Санкт-Петербургский политехнический университет Петра Великого
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчёт по практической работе № 5
по предмету «Проектирование ОС и компонентов»

Обфускация кода

Работу выполнил студент гр. 63501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Санкт-Петербург
2016

Содержание

Постановка задачи

В рамках данной работы необходимо ознакомиться с основными средствами и методами защиты исходного кода методом обфускации.

Нужно дать описание базовых методов обфускации исходного кода и практически применить один обфускатор. Продемонстрировать результат его работы.

Введение

Обфускацией программ называется такое эквивалентное преобразование программ, которое придает программе форму, затрудняющую понимание алгоритмов и структур данных, реализуемых программой, и препятствующую извлечению из текста программы определенной секретной информации, содержащейся в ней. Поскольку обфускация программ может найти широкое применение при решении многих задач криптографии и компьютерной безопасности, задаче оценки стойкости обфускации придается очень большое значение, начиная с самых первых работ в этой области. За последние два года написано свыше 70-ти статей по этой теме, что является показателем настоящий гонки между исследовательскими группами.

Давая пользователям доступ к установочным файлам программ, компании неизбежно раскрывают свои профессиональные секреты и наработки, и ничто не останавливает конкурентов от противоправного копирования и воровства чужих алгоритмов. Обратим внимание и на другой пример, это важные обновления (патчи), исправляющие ошибки в операционных системах. Почти мгновенно очередное обновление анализируется хакерами, они выявляют проблему которую это обновление чинит, и атакуют пользователей, не успевших вовремя обновиться, пользователей.

1 История развития обфускации

Обфускацией программы называется всякое ее преобразование, которое сохраняет вычисляемую программой функцию (эквивалентное преобразование), но при этом придает программе такую форму, что извлечение из текста программы (программного кода) ключевой информации об алгоритмах и структурах данных, реализованных в этой программе, становится трудоемкой задачей. Обфускация программ в противоположность реорганизации (рефакторингу) преследует цель затруднить понимание программ и воспрепятствовать целенаправленной их модификации. Поэтому задачу обфускации программ можно счесть одной из задач системного программирования, подобной другим задачам преобразования программ – трансляции, оптимизации, реорганизации, распараллеливания. С другой стороны, обфускацию можно также рассматривать как особую разновидность шифрования программ. В отличие от традиционных видов шифрования обфускация не предполагает построения эффективных алгоритмов расшифрования, т.е. восстановления исходного текста программы, но зато требует сохранения смысла зашифрованного сообщения – функции, вычисляемой обфускируемой программой. Поэтому задача обфускации программ может быть также отнесена к области криптографии и криптоанализа. Именно двойственность этой задачи и объясняет тот факт, что ее исследование вот уже более 15 лет проводится по двум направлениям – со стороны системного программирования и со стороны криптографии, – которые очень мало взаимодействуют друг с другом. При изучении математической проблемы обфускации программ начинать нужно с определения стойкости обфускации. Требования стойкости существенно зависят от тех приложений, в которых используется обфускация. И поэтому в мы рассмотрим и проанализируем определения понятия стойкости обфускации программ.

Вероятно, задача обфускации была впервые упомянута (без явного употребления термина «обфускация») в 1976 году в основополагающей работе Диффи и Хеллмана[1]. Желая проиллюстрировать концепцию шифрования с открытым ключом, они предложили следующую простую схему ее реализации. Выбирается произвольная криптосистема с секретным ключом, в процедуру шифрования вставляется секретный ключ, и затем инициализированная этим ключом программа шифрования запутывается так, чтобы извлечение из ее текста секретного ключа было очень трудной задачей. Таким образом, модифицированная процедура шифрования становится открытым ключом новой криптосистемы. Запутывание процедуры шифрования с целью предотвращения извлечения из ее текста секретного ключа является одним из возможных применений обфускации программ для решения некоторых задач криптографии и компьютерной безопасности.

В явном виде понятие обфускации программ было введено в 1997 году в работе Коллберга,

Томборсона и Лоу[2]. Авторы этой работы рассматривали обфускацию программ, в первую очередь, как средство защиты прав интеллектуальной собственности на алгоритмы, которые реализуются в программах с открытым кодом. Ими были предложены простейшие виды обфускирующих преобразований программ, проведена их систематическая классификация и прослежена взаимосвязь задачи обфускации программ с некоторыми известными задачами системного программирования.

В 2001[3] году впервые было предложено формальное определение: результирующая программа, выдаваемая обфускатором, должна давать не больше информации, чем просто напросто черный ящик, который имитирует входное/выходное поведение исходной программы. То есть не должно быть никакой разницы между обфусцированным кодом программы и, например, веб сервисом, который просто возвращает результат программы на данном ему входе. Такой алгоритм получил название «Обфускация Черного Ящика» («Black Box Obfuscation»). К сожалению, в той же статье было показано что такой обфускатор невозможно построить для всех программ. А именно, есть весьма специфический класс программ, который невозможно обфусцировать: это программы которые на собственном входе возвращают некоторый секрет[3], Theorem 3.4. С тех пор это направление исследований заглохло, люди приуныли и обфускация программ целых 12 лет считалась невозможной.

В 2013[4] году в этой области был совершен прорыв, теоретиками было вытащено на свет другое определение и предложена настоящая конструкция для него. Этот новый вид обфускатора называется «Обфускация Неразличимости» («Indistinguishability Obfuscation» — «iO»), формально: если имеются две разные программы, но с абсолютно идентичными функциональностями, то обфускации этих двух программ будут неотличимы друг от друга. То есть, если имеются программы P_1, P_2, такие что для любого входа x, $P_1(x) = P_2(x)$, а O – это обфускатор неразличимости, который принимает на вход программу P и возвращает новую программу O(P), то невозможно будет отличить $O(P_1)$ и $O(P_2)$. То есть если невозможно сказать, какая обфускация какой изначальной программе принадлежит, то ли $O(P_1)$ – это обфускация P_1, то ли это обфускация P_2 (Обфускатор O — вероятностный алгоритм).

В 2007[5] году был исследован «лучший» обфускатор. Было предложено называть обфускатор «лучшим», если обфусцированная программа сообщает не больше информации, чем любая другая программа с той же функциональностью. И было показано, что Обфускатор Неразличимости – это и есть «лучший» обфускатор. Таким образом была получена конструкция-кандидат лучшего обфускатора. Обфускатор Неразличимости вместе с односторонними функциями (One-Way Functions) вместе дают:

- криптографию публичного ключа (public key encryption)

- короткие цифровые подписи (short signatures)
- не интерактивные доказательства с нулевым разглашением (NIZKs — Non-Interactive Zero Knowledge Proofs)
- забывчивую передачу (Oblivious Transfer)
- протокол конфиденциального вычисления (Multi-party computation protocols)
- протокол вещания (Broadcast encryption)
- оспариваемое шифрование (Deniable encryption) (в этой схеме можно предоставить ложный ключ к шифру, которые расшифрует все посланные вами сообщения во что вам угодно)
- вместе с полностью гомоморфным шифрованием, дают функциональное шифрование (Functional Encryption)

То есть фактически, Обфускатор Неразличимости это примитив, образующий чуть ли не всю криптографию, с помощью которого можно построить практически всё, что мы имеем в криптографии сегодня. Конечно, требуется еще много работы прежде чем обфускатор станет доступен для широкого использования, но фундамент для этого уже заложен.

Спектр задач, для решения которых можно было бы использовать алгоритмы программной обфускации, весьма обширен, и цели применения обфускации могут быть противоположны. Обфускацию можно использовать как для защиты программ от вирусных атак, так и для маскировки компьютерных вирусов. При обфускации программ для нужд криптографии целью маскировки является сокрытие данных (секретного ключа), но не алгоритмов. Но когда методы обфускации применяются для обеспечения компьютерной безопасности, то целью маскировки является сокрытие алгоритмов, но не обрабатываемых данных. Таким образом, “проблема обфускации программ” включает в себя целое семейство задач маскировки программ, для каждой из которых вводятся специальные требования стойкости обфускации.

Известно немало работ, в которых предлагаются различные практические методы обфускации программ. Некоторые из этих методов были реализованы в коммерческих программных продуктах. Однако влияние фундаментальных теоретических результатов на эту ветвь развития программного обеспечения минимально: требования безопасности, исследуемые в контексте криптографических приложений, либо являются слишком сильными, либо неадекватны тем задачам защиты программного обеспечения, которые возникают на практике. Имеется большой разрыв между теоретическими требованиями стойкости обфускации программ и применяемыми на практике методами и средствами решения этой задачи.

Между положительными и отрицательными результатами решения задачи обфускации образовался большой разрыв. Для некоторых строго формализованных определений стойкости обфускации (стойкость в модели виртуального «черного ящика») существуют такие семейства эффективно вычислимых функций, которые не допускают стойкой обфускации. Стойкую обфускацию удалось построить для существенно более слабых требований стойкости и лишь для очень простых функций - точечных функций и близких к ним семейств функций. Несмотря на то, что положительные результаты были обобщены для более широких классов функций, вопрос о (не) возможности эффективной обфускации для общих криптографических протоколов или же для любого значимого класса программ (например, для конечных автоматов) при стандартных криптографических предположениях остается открытым.

Приведенные наблюдения показывают, что проблема обфускации программ – это очень сложная и многогранная задача, для которой вряд ли удастся найти единый универсальный метод решения. Дальнейший прогресс в исследовании этой проблемы позволит сформировать математические основы для создания широкого многообразия формальных концепций и методов обфускации программ в контексте различных приложений. Создание такого математического аппарата следует начать с разработки различных определений стойкости обфускации и исследования взаимосвязи между предложенными определениями и подходящими понятиями и моделями дискретной математики, математической криптографии, теории сложности вычислений. Это поможет нам открыть наиболее важные свойства для всех типов обфускации программ. Располагая спектром различных определений стойкости обфускации, исследователям будет легче понять, какие требования безопасности обеспечивают те или иные обфускирующие преобразования и оценить, насколько эти преобразования удовлетворяют заявленным целям. Многообразие новых формальных определений стойкости может прояснить решение поставленных задач. Наконец, введение новых формальных требований стойкости обфускации программ откроет новые возможности адаптации формальных методов теоретической информатики к решению задач защиты программ.

2 Виды обфускации

Обфускация имеет под собой серьёзные теоретические основания. Важно не только выполнить основную задачу – запутать код, но и при этом не нанести серьёзного урона как скорости, так и функциональности приложения.

Наибольшую популярность в деле обфускации получил метод Колберга. Пусть программа – A, тогда задача сводится к созданию выходного кода A‘ с использованием преобразований $T_1, T_2 \dots T_n$, опирающихся на использование библиотек $L_1, L_2 \dots L_n$. Эффективность процессов оценивается функциями $E_1, E_2 \dots E_n$. Множественность возникает из-за разбиения кода на разные по важности фрагменты, оцениваемые функциями $I_1, I_2 \dots I_n$. Так мы обеспечим максимальную защиту важнейшим участкам, при этом постараюсь не проиграть в быстродействии.

Преобразования бывают четырех видов:

- лексическая обфускация;
- преобразование данных;
- преобразование управления;
- профилактическая обфускация.

2.1 Лексическая обфускация

Наиболее простая, заключается в форматировании кода программы, изменении его структуры, таким образом, чтобы он стал нечитабельным, менее информативным, и трудным для изучения.

Обфускация такого вида включает в себя:

- удаление всех комментариев в коде программы, или изменение их на дезинформирующие
- удаление различных пробелов, отступов которые обычно используют для лучшего визуального восприятия кода программы
- замену имен идентификаторов (имен переменных, массивов, структур, хешей, функций, процедур и т.д.), на произвольные длинные наборы символов, которые трудно воспринимать человеку
- добавление различных лишних (мусорных) операций

- изменение расположения блоков (функций, процедур) программы, таким образом, чтобы это не коим образом не повлияло на ее работоспособность.

Изменение глобальных имён идентификаторов следует производить в каждой единице трансляции (один файл исходного кода), так чтобы они имели одинаковые имена (в противном случае защищаемая программа может стать не функциональной). Также следует учитывать специфические идентификаторы, принятые в том языке программирования, на котором написана защищаемая программа, имена таких идентификаторов, лучше не изменять.

Можно рассмотреть пример кода:

```
int counter;
bool alarm;

for (counter = 0; counter < 100; counter++)
{
  if (counter == 99)
  {
    alarm = true;
  }
}
```

После процесса лексической обфускации будет получено следующее

```
int plf5ojvb; bool jht4hnv; for(plf5ojvb=0; plf5ojvb<100; plf5ojvb++){if
(plf5ojvb==99) jht4hnv=true;)
```

Читать и понимать такой код становится сложнее с числом роста переменных.

Ниже представлен ещё один пример фрагмента исходного кода программы (написанной на Perl), до и после прохождения лексической обфускации.

До лексической обфускации:

```
my $filter;

if (@pod) {
 my ($buffd, $buffer) = File::Temp::tempfile(UNLINK => 1);
 print $buffd "";
 print $buffd @pod        or die "";
 print $buffd
```

```
 close $buffd        or die "";
 @found = $buffer;
$filter = 1;
}
exit;

sub is_tainted {
my $arg = shift;
my $nada = substr($arg, 0, 0);  # zero-length
local $@;  # preserve caller's version
eval { eval "#" };
return length($@) != 0;
}

sub am_taint_checking {
my($k,$v) = each %ENV;
return is_tainted($v);
}
```

После лексической обфускации:

```
sub z109276e1f2 { ( my $z4fe8df46b1 = shift ( @_ ) ) ; ( my
$zf6f94df7a7 = substr ( $z4fe8df46b1 ,
(0x1eb9+ 765-0x21b6) , (0x0849+ 1465-0x0e02) ) ) ; local $@ ;
eval { eval ( (
"" ) ) ; } ; return ( ( length ( $@ ) != (0x26d2+  59-0x270d) ) )
; } my ( $z9e5935eea4 ) ; if ( @z6a703c020a ) { ( my (
$z5a5fa8125d , $zcc158ad3e0 ) =
File::Temp::tempfile ( "" , (0x196a+ 130-0x19eb) ) ) ; print (
$z5a5fa8125d "" ) ; ( print ( $z5a5fa8125d @z6a703c020a
) or die ( ( ( ( "" . $zcc158ad3e0 ) . "\x3a\x20" ) . $! ) ) ) ;
print ( $z5a5fa8125d "" ) ; ( close ( $z5a5fa8125d ) or die ( ( (
( "" ) ) ) ; ( @z8374cc586e = $zcc158ad3e0 ) ; ( $z9e5935eea4 =
(0x1209+ 1039-0x1617) ) ; } exit ; sub z021c43d5f3 { ( my (
$z0f1649f7b5 , $z9e1f91fa38 ) = each ( %ENV ) ) ; return (
z109276e1f2 ( $z9e1f91fa38 ) ) ; }
```

Данная обфускация программного кода, по сравнению с остальными, позволяет сравни-

тельно быстро привести исходный код программы, в нечитабельное состояние. Один из ее недостатков состоит в том, что она эффективна только для осуществления высокоуровневой обфускации. Современные IDE умеют искать использования методов в коде, это очень упрощает анализ обфусцированного кода. Обход такой обфускации вопрос времени.

2.2 Обфускация данных

Такая обфускация связана с трансформацией структур данных. Она считается более сложной, и является наиболее продвинутой и часто используемой. Ее принято делить на три основные группы, которые описаны ниже.

2.2.1 Обфускация хранения

Заключается в трансформации хранилищ данных, а также самих типов данных (например, создание и использование необычных типов данных, изменение представления существующих и т.д.).

К этой группе относится изменение интерпретации данных определенного типа. Как известно сохранение, каких либо данных в хранилищах (переменных, массивах и т.д.) определенного типа (целое число, символ) в процессе работы программы, очень распространенное явление. Например, для перемещения по элементам массива очень часто используют переменную типа "целое число которая выступает в роли индекса. Использование в данном случае переменных иного типа возможно, но это будет не тривиально и может быть менее эффективно. Интерпретация комбинаций разрядов содержащихся в хранилище данных осуществляется в зависимости от его типа. Так, например, можно сказать, что 16-разрядная переменная целого типа содержащая комбинации разрядов 0000000000001100 представляет целое число 12, но это простое соглашение, данные в такой переменной можно интерпретировать по-разному (не обязательно как 12, а, например как 1100 и т.д.).

Статических (неменяющихся) данные преобразуются в процедурные. Большинство программ, в процессе работы, выводят различную информацию, которая чаще всего в коде программы представляется в виде статических данных таких как строки, которые позволяют визуально ориентироваться в ее коде и определять выполняемые операции. Такие строки также желательно предать обфускации, это можно сделать, просто записывая каждый символ строки, используя его ASCII код, например символ "A"можно записать как 16-ричное число "0x41 но такой метод банален. Наиболее эффективный метод, это когда в код программы в процессе осуществления обфусации добавляется функция, генерирующая требуемую строку в соответствии с переданными ей аргументами, после этого строки в

этом коде удаляются, и на их место записывается вызов этой функции с соответствующими аргументами.

Переменные фиксированного диапазона могут быть разделены на две и более переменных. Для этого переменную "V"имеющую тип "x"разделяют на "k"переменных "v1,...,vk"типа "y"то есть "V == v1,...,vk". Потом создается набор функций позволяющих извлекать переменную типа "x"из переменных типа "y"и записывать переменную типа "x"в переменные типа "y".

Изменение представления (или кодирование). Например, целочисленную переменную "i можно заменить, выражением "i‘ = c1*i + c2"где "c1,c2"являются константами.

2.2.2 Обфускация соединения

Один из важных этапов, в процессе реверсивной инженерии программ, основан на изучении структур данных. Поэтому важно постараться, в процессе обфускации, усложнить представление используемых программой структур данных. Например, при использовании обфускации соединения это достигается благодаря соединению независимых данных, или разделению зависимых.

Две или более переменных "v1,...,vk"могут быть объединены в одну переменную "V если их общий размер ("v1,...,vk") не превышает размер переменной "V". Например, рассмотрим простой пример объединения двух коротких целочисленных переменных "X "Y"(размером 16 бит) в одну целочисленную переменную "Z"(размером 32 бита).

Реструктурирование массивов, заключается в запутывании структуры массивов, путем разделения одного массива на несколько подмассивов, объединения нескольких массивов в один, сворачивания массива (увеличивая его размерность) и наоборот, разворачивая (уменьшая его размерность). Например, один массив "@A"можно разделить на несколько подмассивов "@A1, @A2 при этом один массив "@A1"будет содержать четные позиции элементов, а второй "@A2"нечетные позиции элементов массива "@A".

Изменение иерархий наследования классов, осуществляется путем усложнения иерархии наследования при помощи создания дополнительных классов или использования ложного разделения классов.

2.2.3 Обфускация переупорядочивания

Заключается в изменении последовательности объявления переменных, внутреннего расположения хранилищ данных, а также переупорядочивании методов, массивов (использование

нетривиального представления многомерных массивов), определенных полей в структурах и т.д.

2.3 Обфускация управления

Обфускация такого вида осуществляет запутывание потока управления, то есть последовательности выполнения программного кода.

Большинство ее реализаций основывается на использовании непрозрачных предикат, в качестве, которых выступают, последовательности операций, результат работы которых сложно определить (само понятие "предикат"выражает свойство одного объекта (аргумента), или отношения между несколькими объектами).

Определение. Предикат "P"считается непрозрачным предикатом, если его результат известен только в процессе обфускации, то есть после осуществления процесса обфускации, определение значения такого предиката, становится трудным.

Обозначим непрозрачный предикат, возвращающий всегда значение TRUE как "P(t) а возвращающий значение FALSE, как "P(f) тогда непрозрачный предикат, который может возвратить любое из этих двух значений (то есть или TRUE, или FALSE, что нам неизвестно) как "P(t,f)". Эти обозначения, будут использоваться дальше в контексте описания обфускации управления. Непрозрачные предикаты могут быть:

- локальными - вычисления содержаться внутри одиночного выражения (условия)
- глобальными - вычисления содержаться внутри одной процедуры (функции)
- межпроцедурными - вычисления содержаться внутри различных процедур (функций)

Эффективность обфускации управления в основном зависит от используемых непрозрачных предикат, это вынуждает создавать как можно сложные для изучения, и простые, гибкие в использовании непрозрачные предикаты, но в равной степени также не маловажную роль имеет время их выполнения, а также количество выполняемых операций, помимо всего этого предикат не сильно должен отличаться от тех функций, которые выполняет сама программа, и не должен содержать чрезмерное количество вычислений, в противном же случае злоумышленник, сможет сразу его обнаружить. Так как часто для деобфускации используют технологию статического анализа, а одним из ее недостатков является сложность (трудоемкость) статического анализа структур указателей, то обычно в процессе обфускации управления используют устойчивые непрозрачные предикаты, которые позволяют использовать недостатки технологии статического анализа.

Основная идея устойчивых непрозрачных предикатов состоит в том, что в программу, в

процессе обфускации добавляется код, который создает набор динамических структур, а также глобальных указателей, которые будут ссылаться на различные элементы внутри этих структур. Помимо этого, данный код должен иногда обновлять эти структуры (добавлять новые элементы в них, объединять или разделять некоторые их них, изменять значения глобальных указателей, и т.д.), но таким образом, чтобы при этом были сохранены некоторые условия, например "указатель p и q никогда не будут указывать на один и тот же элемент"или "указатель p может ссылаться (указывать) на указатель q"и т.д. Эти условия в последствии позволяют создавать требуемые непрозрачные предикаты.

Таких манипуляций с указателями, и структурами, можно делать очень много, они могут быть добавлены в разные участки программы, и их можно усложнить, а также добавить какие-то уникальные процедуры для работы со структурами. При этом существующие алгоритмы статического анализа становятся не эффективны.

Методы позволяющие осуществить обфускацию управления, классифицируются на три основных группы:

2.3.1 Обфускация вычислительная

Изменение касающиеся главной структуры потока управления. К ним можно отнести:

- расширения условий циклов. Для этого обычно используют непрозрачные предикаты, таким образом, чтобы они не коим образом не влияли на количество выполнений циклического кода.
- добавления недостижимого кода, (который не будет выполняться в процессе работы программы)
- устранение библиотечных вызовов. Большинство программ, используют функции, которые определены в стандартных библиотеках исходного языка, на котором писалась программа (например, в Си это библиотека "libc"), работа таких функции хорошо документирована и часто известна злоумышленникам, следовательно, их присутствие в коде программы, может помочь в процессе ее реверсивной инженерии. Поэтому имена функций из стандартных библиотек, также желательно придать обфускации, т.е. изменить на наиболее бессмысленные, которые потом будут фигурировать в коде защищаемой программы.
- добавление избыточных операций (мертвого кода) в те участки программного кода, которые наиболее трудные (изначально) для изучения. Часто избыточные операции, используются для расширения арифметических выражений (например, в непрозрачных предикатах).

- параллелизирование кода, заключается в разделении кода на отдельные независимые участки, которые во время работы программы будут выполняться параллельно (т.е. одновременно), такая обфускация также может заключаться в импровизации параллелизирования кода программы, для это создается так называемый макет процесса, который на самом деле не будет выполнять не каких полезных операций.

2.3.2 Обфускация соединения

Объединение или разделение определенных фрагментов кода программы, для того чтобы убрать логические связи между ними. Ниже приведены основные методы, позволяющие осуществить такую обфускацию:

- встраивание функций, осуществляется путем встраивания кода функции, в места ее вызова (если ее код будет встроен во все места ее вызова, тогда саму функцию можно убрать из кода программы).
- извлечение функций, является обратным действием, по отношению к встраиванию функций. Осуществляется в результате объединения некоторой группы взаимосвязанных операторов в коде исходной программы в отдельную функцию (при необходимости для этой функции можно определить некоторые аргументы), которой потом замещают эти группы операторов. Но следует учесть, что такое преобразование может быть снято компилятором в процессе компиляции кода программы.
- чередование, объединение фрагментов кода программы (функций например), выполняющих различные операции, воедино (в одну функцию, при этом в такую функцию, следует добавить объект, в зависимости от значения которого, будет выполняться код одной из объединенных функций).
- клонирование, данный метод позволяет усложнить анализ контекста использования функций, и объектов используемых в коде исходной программы. Процесс клонирования функций состоит в выделении определенной функции "F часто используемой в коде программы, после чего над кодом этой функции осуществляется трансформация, и создается ее клон "F‘ который также будет добавлен в код исходной программы, при этом часть вызовов функции "F"в коде исходной программы, будет замещена на вызов функции "F‘". В результате этого у злоумышленника создастся представление о том, что функции "F и "F‘"различны. Клонирование объектов осуществляется аналогичным способом.
- трансформация циклов. Циклы встречаются в коде различных программ, и их также можно придать трансформации. Блокирование циклов, заключается в добавлении

вложенных циклов в существующие, в результате работа существующих циклов будет заблокирована, на какой-то диапазон значений.

- Развертка циклов, повторение тела цикла один или более раз (если количество выполняемых циклов известно в процессе осуществления обфускации (например, равно "N"), то цикл, может быть, развернут полностью, в результате повторения его тела в коде N раз)
- Разделение циклов, цикл состоящий из более чем одной независимой операции можно разбить на несколько циклов (которые должны выполняться одинаковое количество раз), предварительно разбив на несколько частей, его тело.

2.3.3 Обфускация последовательность

Заключается в переупорядочивании блоков (инструкций переходов), циклов, выражений.

2.4 Превентивная обфускация

Защищает код от деобфускации специальными программами-деобфускаторами. Они основываются на обнаружении неиспользуемых кусков кода, нахождении наиболее сложных структур (фрагментов максимальной важности) и анализе статистических и динамических данных. Именно борьба с этими операциями – наиболее сложный и эффективный процесс обфускации. Здесь необходимо максимально точно подойти к анализу исходных данных, задействовать максимум предоставленных ресурсов, учесть подходы потенциальных оппонентов.

3 Практическое применение обфускатора

3.1 Исходный код

Для примера возьмём программу на языке C++ из первой лабораторной работы (листинг 1).

Листинг 1: Исходный код программы до обфускации (src/obfuscation/main.cpp)

```
#include <iostream>
#include <fstream>
#include <string>
#include <regex>

bool parse(char* ifname, long long* rx_bytes, long long* rx_packets,
           long long* tx_bytes, long long* tx_packets);

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " interface_name" <<
           std::endl;
        return 1;
    }

    long long rx_bytes = 0;
    long long rx_packets = 0;
    long long tx_bytes = 0;
    long long tx_packets = 0;

    if (!parse(argv[1], &rx_bytes, &rx_packets, &tx_bytes, &
       tx_packets)) {
        std::cerr << "Can't find such interface: " << argv[1] << std
           ::endl;
        return 1;
    }

    std::cout << argv[1] << ":" << std::endl;
    std::cout << "\tReceive " << rx_bytes << " bytes (" <<
       rx_packets << " packets)" << std::endl;
```

```
27      std::cout << "\tTransmit " << tx_bytes << " bytes (" <<
            tx_packets << " packets)" << std::endl;
28
29      return 0;
30  }
31
32  bool parse(char* ifname, long long* rx_bytes, long long* rx_packets,
33              long long* tx_bytes, long long* tx_packets) {
34      std::string interface(ifname);
35      interface.append(":");
36      std::string buff;
37      std::ifstream netstat("/proc/net/dev");
38
39      while(std::getline(netstat, buff)) {
40          size_t shift = buff.find_first_not_of(' ');
41          if (buff.compare(shift, interface.length(), interface) == 0)
                {
42              std::regex rx(R"([^[:alpha:]][[:digit:]]+[^[:alpha:]])")
                    ;
43              std::sregex_iterator pos(buff.cbegin(), buff.cend(), rx)
                    ;
44
45              *rx_bytes = std::stoll(pos->str());
46              ++pos;
47              *rx_packets = std::stoll(pos->str());
48              std::advance(pos, 7);
49              *tx_bytes = std::stoll(pos->str());
50              ++pos;
51              *tx_packets = std::stoll(pos->str());
52
53              return true;
54          }
55      }
56      return false;
57  }
```

3.2 Открытый обфускатор из стека LLVM

Low Level Virtual Machine (LLVM) – универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями. Может использоваться как оптимизирующий компилятор этого байт-кода в машинный код для различных архитектур, либо для его интерпретации и JIT-компиляции (для некоторых платформ).

Расширение obfuscator-llvm позволяет проводить обфускацию кода из отдельной сборки clang. Для этого нужно скачать исходный коды из git-репозитория разработчиков и собрать их на своей машине.

В этом обфускаторе реализована замена инструкций и уплотнение графа исполнения, только не для машинного кода x86/x86-64, а для промежуточного представления LLVM-IR. Он превращает скомпилированную программу в набор символов, который практически бесполезно изучать в дизассемблере.

```
$ git clone -b llvm-3.6.1 https://github.com/obfuscator-llvm/obfuscator.git
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE:String=Release ../obfuscator/
$ make -j5
$
```

3.3 Результат обфускации

Обфускатор работает на уровне промежуточного представления. Для компиляции использовался обфускатор, собранный на предыдущем шаге.

```
$ build/bin/clang++ main.cpp -o netmonitor -mllvm -sub -mllvm -fla -std=c++14
```

Сам граф получен путём визуализации вызовов, полученных из valgrand

```
$ valgrind --tool=callgrind -v --dump-every-bb=10000000 ./netmonitor enp2s0
```

Граф без обфускации показан на рисунке 1.

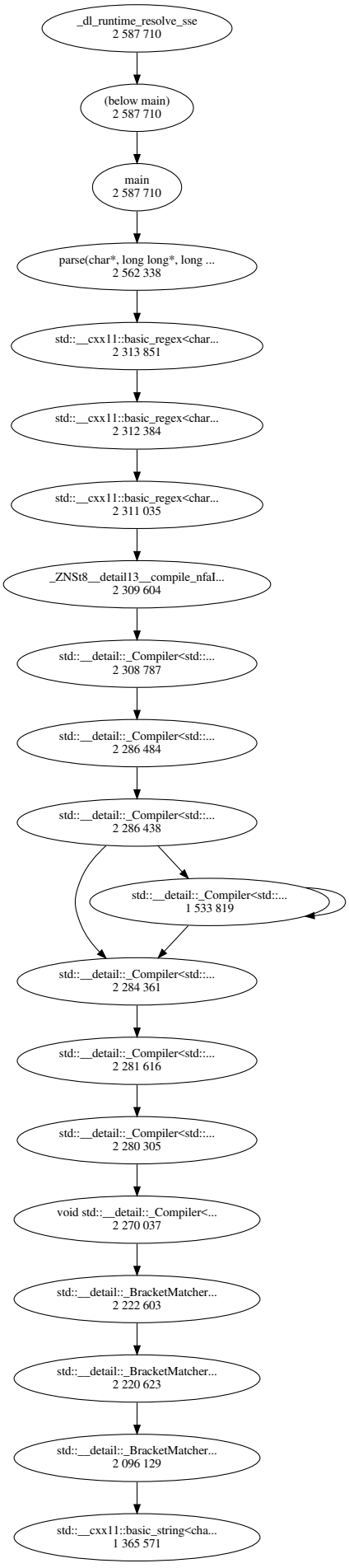

Рис. 1: Граф вызовов до обфускации

Графов вызовов после обфускации (рисунок 2).

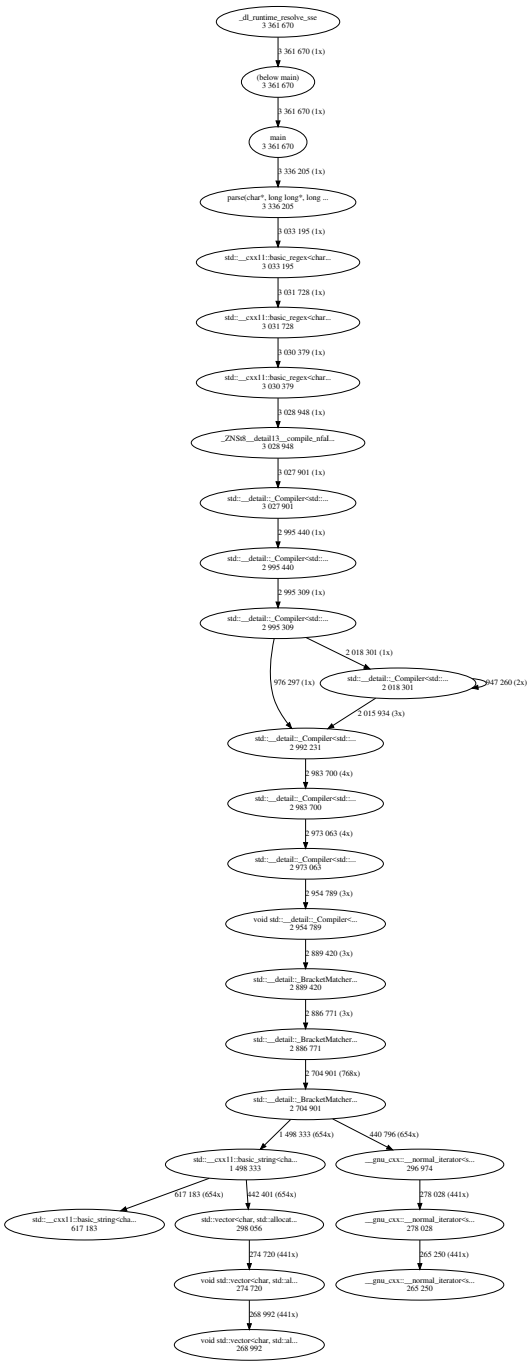

Рис. 2: Граф вызовов после обфускации

Как видно, графы достаточно похожи. Дело в том, что большая часть работы (парсинг строки с использованием регулярных выражений) происходит в стандартной библиотеке C++. Для повышения зашиты можно либо обфусцировать стандартную библиотеку, либо отказаться от её использования и переписать парсинг самостоятельно. Но та часть, которая отвечает за вывод информации, и которая как раз подверглась обфускации, отличается достаточно значительно.

Заключение

В распоряжении разработчиков на данный момент не существует хороших обфускаторов, а те обфускаторы, которые широко используются сегодня, весьма примитивны — они могут переставлять инструкции, заменять имена переменных, вставлять куски кода, которые на самом деле имеют нулевой эффект и делать аналогичные вещи, которые в целом можно назвать «безопасность через непонятность». Но такие обфускации при небольшом усердии легко деобфусцировать, а потому это не преграда для хороших хакеров.

Таким образом, обфускация не может рассматриваться как последний бастион защиты исходных кодов, однако в ситуации, когда де-обфускация стоит дороже создания новой программы это может стать достаточным способом защиты данных.

Список литературы

[1] Diffie W., Hellman M. New directions in cryptography // IEEE Transactions on Information Theory, IT-22(6), 1976, p.644-654.

[2] Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations // Technical Report, N 148, Univ. of Auckland, 1997.

[3] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S. and Yang K. «On the (im) possibility of obfuscating programs.» CRYPTO 2001.

[4] Garg S., Gentry C., Halevi S., Raykova M., Sahai A., and Waters B. «Candidate indistinguishability obfuscation and functional encryption for all circuits.» FOCS 2013.

[5] Goldwasser S., and Guy N. R. «On best-possible obfuscation.» TCC 2007.