Статья Метапрограммирование в C++. Или пишем метаморфный движок

Всем привет.
Многие считают что метапрограммирование в Си невозможна, отчасти они правы, но мало кто знает, что в Си есть очень мощный препроцессор, при помощи которого можно делать много вещей.)
Препроцессорные макросы используется обычно для уменьшение рутинных операций и для ускорения исполнения кода, ведь всё-что у нас в препроцессоре, компилятор вставит в код, это почти-как инлаин функция.
Отличие инлаин функций от препроцессорного макроса, в том-что макрос разворачивает Си-код в процессе компиляции, а вместо инлайн функции, компилятор пытается вставить ассемблерный код, в месте вызова такой функции.
Ладно, перейдём к делу...

Зачем это нужно ?

Ну разумеется мы будем использовать для "темных дел".)))

Как и все проекты в этом репозитории, кроме как обучающей цели, можно использовать в двух направлениях:

- Белое направление: Для защиты кода от исследований (Запутывание реверсера).
- Чернуха: Для обхода детекта антивирусов.

Суть идеи:

Думаю все знают, что сигнатурный детект антивируса, не важно эмулятор это, или просто сигнатурный детект на какую-то часть кода, основан на каком-то слепке кода.

Так-вот для обхода детекта, хакеры часто "разбовляют код мусором", мусор этот как-правило представляет собой код разных математических операций, ветвлений и т.д.

Идеально, что-бы после сборки вируса, он был разный, причем разный как на уровне кода, так и на уровне исполнения инструкций.

Для автоматизации этого всего разрабатывают так-называемые "генераторы мусора кода", или ещё можно использовать "обфускацию кода".

Причем эти генераторы могут-быть разные, рассмотрим пару типов (Это классификация моя и не начто не претендует):

1)Генератор мусора, на уровне исполнения программы:

Смысл такой, что в дата-секции мы распологаем например ассеблерные инструкции (опкоды), которые исполняются в момент исполнения программы, обычно в случайном порядке и случайном колличестве.

Также можно исполнять и API-windows в случайном порядке.

Пример такого генератора тут:https://github.com/XShar/simple_trashe_gen_module

Минусы такого решения, что реально секция кода не меняется, ну и реверсер может обнаружить, что код генерируется...

2)Генерация мусора, на уровне компиляции кода:

Смысл заключается в добовлении в исходник, в нужное место мусора, для этого можно использовать так-называемые Junk Code Generator (Пример реализации https://github.com/gehaxelt/PHP-C---JunkCodeGenerator)

Мы-же реализуем такой генератор препроцессором и вставим такой код в нужное место:

Для работы нам будет нужен **Boost.Preprocessor (https://github.com/boostorg/preprocessor)** - это библиотека, делающая метапрограммирование с помощью препроцессора простым, более эффективным и, при правильном пользовании и должном умении, читабельным (тут, конечно же, без ответного мало-мальского умения от читателя такого кода тоже не обойтись).

Данная библиотека хорошо документирована:https://github.com/boostorg/preprocessor/tree/develop/doc/index.html

Вкратце основная идея реализации:

Разместим движок:

1)\metamorph_gen_engine\metamorph_code\boost - Библиотека для работы с препроцессором

2)\metamorph_gen_engine\metamorph_code\morph.h - Реализация функций (мусора, я реализовал метематические операции, т.е. сложение, вычитание, XOR и т.д.), я добавил 14-ть функций для теста, можно добавить больше.

Функции реализованы на макросах, по примеру в документации.)))

3)\metamorph_gen_engine\metamorph_code\config.h - Конфигурационные переменные:

#define START_MORPH_CODE 1 #define END_MORPH_CODE 14

START_MORPH_CODE - Какой номер функции генерировать первой.
END_MORPH_CODE - Какой номер функции генерировать последней.

Т.е. если делать рандомные значение этих макросов (например на этапе сборки, или вручную в этом файле), то можно получать рандомную генерацию функций в коде, в нужном месте.

Как работать с движком (С комментариями что, к чему):

Пример:\metamorph_gen_engine\metamorph_gen_engine\metamorph_gen_engine.cpp:
#define DECL(z, n, text) BOOST_PP_CAT(text, n) ();
BOOST_PP_REPEAT_FROM_TO(START_MORPH_CODE, END_MORPH_CODE, DECL, function)

Пояснения к коду:

BOOST_PP_CAT(text, n) - По сути, это то-же что и cat в Linux (Т.е. на выходе мы получим "textn").

BOOST_PP_REPEAT_FROM_TO - Разворачивает макросы в цикле.

Его прототип:

BOOST_PP_REPEAT_FROM_TO(first, last, macro, data)

first - Начальное значение (У нас START_MORPH_CODE). last - Конечное значение (У нас END_MORPH_CODE).

macro - Макрос, который нужно развернуть, его прототип должен-быть такой:

macro(z, n, data)

В итоге будет разворачиваться так:

macro(z, first, data) macro(z, first + 1, data) ... macro(z, last - 1, data)

Но у нас (наш пример):
Code:

```
#define DECL(z, n, text) BOOST_PP_CAT(text, n) (); BOOST_PP_REPEAT_FROM_TO(START_MORPH_CODE, END_MORPH_CODE, DECL, function)
```

text -> functionN() N -> номер функции

И тогда, после сборки будет так:

```
function1();
function2();
function3();
function4();
function5();
function6();
function7();
function8();
function9();
function10();
function11();
function12();
function13();
function14();
```

Генерация успешна.)))

Исходник движка можно скачать здесь:
https://github.com/XShar/metamorph_gen_engine

автор @X-Shar, ru-sfera