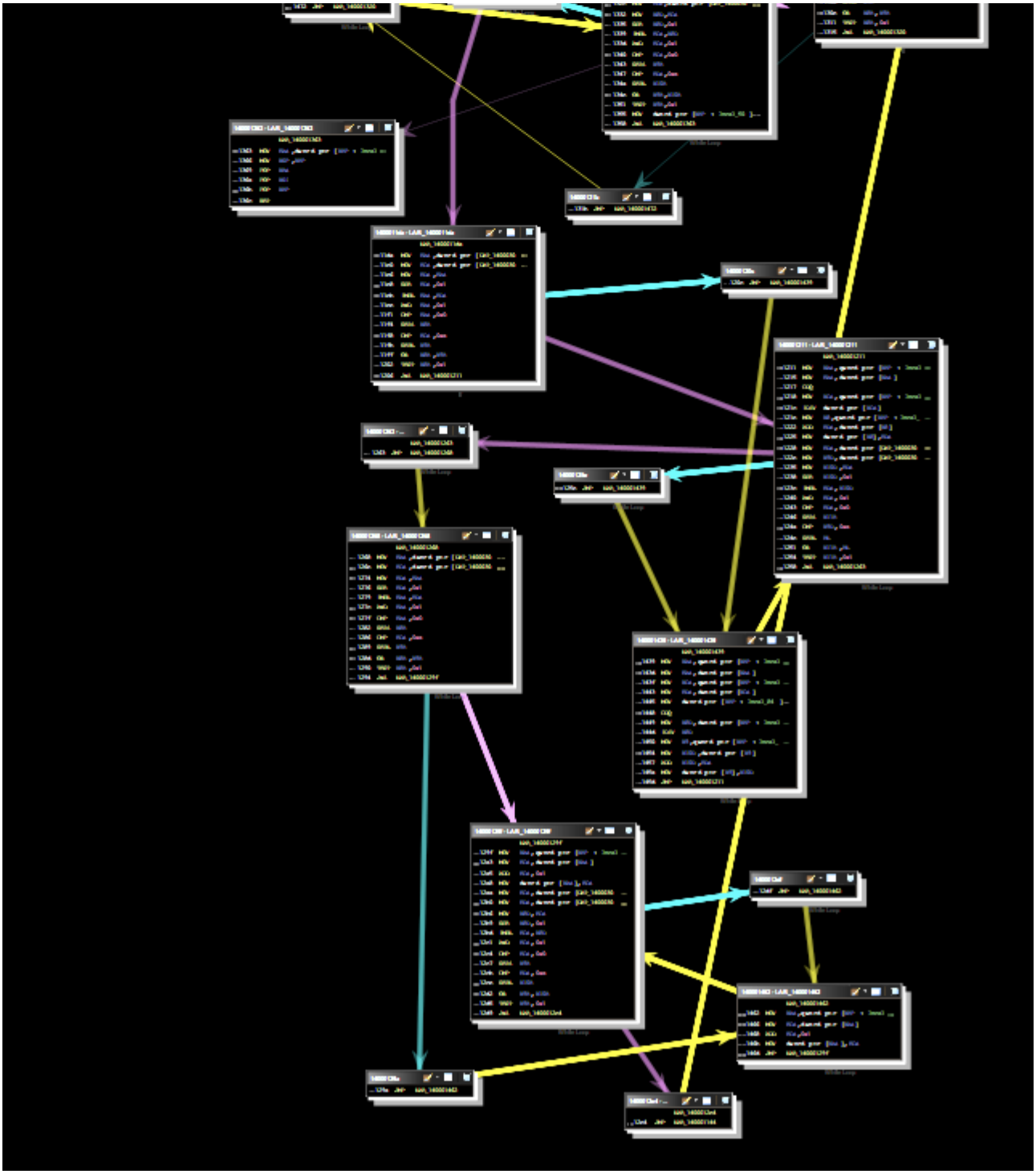
```
ulonglong entry(void)
{
  int *piVar1;
  ULONGLONG UVar2;
  undefined *puVar3;
  undefined *puVar4;
  undefined auStack136 [8];
  undefined4 *local_80;
  undefined4 *local_78;
  int *local_70;
  int *local_68;
  undefined *local_60;
  undefined8 local_58;
  int *local_48;
  uint *local_40;
  int *local_38;
  int *local_30;
  undefined *local_28;
  undefined8 local_20;

  puVar3 = auStack136;
  puVar4 = auStack136;
  if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_1400
  do {
    local_58 = 0x10;
    *(undefined8 *)(puVar4 + -8) = 0x140001380;
    local_60 = puVar4 + -0x20;
    *(undefined8 *)(puVar4 + -0x28) = 0x140001396;
    local_68 = (int *)(puVar4 + -0x40);
    *(undefined8 *)(puVar4 + -0x48) = 0x1400013ac;
    local_70 = (int *)(puVar4 + -0x60);
    *(undefined8 *)(puVar4 + -0x68) = 0x1400013c2;
    local_78 = (undefined4 *)(puVar4 + -0x80);
    *(undefined8 *)(puVar4 + -0x88) = 0x1400013d8;
    puVar3 = puVar4 + -0xa0;
    *(undefined4 *)(puVar4 + -0x20) = 0;
    local_80 = (undefined4 *)(puVar4 + -0xa0);
    *(undefined8 *)(puVar4 + -200) = 0x1400013f9;
    UVar2 = GetTickCount64(puVar4[-200]);
    piVar1 = local_68;
    *local_68 = (int)UVar2;
    *local_70 = *piVar1 % 10;
    *local_78 = 0;
    *local_80 = 0;
```

```
LAB_140001043:
    local_20 = 0x10;
    *(undefined8 *)(puVar3 + -8) = 0x140001056;
    local_28 = puVar3 + -0x20;
    *(undefined8 *)(puVar3 + -0x28) = 0x14000106c;
    local_30 = (int *)(puVar3 + -0x40);
    *(undefined8 *)(puVar3 + -0x48) = 0x140001082;
    local_38 = (int *)(puVar3 + -0x60);
    *(undefined8 *)(puVar3 + -0x68) = 0x140001098;
    local_40 = (uint *)(puVar3 + -0x80);
    *(undefined8 *)(puVar3 + -0x88) = 0x1400010ae;
    puVar4 = puVar3 + -0xa0;
    *(undefined4 *)(puVar3 + -0x20) = 0;
    local_48 = (int *)(puVar3 + -0xa0);
    *(undefined8 *)(puVar3 + -200) = 0x1400010cf;
    UVar2 = GetTickCount64(puVar3[-200]);
    piVar1 = local_30;
    *local_30 = (int)UVar2;
    *local_38 = *piVar1 % 10;
    *local_40 = 0;
    *local_48 = 0;
  } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  while( true ) {
    do {
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
    if (*local_38 <= *local_48) break;
    if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_140001211;
    do {
      *local_40 = *local_40 + *local_30 % *local_48;
LAB_140001211:
      *local_40 = *local_30 % *local_48 + *local_40;
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
    if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_14000129f;
    do {
      *local_48 = *local_48 + 1;
LAB_14000129f:
      *local_48 = *local_48 + 1;
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  }
  do {
  } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  return (ulonglong)*local_40;
}
```



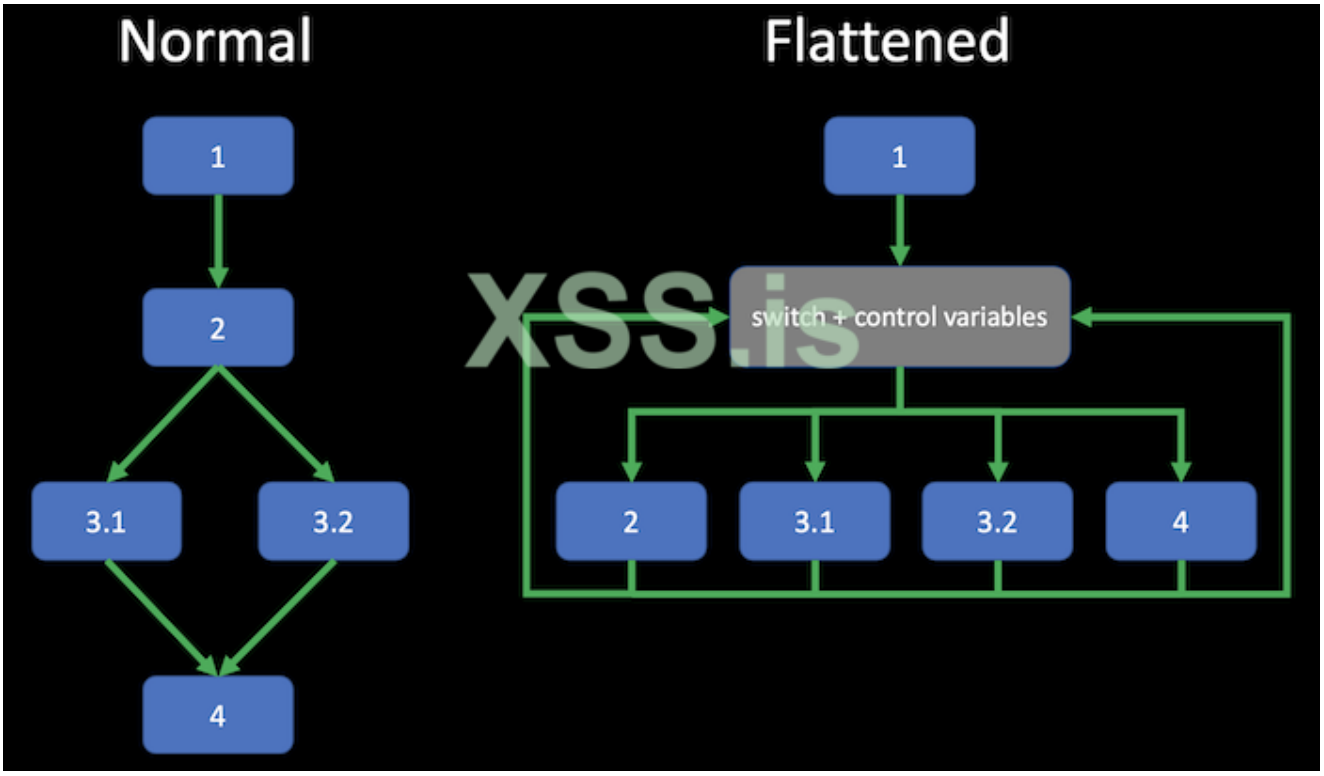
И его граф:





**Сглаживание потока управления**

Это нарушает последовательность блоков инструкций, помещая их на один уровень в зацикленном операторе switch. Определены дополнительные переменные, которые фактически контролируют порядок выполнения. Смотри диаграмму ниже - она должна прояснить ситуацию:




Эту обфускацию также можно применить несколько раз к одному блоку.

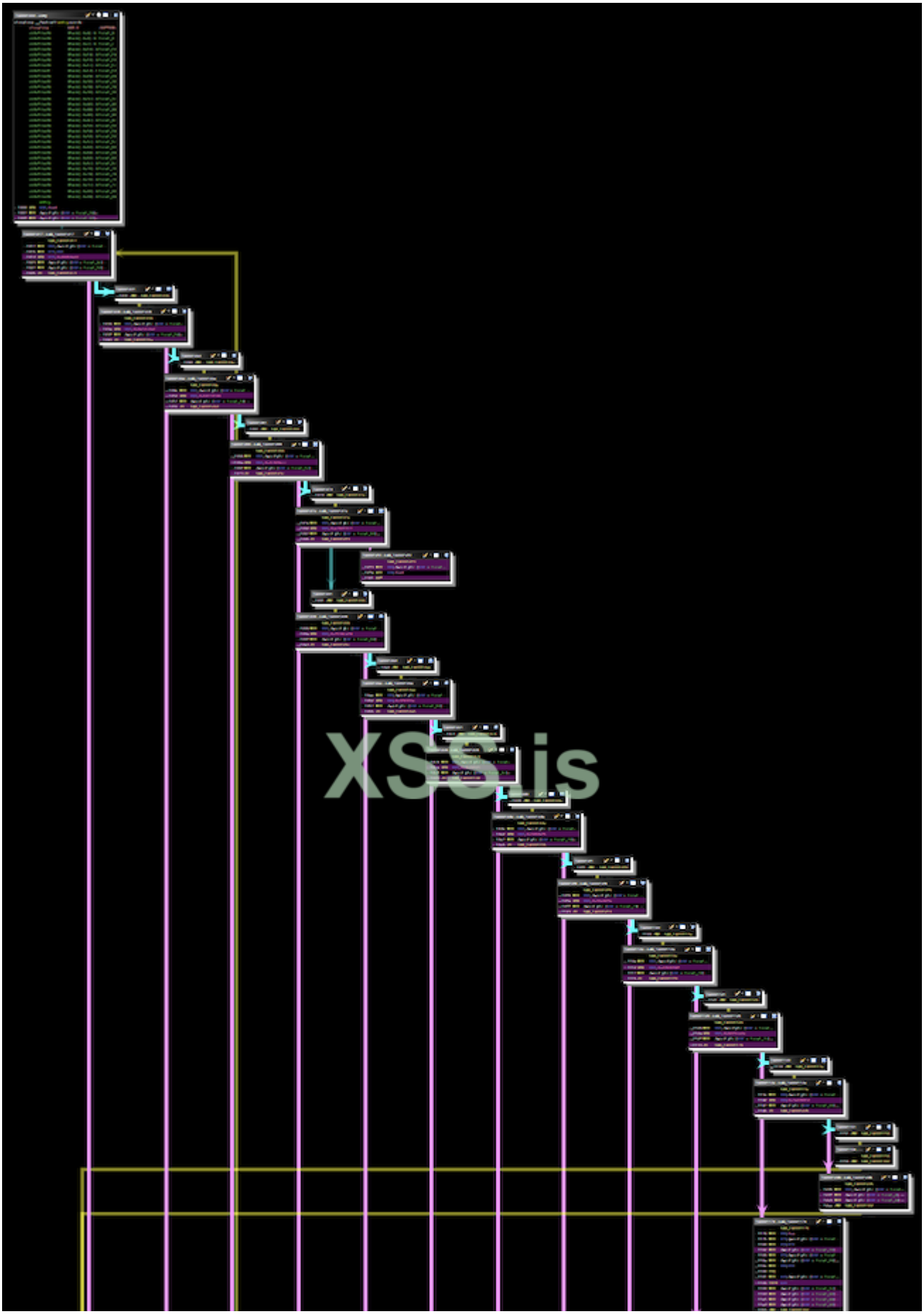
Пример использования: -mllvm -fla -mllvm -split -mllvm -aesSeed = 1234567890ABCDEF1234567890ABCDEF

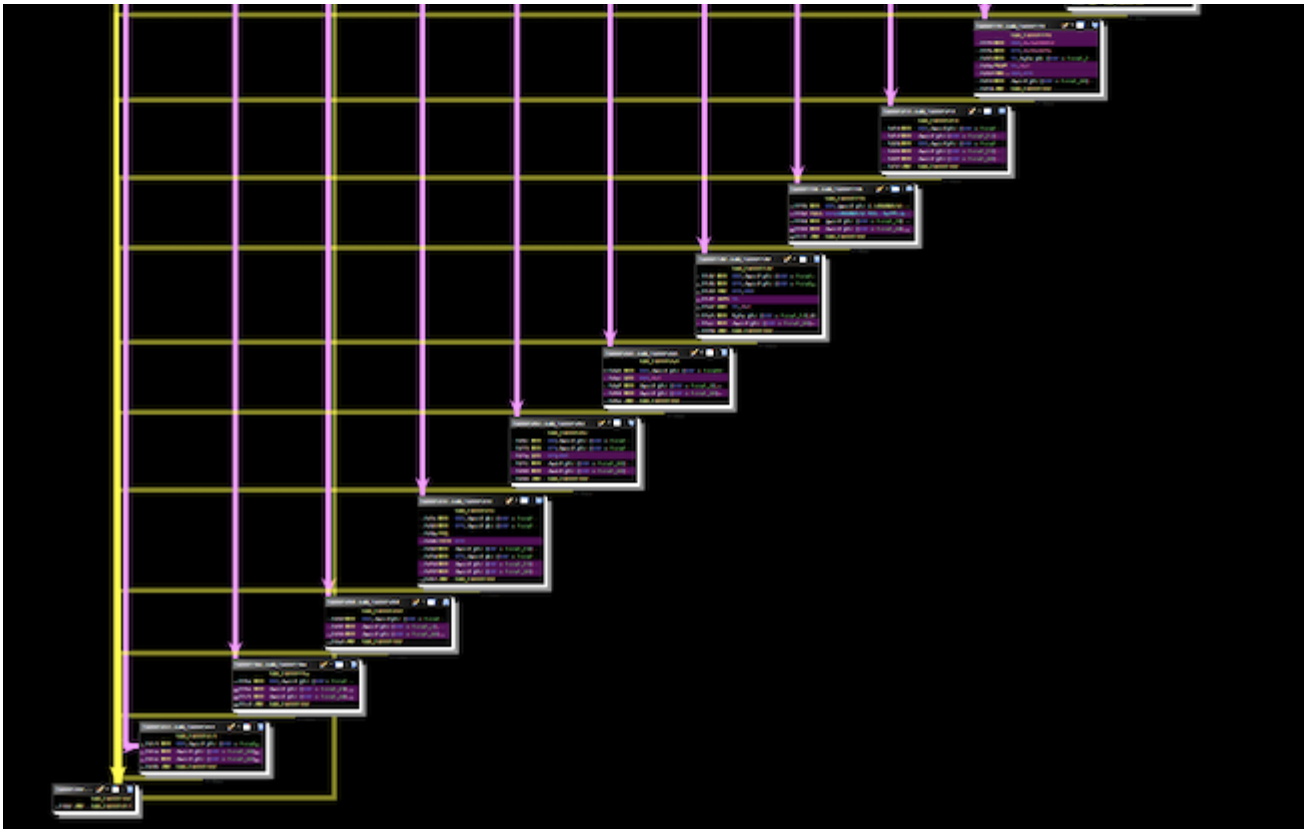
Полученный ассемблерный код (декомпилированный):

```
ulonglong entry(void)
{
  int local_48;
  int local_44;
  int local_40;
  int local_3c;
  int local_38;
  ULONGLONG local_30;
  int local_24;
  bool local_1d;
  int local_1c;
  int local_18;
  int local_14;
  int local_10;
  int local_c;
  int local_8;
  uint local_4;

  local_48 = 0x589dafb;
  while( true ) {
    while( true ) {
      while( true ) {
        while( true ) {
          while (local_48 == -0x797495de) {
            local_44 = local_8;
            local_48 = 0x9a12c2a2;
          }
          if (local_48 != -0x65ed3d5e) breal
          local_24 = local_44;
          local_48 = 0x4e28521;
        }
        if (local_48 != -0x49eefe7a) break;
        local_c = local_44;
        local_48 = 0x3f8355e;
      }
      if (local_48 != -0x236d6134) break;
      local_14 = local_1c % local_18;
      local_10 = local_40;
      local_48 = 0xf534caf4;
    }
    if (local_48 == -0x1872eae9) break;
    if (local_48 == -0xacb350c) {
      local_40 = local_10 + local_14;
      local_48 = 0xb6110186;
    }
    else {
      if (local_48 == 0x3f8355e) {
        local_8 = local_c + 1;
        local_48 = 0x868b6a22;
      }
      else {
        if (local_48 == 0x4e28521) {
          local_1d = local_24 < local_3c;
          local_48 = 0x2d6d294f;
        }
        else {
          if (local_48 == 0x589dafb) {
            local_30 = GetTickCount64();
            local_48 = 0x62d3ce2a;
          }
          else {
            if (local_48 == 0x5be82fa) {
              local_1c = local_38;
              local_18 = local_44;
              local_48 = 0xdc929ecc;
            }
            else {
              if (local_48 == 0x2d6d294f) {
                local_48 = 0x7a2d8912;
                if ((local_1d & 1U) != 0) {
                  local_48 = 0x5be82fa;
                }
              }
              else {
                if (local_48 == 0x62d3ce2a) {
                  local_38 = (int)local_30;
                  local_3c = local_38 % 10;
                  local_40 = 0;
                  local_44 = 0;
                  local_48 = 0x9a12c2a2;
                }
                else {
                  if (local_48 == 0x7a2d8912) {
                    local_4 = local_40;
                    local_48 = 0xe78d1517;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  return (ulonglong)local_4;
}
```



И его граф:


XSS.is



**Тестирование**

Теперь давайте скомпилируем и обфусцируем несколько простых вредоносных программ. Помните простейший инжектор шелл-кода из первой части серии? Обфускация LLVM не поможет, потому что наиболее очевидные индикаторы (шелл-код и импорт) останутся неизменными.

Вот почему мы протестируем другой код - например, эту классическую обратную оболочку https://github.com/sh3llc0d3r1337/windows_reverse_shell_1/blob/master/ReverseShell.cpp. На самом деле здесь используется тот же метод, что и в шеллкоде shell_reverse_tcp (создать IP-сокет и создать процесс cmd со стандартными потоками, подключенными к сокету).

Интересно, что при загрузке скомпилированных двоичных файлов в VirusTotal было обнаружено только одно обнаружение кода, скомпилированного без обфускации, и 6 обнаружений при применении нескольких методов обфускации.

**Заключение**

Obfuscator-LLVM - отличный ресурс для изучения и понимания того, что на самом деле происходит во время компиляции кода и как можно изменить этот процесс, чтобы сделать статический анализ ассемблерного кода более сложным и трудоемким.

Однако важно помнить, что обфускация на уровне IR может быть отменена (не полностью, но все же).Смотри Эту замечательную статью для примера процесса деобфускации https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html .

Вот некоторые общие мысли и соображения: С точки зрения пентестера важно объединить несколько уровней мер защиты кода, чтобы минимизировать шансы обнаружения и максимально затруднить ручной анализ (ну, при разумном объеме наших усилий). Это помогает обеспечить эффективную эмуляцию противника, ориентированную на реальные цели. Конечно, более продвинутое вредоносное ПО требует большей работы со стороны защитных групп, что тоже хорошо.

В любом случае, не забудьте рассмотреть возможность внедрения некоторой обфускации на промежуточном уровне представления в процесс ассемблирования наступательного инструментария.

**Другие обфускаторы на основе LLVM**

Также не забудьте проверить другие обфускаторы на основе LLVM и статьи о создании пользовательских обфускаторов с помощью LLVM:

https://github.com/HikariObfuscator/Hikari/

https://medium.com/@polarply/build-your-first-llvm-obfuscator-80d16583392b

http://www.babush.me/dumbo-llvm-based-dumb-obfuscator.html

https://github.com/emc2314/YANSOllvm

https://blog.scrt.ch/2020/06/19/engineering-antivirus-evasion/

https://blog.scrt.ch/2020/07/15/engineering-antivirus-evasion-part-ii/

**Метапрограммирование шаблона**

Прежде чем углубляться в детали конструкций C++, таких как шаблоны, константные выражения и метапрограммирование, давайте рассмотрим простой случай: у нас есть исходный код с некоторыми строковыми литералами (например, IP-адресами, доменными именами и т.д.), которые необходимо скрыть, чтобы они были невидимы в ассемблерном коде и раскрывались только во время выполнения. Здесь проще всего зашифровать эти литералы и заменить их вызовом процедуры дешифрования, например:

C++:

```
const char* address = "www.example.com";
```

заменить на:

C++:

```
char* Decrypt(const char* data);
(...)
char* addr = Decrypt("xxx.yyyyyyy.zzz");
```

Конечно, нам придется учитывать длину строки, терминаторы нулевого байта и т.д.

Мы бы предпочли использовать значения открытого текста в исходном коде и автоматически скрывать/шифровать их в процессе ассемблирования. Замену простых строк на зашифрованные можно автоматизировать с помощью задачи предварительного ассемблирования, например какой-нибудь скрипт на Python. Но есть другой, более крутой способ сделать это.

**Введение**

Давайте познакомимся с некоторыми функциями, представленными в стандарте C++ 11: шаблонами и выражениями. Следующее не охватывает всех деталей концепций метапрограммирования - это всего лишь простое введение, которое поможет понять, как на самом деле работает обфускация на основе метапрограммирования шаблонов.

**Шаблоны**

Шаблоны - это функции, которые работают с универсальными типами. Шаблоны позволяют просто создавать функции, которые работают с несколькими типами (базовыми типами, структурами, классами). Например, мы можем использовать следующий шаблон:

C++:

```
template <typename T>
bool Equal(T arg1, T arg2)
{
    return (arg1 == arg2)
}
```

вместо определения перегруженных функций:

C++:

```
bool Equal(int arg1, int arg2);
bool Equal(double arg1, double arg2);
```

И пример использования шаблона:

C++:

```
Equal <int>(1, 2));
```

Конечно, типы должны реализовывать оператор ==, чтобы использовать шаблон функции Equal.

Шаблоны также можно использовать для создания общей структуры или класса, которые затем могут быть созданы для использования с определенным типом:

C++:

```
template <typename T>
struct Stack
{
    void push(T* object);
    T* pop();
};

Stack<Fruit> fruitStack;
Stack<Vegetable> vegetableStack;
```

Это также обеспечивает безопасность типов, в этом случае вы не сможете смешивать фрукты с овощами - fruitStack.push (new Vegetable ()); выдаст ошибку компиляции.

Рассмотрим другой пример - использование шаблона для рекурсивного факториального вычисления:

C++:

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
    enum { value = 1 };
};

Factorial<5>::value // 5! = 120
```

Здесь мы видим, что целое число может быть аргументом шаблона и что специализация шаблона (template <>) необходима для определения значения для конкретного аргумента.

**Постоянные выражения**

Спецификатор constexpr указывает, что значение некоторого выражения может быть оценено во время компиляции. Например, когда такое постоянное выражение определено:

C++:

```
constexpr int sum(int a, int b)
{
    return (a + b);
}
```

Sun(1 + 2) будет предварительно вычислена во время компиляции - это вычисление не потребляет ресурсы во время выполнения приложения.

**Метапрограммирование**

Метапрограммирование - это просто изменение программ другими программами или самими собой. Оказывается, шаблоны являются своего рода функциональным языком программирования и могут использоваться компилятором для генерации исходного кода.

Помните? Это именно то, что мы делали со сценариями предварительной сборки - создавали временный исходный код с обфусцированными конфиденциальными данными.

**Обфускация строк**

Поняв способность писать код, который может выполняться компиляторами, давайте создадим простой обфускатор строк, который заменит данные в виде открытого текста значениями, обработанными XOR, непосредственно перед компиляцией. Мы хотели бы использовать обфускацию следующим образом: Obfuscated ("secret") ;. Обфусцированный макрос должен заменить "secret" на функцию дешифрования с зашифрованным аргументом: Decrypt_runtime(Encrypt_compiletime (secret)).

Чтобы использовать постоянную строку во время компиляции, нам нужно знать ее точную длину. Поэтому нам понадобится функция времени компиляции, которая работает с этим значением длины. Итак, сначала нам нужно создать шаблон, который получит целое число в качестве аргумента: template <unsigned int N>.

Теперь мы создадим структуру, которая содержит обфусцированную строку (которая заменит открытый текст в исходном коде) и имеет функцию времени компиляции (constexpr) в качестве конструктора для обфускации открытого текста:

C++:

```
struct Obfuscator
{
    char data[N] = { 0 };
    constexpr Obfuscator(const char* plaintext)
    {
        for (int i = 0; i < N; i++)
        {
            data[i] = plaintext[i] ^ 0x00;
        }
    }
}
```

Теперь мы обфусцируем данные в исходном коде, создавая структуру Obfuscator<7> из шаблона Obfuscator<N> (7 = длина строки + нулевой байт):

C++:

```
constexpr Obfuscator<7> obfuscated = Obfuscator<7>("secret");
```

Чтобы фактически использовать данные в приложении, нам нужно их расшифровать, поэтому мы добавляем функцию деобфускации (которая работает с постоянным значением, следовательно, с идентификатором const после его объявления) в шаблон Obfuscator:

C++:

```
const char* Deobfuscate() const
{
    char plaintext[N] = { 0 };
    for (int i = 0; i < N; i++)
    {
        plaintext[i] = data[i] ^ 0x11;
    }
    return plaintext;
}
```

Теперь мы можем деобфускировать постоянную переменную obfuscated : obfuscated.Deobfuscate ().

Последнее, что нужно сделать, - это создать вспомогательный макрос, который упрощает обфускацию в исходном коде. Воспользуемся еще одним плюсом C++ 11 - лямбда-функциями:

C++:

```
#define Obfuscated(string) []() -> const char* \
{ \
    constexpr auto secret = Obfuscator<sizeof(string) / sizeof(string[0])>(string); \
    return secret.Deobfuscate(); \
}()
```

Благодаря этому строковые литералы, появляющиеся в двоичном файле, зашифрованы XOR. Можно улучшить этот метод, чтобы приложение создавало строки на основе стека, которые не отображаются в разделе .text PE-файла.

**Другие возможности**

С помощью метапрограммирования шаблонов можно реализовать довольно продвинутую обфускацию строк и кода. Для более подробного объяснения смотри этот замечательный воркпепер Себастьяна Андриве. https://www.blackhat.com/docs/eu-14...amming-Applied-To-software-Obfuscation-wp.pdf и его инструмент ADVobfuscator https://github.com/andrivet/ADVobfuscator , который реализует описанные концепции. Доступно несколько таких обфускаторов, и самое лучшее в них то, что мы можем использовать их, просто добавляя файлы заголовков в проект:

https://github.com/fritzone/obfy

https://github.com/revsic/cpp-obfuscator

**Резюме**

Этот пост был просто введением в продвинутые и мощные методы обфускации, которые используют инфраструктуру компилятора LLVM и метапрограммирование шаблонов.

В следующий раз мы поговорим о кейлоггерах и реализуем один из них.

Переведено специально для **XSS.is**
Автор перевода: yashechka
Источник: https://0xpat.github.io/Malware_development_part_6
Last edited: Oct 5, 2021