Статья Мета-программирование Nim и обфускация

Копипаст моей статьи, оригинал статьи тут: https://wasm.in/blogs/meta-programmirovanie-nim-i-obfuskacija.706/

Здравствуйте, друзья, мы с вами продолжаем исследовать возможности мета-программирования различных высокоуровневых языков в одной единственной прикладной сфере - обфускации. На этот раз мы рассмотрим язык программирования Nim - модный, молодежный, немного питонный, немного паскальный язык, который позиционируется как альтернатива C++. Насколько дизайн и реализация языка Nim удачны - вопрос спорный, но нас как бы это не особо то и интересует. Нас интересуют только весьма богатые возможности мета-программирования, вшитые в язык.

И так, установим компилятор языка, на момент написания статьи последняя актуальная версия компилятора была 1.2.2. Если вы используете операционные системы семейства Windows, то на официальном сайте есть специальные инсталляторы для вас. Под Linux Nim скорее всего будет доступен в официальных репозиториях вашей операционной системы. Если нет, то его весьма просто собрать из исходников, мануал по сборке есть на сайте и в папке с исходниками. Хочу заметить то, что компилятор языка Nim по сути дела является транспилятором, то есть в качестве результата своей работы генерирует не машинный код, а код на языках программирования C, C++, Objective C или JavaScript. Затем сгенерированные исходные коды подаются на вход соответсвующему компилятору. Если вы на Linux'e, то скорее всего у вас уже установлен GCC, для операционных систем семейства Windows сообщество Nim заботливо предоставляет версии MinGW компиляторов (конечно очень старые, но для разработки на Nim вполне подойдут).

После установки создадим привычный всем хеллоу ворлд, чтобы проверить, что все работает:
Code:

```
echo "Hello World"
```

Ну и конечно создадим скрипт, который мы будет использовать для сборки проекта (для разнообразия в этот раз я буду производить сборку и анализ результатов компиляции на операционной системе семейства Linux, но на других операционных системах все должно происходить аналогично):
Code:

```
nim c -f -r -d:release --nimcache:./Temp -o:test test.nim
```

Команда 'c' означает, что мы хотим собрать исполняемый файл, используя основной для языка программирования Nim бекэнд, а именно C-компилятор. Флаг '-f' указывает компилятору, что нам необходимо каждый раз совершать новую компиляцию всех исходников, без этого флага компилятор Nim не будет повторно компилировать исходники, которые не менялись, но нам же нужно проверять, что результаты компиляции меняются в каждой новой сборке проекта. Флаг '-r' заставляет компилятор не только собрать проект, но и запустить его. Параметром '-d' мы определяем константу компиляции (дефайн в терминологии C/C++ компиляторов), в данном случае с помощью константы 'release' мы говорим компилятору использовать оптимизации при транскомпиляции в C код, а так же передать соответствующий параметр C-компилятору, чтобы тот тоже использовал оптимизации. Параметр '--nimcahce' указывает папку кеша компилятора, то есть куда компилятор будет складывать скомпилированные исходные коды на языке C и объектные файлы. Параметр '-o' указывает имя исполняемого файла, который мы компилируем, а за всеми флагами и параметрами компиляции идет имя основного файла с исходниками (да, Nim достаточно умен, чтобы определить все зависимости проекта сам, нам с вами достаточно просто указать файл, содержащий точку входа).

В языке программирования Nim есть так называемый CTFE (compile-time function evaluation), то есть на этапе компиляции можно использовать достаточно большое подмножество языка (думайте об этом, как о constexpr на стероидах, при этом всегда исполняемый на этапе компиляции). По этой причине мы пойдем по немного другому пути в сравнении с моей аналогичной статьей по C++, но реализуем все алгоритмы, описанные там. Начнем мы с алгоритма генерации уникального зерна компиляции, которое будет использоваться для получения уникальных сборок проекта. Рассмотрим следующий код:
Code:

```
import parseutils
import strutils
import sequtils

proc get_compilation_seed(): uint32 {. compile_time .} =
    let date = CompileDate.split('-').map(parse_uint)
    let time = CompileTime.split(':').map(parse_uint)

    result += uint32(time[2])
    result += uint32(time[1]) * 60
    result += uint32(time[0]) * 60 * 60
    result += uint32(date[2]) * 60 * 60 * 24
    result += uint32(date[1]) * 60 * 60 * 24 * 30
    result += uint32(date[0] - 1970) * 60 * 60 * 24 * 30 * 12

const compilation_seed = get_compilation_seed()

echo compilation_seed
```

В начале кода мы подключаем необходимые библиотеки. Parseutils подтребуется нам для парсинга целых чисел (функция parse_uint), strutils для разбивание строки на токены (функция split), sequtils для применения функции к последовательности в функциональном стиле (с помощью функции map). Дальше мы объявляем функцию get_compilation_seed, которая возвращает нам 32-битное без знаковое число (типа uint32). Для этой функции мы устанавливаем атрибут compile_time, он говорит компилятору о том, что эта функция обязана исполняться на этапе компиляции, а после символа '=' идет тело самой функции. Константы CompileDate и CompileTime являются строками, содержащими дату и время текущей компиляции проекта. Мы разбиваем эти строки на токены функцией split, затем к каждому токену применяем функцию parse_uint, таким образом мы получаем массивы целых чисел, содержащие в себе годы, месяцы, дни, часы, минуты и секунды времени текущей сборки. Затем мы просто складываем эти числа таким образом, чтобы получить что-то типа unix epoch (да-да, я знаю, что это не совсем правильный код, но для наших целей он подойдет). После объявления функции мы объявляем константу compilation_seed, которая будет содержать в себе наше вычисленное зерно текущей компиляции. Ну и выведем это зерно в консоль с помощью процедуры echo. К слову: я пытался сделать аналогичную функцию с помощью модуля times из стандартной библиотеки, который имеет кучу различных функций для манипуляций со временем и датами, но к сожалению при использовании его функций на этапе компиляции возникают ошибки, вероятно это баги, тк я не вижу особых причин, почему этот модуль не может быть использован во время компиляции, но да ладно, такая реализация тоже подойдет.

Теперь давайте убедимся, что все вычисления действительно происходят на этапе компиляции. Для этого зайдем в папку Temp и откроем C-файл, соответствующий нашему модулю (в моем случае модуль назывался test.nim, а соответствующий ему C-файл назывался @mtest.nim.c). Спускаемся вниз исходного текста на языке C и находим функцию NimMainModule, в ней видим, что действительно все вычисления произошли на этапе компиляции:
C:

```
T1_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(1572519577ULL);
echoBinSafe(T1_, 1);
```

Аналогично предыдущей статье, давайте реализуем функцию для хеширования строк на этапе компиляции, рассмотрим следующий код:
Code:

```
proc fnv_hash(str:string):uint32 =
    result = compilation_seed
    for chr in str:
        result = result * 16777619
        result = result xor uint32(chr)

proc h(str:string):uint32 {. compile_time .} =
    return fnv_hash(str)

let str = read_line(stdin)
echo to_hex(fnv_hash(str))
echo to_hex(h"Hello!")
```

Функция fnv_hash - это реализация простого хеширования строк FNV-1, которая может работать как на этапе компиляции (благодаря CTFE), так и на этапе выполнения. Для того, чтобы форсировать ее исполнение на этапе компиляции мы создаем дополнительную функцию h. Благодаря UFCS (unified function call syntax) мы можем получить весьма красивую на мой взгляд конструкцию h"Hello!", которая в нашем проекте будет означать хеш-значение строки "Hello!" на этапе компиляции (по аналогии с плюсовым L"Hello!"). При этом функцию fnv_hash можно использовать для хеширования строк на этапе выполнения кода, и она будет выдавать те же самые значения, что и на этапе компиляции. Проверим, что все действительно работает так, как мы предполагаем, заглянув в сгенерированный C-код:
C:

```
asgnRef((void**) (&str__uiVcSp7VNuojRYnPD6Il7g), readLine__IfmAdseskhTUnfEYpOo5fA(stdin));
nimZeroMem((void*)T1_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T2_ = (NU32)0;
T2_ = fnv_hash__UCE9bfXFRSBtDbPsnj7DaYg(str__uiVcSp7VNuojRYnPD6Il7g);
T1_[0] = toHex__PFvyltn6F6Mr3iXX8ZBpvw(T2_);
echoBinSafe(T1_, 1);
nimZeroMem((void*)T3_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T3_[0] = toHex__PFvyltn6F6Mr3iXX8ZBpvw(((NU32) 1760887554));
echoBinSafe(T3_, 1);
```

Теперь попробуем реализовать генератор псевдо-случайных чисел, работающий на этапе компиляции. Мы могли бы сделать так же, как и в моей аналогичной статье, посвященной C++, но зачем, если в стандартной библиотеке Nim есть добротный ГПСЧ, основанный на алгоритме Вихрь Мерсенна и без проблем работающий на этапе компиляции. Рассмотрим код:
Code:

```
import mersenne

var random_gen {. compile_time .} = new_mersenne_twister(compilation_seed)

proc random(mn:uint32 = 0, mx:uint32 = 0xFFFFFFFE'u32):uint32 {. compile_time .} =
    return mn + random_gen.get_num() mod (mx - mn + 1)

echo random()
echo random()
echo random()
echo random()
```

В начале кода мы подключаем модуль стандартной библиотеки с реализацией алгоритма Вихрь Мерсенна. Далее мы объявляем глобальную изменяемую переменную (ключевое слово var), которая будет работать на этапе компиляции (используя атрибут compile_time). В качестве зерна для алгоритма мы используем наше вычисленное ранее зерно текущей компиляции, таким образом алгоритм будет выдавать разные значения при последующих сборках проекта. Далее для простоты мы сделаем функцию, которая может принимать параметры минимального и максимального значения для псевдо-случайного числа. Кстати, глазастые вы мои, напишите в комментариях, почему значения этого диапазона по-умолчанию заданы как 0-0xFFFFFFFE, а не 0-0xFFFFFFFF? Проверим, что у нас получилось в C-файле:
C:

```
nimZeroMem((void*)T1_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T1_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(80813457ULL);
echoBinSafe(T1_, 1);
nimZeroMem((void*)T2_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T2_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(1583869454ULL);
echoBinSafe(T2_, 1);
nimZeroMem((void*)T3_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T3_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(3312074670ULL);
echoBinSafe(T3_, 1);
nimZeroMem((void*)T4_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T4_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(1194960597ULL);
echoBinSafe(T4_, 1);
```

Переходим к шифрованию строк, но для начала стоит сделать небольшое лирическое отступление. В Nim есть понятие макросов, но это - далеко не те макросы, которые есть в C/C++, они больше похожи на макросы, которые есть в Lisp. Существенное отличие в том, что они работают не просто как текстовые подстановки, а обрабатывают AST (abstract syntax tree - абстрактное синтаксическое дерево). AST по сути является древовидной структурой данных, которая описывает код на языке программирования. А макрос в Nim - это некая функция, которая принимает на вход AST и возвращает AST. Давайте рассмотрим следующий код:
Code:

```
proc xor_string(str:string, key:uint32):string =
    var sk = [
        uint8((key shr 0)  and 0xFF),
        uint8((key shr 8)  and 0xFF),
        uint8((key shr 16) and 0xFF),
        uint8((key shr 24) and 0xFF)
    ]

    result = new_string(str.len)
    for i in str.low .. str.high:
        var chx = uint8(str[i])
        var xxk = sk[i mod sk.len]
        result[i] = chr(chx xor xxk)

        sk[0] = sk[0] + 1
        sk[1] = sk[1] + 2
        sk[2] = sk[2] + 3
        sk[3] = sk[3] + 4

macro e(str:string):untyped =
    let key = random()
    let enc = xor_string($str, key)

    result = quote do:
        xor_string(`enc`, `key`)


echo e"Hello"
```

В начале мы определяем нашу функцию шифрования, на этот раз сделаем ее чуть более сложной, чем простой xor в цикле, только потому, что мы можем, конечная реализация этой функции не имеет существенного значения. Далее мы определяем макрос под названием e, который принимает строку, шифрует ее и генерирует AST для расшифровки строки на этапе выполнения. Тип возвращаемого AST определен как untyped для того, чтобы компилятор проверил соответствие типов после генерации кода. В коде макроса мы генерируем псевдо-случайный ключ для шифрования, сохраняем его в переменную key, затем мы шифруем строку с помощью нашей функции xor_string и сохраняем в переменную enc. Обратите внимание на символ '$' перед переменной str, он указывает, что нам нужно получить непосредственное значение строки (str передается в макрос как NimNode - элемент абстрактного синтаксического дерева, оператор '$' конвертирует его в строку). Затем в качестве результата мы возвращаем AST, полученное с помощью макроса quote. Этот макрос реализует так называемое quasi-quoting (квази цитирование), он определяет блок кода, внутри которого можно производить цитирование значений с помощью ``, затем на этапе компиляции он парсит код внутри блока заменяя цитированные значения и возвращает полученное AST. То есть мы могли бы сгенерировать AST вручную, но

зачем так себя мучить, если доступно квази цитирование. И в результате всего этого мы можем помечать все строки, которые нам нужно зашифровать префиксом 'e', как например e"Hello". Посмотрим, что же на это все нам сгенерировал компилятор:
Code:

```
// В начале файла объявлена зашифрованная строка
STRING_LITERAL(TM__ipcYmBC9bj9a1BW35ABoB1Kw_4, "\377\010\020T\324", 5);

// В функции NimMainModule код для ее расшифровки и вывода
nimZeroMem((void*)T1_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T1_[0] = xor_string__qZXv0dHgqP9c7JNqwosQMQA(((NimStringDesc*)
&TM__ipcYmBC9bj9a1BW35ABoB1Kw_4), ((NU32) 745958327));
echoBinSafe(T1_, 1);
```

Но тут остается один вопрос: что если я хочу, чтобы все строки были зашифрованы, и помечать их все префиксом 'e' - долгое и муторное занятие. И в принципе ответ на эту хотелку есть, но для этого нужно немного похакать компилятор, в том плане, что мы будем для этого использовать фичу, которая изначально для этого не предполагалась. Рассмотрим следующий код:
Code:

```
type estring = distinct string

proc xor_string(str:estring, key:uint32):string =
    return xor_string(string(str), key)

macro encrypt_strings*{str}(str:string{lit}):untyped =
    var key = random()
    var enc = xor_string(estring($str), key)

    result = quote do:
        xor_string(estring(`enc`), `key`)

echo "Hello"
```

Для начала мы определяем новый тип для зашифрованной строки, он определен как distinct, то есть на уровне системы типов языка Nim запрещено автоматическое конвертирование его в тип string и наоборот. Для чего он нужен, я поясню чуть позже. Далее мы определяем новую функцию xor_string, которая использует оригинальную, но только конвертирует параметр из estring в string при вызове. Далее переходим к достаточно сложному моменту - особому типу макросов, которые называются term rewriting macro. Такие макросы автоматически применяются ко всему модулю, самостоятельно находят некоторые шаблоны исходного кода в модуле и заменяют эти шаблоны на сгенерированный макросом код. '*{str}' определяет, что нам нужны все

вхождения шаблона с именем str в модуле, сам шаблон определен как строка с модификатором '{lit}', этот модификатор ограничивает все подмножество строк литералами (то есть чисто формально об этом можно думать, как о константных строковых значениях, определенных прямо в коде, например как "Hello"). Эти term rewriting macro были изначально добавлены в язык с целью упрощения написания собственных оптимизаций, но их толком никто не использовал, а выпилить их из языка нельзя из-за обратной совместимости, так что в принципе их можно спокойно использовать в этом ключе. Проблема с такими макросами в том, что они могут рекурсивно применяться до бесконечности, тут в дело вступает тип estring, который ограничивает макрос всего одним вызовом. Если бы мы не использовали отдельный distinct тип, то после автоматического вызова макроса у нас получилось бы xor_string("<шифр_строка>", <ключ>), затем макрос бы снова применился уже к "<шифр_строка>" и у нас бы вышло xor_string(xor_string("<шифр_строка>", <ключ>), <ключ_2>) и так до бесконечности. Ну точнее не до бесконечности, а до ошибки компиляции. Давайте убедимся, что компилятор сгенерировал то, что мы ожидаем, посмотрев C-файл:
C:

```
// В начале файла объявлена зашифрованная строка
STRING_LITERAL(TM__ipcYmBC9bj9a1BW35ABoB1Kw_4, "O\247\361\006d", 5);

// В функции NimMainModule код для ее расшифровки и вывода
nimZeroMem((void*)T1_, sizeof(tyArray__nHXaesL0DJZHyVS07ARPRA));
T1_[0] = xor_string__YajiDjBvHPaHNQMizuAyEA(((NimStringDesc*)
&TM__ipcYmBC9bj9a1BW35ABoB1Kw_4), ((NU32) 1587003399));
echoBinSafe(T1_, 1);
```

Теперь давайте по фану напишем тоже самое псевдошифрование, или обфускацию данных, но для литералов типа uint32. Просто поксорим все константы uint32 на некое рандомное число на этапе компиляции. Но тут мы столкнемся с небольшой проблемой: компилятор с радостью соптимизирует все наше псевдошифрование. Чтобы этого не произошло, мы используем классический трюк, который я давным давно подсмотрел в одном коммерческом C++ обфускаторе. Рассмотрим следующий код:
Code:

```
type euint32 = distinct uint32

const uint_xor_key_ct = random()
var   uint_xor_key_rt = uint_xor_key_ct

proc decrypt_uint32(val:euint32):uint32 =
    return uint32(val) xor uint_xor_key_rt

macro encrypt_uints*{val}(val:uint32{lit}):untyped =
    var nvl = uint32(val.int_val()) xor uint_xor_key_ct

    result = quote do:
        decrypt_uint32(euint32(`nvl`))

echo 42'u32
```

Объявим distinct тип для типа uint32, как и в прошлый раз, чтобы вовремя остановить рекурсию. Затем нам понадобится два одинаковых значения, одно - константное значение времени компиляции (uint_xor_key_ct), другое объявлено как изменяемое значения времени выполнения (uint_xor_key_rt). Поскольку значение объявлено как изменяемое, то оно может быть изменено в любой момент, в том числи и из другого потока, например. Мы же нигде не собираемся его менять, но тот факт, что оно потенциально может измениться в любой момент, не дает компилятору возможности соптимизировать математические операции с ним, тем самым удалив все наше псевдоширование. Далее весь код аналогичен коду обфускации строк и должен быть уже понятен. Давайте посмотрим, что нам сгенерировал компилятор на это:
Code:

```
// В начале файла объявлена наша изменяемая неизменяемая переменная
N_LIB_PRIVATE NU32 uint_xor_key_rt__7JNztgxwAoubbCNsarXuaQ = ((NU32) IL64(4145949542));

// В функции NimMainModule код для ее расшифровки и вывода
T2_ = decrypt_uint32__GnQqsUXYhfC0wyRLnGnq3Q(((NU32) IL64(4145949516)));
T1_[0] = dollar___RkX9btpg5sQIaP8yYXB6tbA(((NU64) (T2_)));
echoBinSafe(T1_, 1);
```

На всякий случай запустим дизассемблер и убедимся, что C-компилятор тоже не стал удалять нашу обфускацию:
Code:

```
mov  edi,DWORD PTR [rip+0x497e] # 12058 <uint_xor_key_rt__7JNztgxwAoubbCNsarXuaQ>
mov  rax,QWORD PTR fs:0x28
mov  QWORD PTR [rsp+0x8],rax
xor  eax,eax
mov  QWORD PTR [rsp],0x0
xor  edi,0xf71e2b4c
call ca40 <dollar___RkX9btpg5sQIaP8yYXB6tbA>
```

Выглядит неплохо. На этом думаю, что можно закончить. Резюмируя все вышесказанное, стоит заметить, что Nim - куда более развитый язык в плане мета-программирования, чем пресловутый C++. Конечно язык имеет свои недостатки, но в целом мне было весело копаться в нем и писать на нем эту программу. Думаю, что теперь, зная его преимущества и недостатки, я подумаю о практическом применении этого языка для свои проектов. Говоря об обфускации в частности, язык Nim предоставляет API для парсинга и обработки AST как внутри макросов, так и для сторонних программ. То есть можно сравнительно легко написать внешний обфускатор для Nim кода. Надеюсь, что эта статья вам понравилась! Спасибо за внимание.

ЗЫ для тех, кто потерялся в обрывках кода, вот полный код модуля:
Code:

```
import parseutils
import strutils
import sequtils
import mersenne
import macros

proc get_compilation_seed():uint32 {. compile_time .} =
    let date = CompileDate.split('-').map(parse_uint)
    let time = CompileTime.split(':').map(parse_uint)

    result += uint32(time[2])
    result += uint32(time[1]) * 60
    result += uint32(time[0]) * 60 * 60
    result += uint32(date[2]) * 60 * 60 * 24
    result += uint32(date[1]) * 60 * 60 * 24 * 30
    result += uint32(date[0] - 1970) * 60 * 60 * 24 * 30 * 12

const compilation_seed = get_compilation_seed()

proc fnv_hash(str:string):uint32 =
    result = compilation_seed
    for chr in str:
        result = result * 16777619
        result = result xor uint32(chr)

proc h(str:string):uint32 {. compile_time .} =
    return fnv_hash(str)

var random_gen {. compile_time .} = new_mersenne_twister(compilation_seed)

proc random(mn:uint32 = 0, mx:uint32 = 0xFFFFFFFE'u32):uint32 {. compile_time .} =
    return mn + random_gen.get_num() mod (mx - mn + 1)

proc xor_string(str:string, key:uint32):string =
    var sk = [
        uint8((key shr 0)  and 0xFF),
        uint8((key shr 8)  and 0xFF),
        uint8((key shr 16) and 0xFF),
        uint8((key shr 24) and 0xFF)
    ]

    result = new_string(str.len)
    for i in str.low .. str.high:
        var chx = uint8(str[i])
        var xxk = sk[i mod sk.len]
        result[i] = chr(chx xor xxk)

        sk[0] = sk[0] + 1
        sk[1] = sk[1] + 2
        sk[2] = sk[2] + 3
        sk[3] = sk[3] + 4
```

```
macro e(str:string):untyped =
    let key = random()
    let enc = xor_string($str, key)

    result = quote do:
        xor_string(`enc`, `key`)

type estring = distinct string

proc xor_string(str:estring, key:uint32):string =
    return xor_string(string(str), key)

macro encrypt_strings*{str}(str:string{lit}):untyped =
    var key = random()
    var enc = xor_string(estring($str), key)

    result = quote do:
        xor_string(estring(`enc`), `key`)

type euint32 = distinct uint32

const uint_xor_key_ct = random()
var   uint_xor_key_rt = uint_xor_key_ct

proc decrypt_uint32(val:euint32):uint32 =
    return uint32(val) xor uint_xor_key_rt

macro encrypt_uints*{val}(val:uint32{lit}):untyped =
    var nvl = uint32(val.int_val()) xor uint_xor_key_ct

    result = quote do:
        decrypt_uint32(euint32(`nvl`))
```