------------------------------------- Слайд № 1

**Введение**

Инструментирование — это возможность наблюдать, контролировать и изменять поведение ПРОГРАММЫ. То есть представьте, что у нас есть какая-то программа и нам надо либо обнаружить уязвимости, которым она подвержена, либо для тестирования и улучшения безопасности, либо для написания определенных эксплойтов. Методы инструментации придуманы как раз для этого.

------------------------------------- Слайд № 2

Функций у инструментации очень много и рядовой разработчик сталкивается с ней достаточно часто, например, в среде разработки Visual Studio при запуске программы мы видим использование памяти и ЦП процесса. Или, например, интерактивный дизассемблер IDA тоже является средством инструментирования, только уже статическим. Сегодня я расскажу о классификации средств иструментирования, зачем эта технология используется в нашей с вами профессиональной среде, примеры фреймворков и инструментов, а также рассмотрим некоторые из них с примерами кода.

------------------------------------- Слайд № 4

Цели работы:

1. Изучение теоретической базы
   * Устройство программ
   * Взаимодействие программ с ядром ОС Linux
   * Что такое инструментация
   * Её виды
   * Существующее ПО для инструментации
2. Выбор средств инструментации из имеющихся
   * Анализ имеющихся средств
   * Выбор подходящего ПО для реализации
3. Изучение выбранного средства
   * Изучение
   * Рассмотрение примеров использования
4. Разработка ПО, основанного на выбранном, для логгирования всех функций, вызываемых в приложении
   * Разработка
   * Тестирование
   * Отладка
5. Описание возникших проблем

------------------------------------- Слайд № 5

Для чего «безопасники» используют инструментирование

* Data flow analysis (Анализ потока данных)

Мы можем отслеживать как, куда и по каким инструкциям перемещаются данные в программе в реальном времени, в виду большой нелинейности в современных программах, это достаточно востребованная функция

* Control flow analysis (Анализ потока управления)

Собственно, анализируя выполнение программы в динамике, мы можем точно понять, какие ветки выбираются, как происходит вызов функций, с какими параметрами и в каком состоянии, куда передаётся управление, а также многое другое

* Fuzzing

Очень часто в связке с DBI используются средства для фазинга, чтобы протестировать как можно больше кода, как можно веток обойти и увеличить покрытие кода

* Vulnerability detection (Обнаружение уязвимостей)

Есть множество исключительных ситуаций, которые приложения не обрабатывают, которые возникают не только по вине программистов, а также из-за принципиальных особенностей ПО, либо же самой ОС (далее мы рассмотрим средство инструментации ядра ОС Linux)

* Program visualization (Визуализация программы)

Опять же, используя вышеупомянутые методы и средства, можно визуализировать ход выполнения программы

* Taint analysis

Taint Analysis - популярный метод, который заключается в проверке того, какие переменные могут быть изменены пользователем. Любой пользовательский ввод может быть опасен, если он не проверен должным образом. С помощью этого метода можно проверить регистры и области памяти, которыми может управлять пользователь.

* Reverse engineering
* Transparent debugging (Прозрачная отладка)

Существует необходимость отлаживать динамически инструментированные программы, сохраняя при этом присутствие динамической инструментальной системы скрытым от пользователей отладки. Существующие отладчики используют отладочную информацию в двоичных файлах программы, которые были сгенерированы компилятором во время статической компиляции, для обеспечения своей отладочной поддержки. Поскольку динамические измерительные системы генерируют программный код во время выполнения, существующие отладчики не могут обеспечить такую же поддержку отладки.

------------------------------------- Слайд № 6

**Динамический vs Статический анализ**

При анализе программ есть два возможных сценария: когда у нас есть исходный код приложения и когда у нас его нет. В большинстве случаев исходный код исследуемого приложения отсутствует, что при статическом анализе исполняемых файлов приводит к появлению целого ряда проблем. Например, невозможности отличить данные от кода, что, в свою очередь, ведет не только к неполному анализу, но и к получению ложных данных. А если исследуемая программа использует динамически генерируемый код, то даже наличие исходников не даст полную картину ее работы. Современные приложения все чаще собираются и определяются во время своего выполнения, используя разделяемые библиотеки, виртуальные функции, подключаемые плагины, динамически генерируемый код и прочие динамические механизмы. Поэтому количество получаемой с помощью статического анализа информации об исследуемой программе со временем становится все меньше и меньше. Статические инструменты уже не способны дать даже полного покрытия кода, что раньше было их неоспоримым преимуществом, не говоря уже о невозможности точно отличать данные от кода. В связи с этим появляется четкая необходимость в мощном динамическом подходе, сочетающем в себе простоту реализации и хорошую производительность.

Давайте рассмотрим особенности разных подходов.

В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемой программы. Обычно при этом строится абстрактная модель программы, которая, собственно, и является объектом анализа.   
При этом следует подчеркнуть следующие характерные особенности статического анализа:

* Возможен раздельный анализ отдельно взятых фрагментов программы (обычно отдельных функций или процедур), что дает достаточно эффективный способ борьбы с нелинейным ростом сложности анализа.
* Возможны ложные срабатывания, обусловленные либо тем, что при построении абстрактной модели некоторые детали игнорируются, либо тем, что анализ модели не является исчерпывающим. То есть при поиске проблемных частей, мы видим либо не точный вид исследуемой программы, либо не весь код нам доступен
* При обнаружении дефекта возникают, во-первых, проблема проверки истинности обнаруженного дефекта ([false positive](http://ru.wikipedia.org/wiki/False_positive)) и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определенных входных данных. То есть то, что мы обнаружили и пометили, как потенциальная уязвимость, не всегда таковым является. А уж дойти до предполагаемого места дефекта – отдельная задача.

В отличие от статического анализа, динамический анализ осуществляется во время работы программы. При этом:

* Для запуска программы требуются некоторые входные данные.
* Динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными; дефекты, находящиеся в других частях программы, не будут обнаружены.
* В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения.

На первом этапе идет подготовка приложения к профилированию — инструментация. Затем запуск, и уже при выполнении накапливается информация о работе приложения. Собранная информация представляет собой трассу приложения (список выполненных инструкций программы), которая затем обрабатывается. Как уже говорилось, инструментацией называют процесс модификации исследуемой программы. За вставку дополнительного кода обычно отвечают инструментирующие процедуры. Они вызываются только раз при возникновении необходимого события и модифицируют целевую программу, добавляя в нее анализирующие процедуры, которые отвечают за необходимый анализ, изменение и мониторинг исследуемой программы и вызываются каждый раз при достижении определенного участка кода. Инструментация бинарного приложения может быть выполнена на разных уровнях гранулярности программы:

* инструкции;
* базового блока;
* трассы;
* процедуры;
* секции бинарного файла;
* бинарного образа.

Анализирующие процедуры обычно реализованы в виде процедур обратного вызова, которые срабатывают при достижении определенного участка кода или когда в программе происходит заданное событие. Для создания таких инструментов, работающих во время выполнения программы, были разработаны специальные наборы библиотек динамической бинарной инструментации.

------------------------------------- Слайд № 7

**Виды инструментации**

Для составления полной картины об инструментации кода рассмотрим все ее виды. В этом нам поможет следующая классификация:

1. Исходные данные: исходный код, байт-код, исполняемый файл.
2. Объект изменения: исходный код, компилятор, исполняемый файл, процесс, интерпретатор / виртуальная машина, среда выполнения, архитектурные возможности аппаратного обеспечения.
3. Технология изменения: инструментация исходного кода, инструментация кода компилятором, статическая инструментация кода, динамическая инструментация кода, отладчики, воспроизведение работы среды, возможности по отладке, предоставленные аппаратным обеспечением.
4. Способ достижения технологии: ручная/автоматизированная вставка анализирующих функций, настройка компилятора, статическая бинарная инструментация, модификация загрузчика, эмуляция, виртуализация, паравиртуализация, патчинг, статическая бинарная инструментация, динамическая бинарная инструментация, отладочный API OC, эмуляция работы оборудования, отладочные регистры процессора, аппаратные отладчики.

Ситуация, когда доступен исходный код, возможна, только если ты являешься автором проекта или исследуемый проект опенсорсный. При этом возможно проанализировать только ту часть приложения, исходный код которой присутствует, — проанализировать сторонние (например, системные или чужие) библиотеки невозможно, так как их исходники почти всегда отсутствуют. Значительный плюс наличия исходников исследуемого приложения — полное знание о его высокоуровневом устройстве (названия переменных, типы переменных и так далее). Обратной стороной этого является полное отсутствие низкоуровневой информации (значение регистров, флагов, представления, как данные хранятся в памяти, и так далее).

Байт-код — машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором / виртуальной машиной. Большинство инструкций байт-кода эквивалентны одной или нескольким командам ассемблера. Трансляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией. C ним можно встретиться при анализе программ, написанных на Java/C#. Но это также нас не интересует.

Наличие только исполняемого бинарного файла — наиболее часто встречаемая ситуация, тем более при работе исследователя информационной безопасности. В этом случае уже хорошо, если для исследуемой программы доступны хотя бы отладочные символы.

Вариант, когда на руках у нас только сам исполняемый файл. Для его анализа мы можем влиять на него либо непосредственно (вставить в файл инструментирующий код, а затем запустить) либо в процессе работы (на процесс, на программное окружение, в котором он выполняется). Также мы можем использовать архитектурные возможности аппаратного обеспечения, на котором анализируется приложение. Все варианты можно представить в виде следующего списка:

------------------------------------- Слайд № 8

* Исполняемый файл
  + Статическая инструментация кода
    - Статическая бинарная инструментация
* Процесс
  + Динамическая инструментация кода
    - Динамическая бинарная инструментация
  + Перехват таблицы вызовов
    - IAT
  + Отладчики
    - Отладочный API OC
* Среда выполнения
  + Перехват таблицы вызовов
    - IDT, CPU MSRs, GDT, SSDT, IRP-таблица
  + Воспроизведение работы среды
    - Эмуляция
    - Виртуализация
* Архитектурные возможности
  + Возможности по отладке, предоставленные аппаратным обеспечением
    - Отладочные регистры процессора
    - Аппаратные отладчики

Нас интересуют статическая и динамическая бинарные инструментации, но времени не так много, поэтому перейдём сразу к DBI.

**Динамическая бинарная инструментация**

Динамическая бинарная инструментация выполняется во время работы программы и состоит из нескольких базовых состояний: 1) получение управления от приложения; 2) сохранение состояния программы; 3) выполнение задач; 4) восстановление состояния программы; 5) возвращение управления приложению.

Получение управления от приложения обычно происходит с помощью перезаписи текущих инструкций, с их сохранением в определенном месте.

Для реализации динамической инструментации создают специальные программные библиотеки.

------------------------------------- Слайд № 9

Те самые программы, которые всё делают за нас:

1. PIN
2. DynamoRIO
3. Dyninst
4. Syzcaller
5. Valgrind
6. BAP
7. KEDR
8. ERESI
9. Detours
10. Vulcan
11. SpiderPig

Назначение DBI инструментов - добавить обработчик pre / post для каждой инструкции. Когда вызывается обработчик, мы можем получить всю необходимую информацию об инструкции или среде (памяти).

Прежде чем рассмотреть эти инструменты, стоит заметить, что для грамотного использования бинарной динамической инструментации, а главное использования в нужных местах, необходимо знать устройство программ (kernel и user space, trace, BBL и тд), исполняемых файлов (сегменты кода, данных, heap, stack и тд), взаимодействие программ с операционной системой и её ядром (системные вызовы и тд), базовые понятия о работе компиляторов и их компоновщиков, так же для метода reversing, потребуются базовые знания ассемблера.

**Инструменты динамической бинарной инструментации**

Существует достаточно много инструментов для DBI, который позволяют во время выполнения манипулировать программой, её сегментами, используемыми регистрами и средой выполнения. **Суть таких фреймворков заключается в создании окружения выполнения программ и контролировании всех ресурсов, от уже упомянутых регистров и сегментов кода и данных, до процессов, потоков и сигналов.** В отличии от статического анализа мы проверяем только те инструкции, которые выполняются в зависимости от входных данных, поэтому DBI следует использовать в связке с фазингом, проверяя не только возможные уязвимости, но и возможные инструкции, появляющиеся в следствии ветвления.

------------------------------------- Слайд № 10

**Intel Pin**

Первое, найденное мной и подходящее под задачу, решение является Intel Pin.

Intel Pin — это DBI фреймворк для создания инструментов анализа userspace приложений. Возможности фреймворка позволяют управлять поведением уже скомпилированных программ, встраивая произвольный код, написанный на С/C++, прямо во время исполнения программы.

Pintool представляет из себя скомпилированный бинарный файл. Для Linux-систем — это разделяемая библиотека с расширением .so, для Windows – динамическая библиотека с расширением .dll.

Intel активно использует возможности своего фреймворка в своих продуктах, например: Intel® VTune ™ Amplifier, Intel® Inspector, Intel® Advisor и Intel® Software Development Emulator (Intel® SDE).

------------------------------------- Слайд № 11

Взаимодействие Intel Pin и анализируемой программы можно изобразить в виде схемы:

Init.

Translate code

Instrument trace

Place trace in cache

Execute from cache

Fetch next trace

Чтож хотелось бы уже рассмотреть практическое применение, однако давайте разберёмся с алгоритмом выполнения PIN

1. Инициализация – тут всё очевидно и понятно
2. Далее мы работаем с трассой – есть несколько уровней гранулированности программы, с которыми может работать PIN

------------------------------------- Слайд № 12

* 1. INS – Instruction

Простейшая инструкция на языке ассемблера.

* 1. BBL – Basic Block

понятие, обозначающее последовательность инструкций или кода, имеющую одну [точку входа](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BA%D0%B0_%D0%B2%D1%85%D0%BE%D0%B4%D0%B0), одну точку выхода и не содержащую инструкций передачи управления ранее точки выхода. То есть это линейный отрезок программы, без ветвления, без циклов без вызовов и передачи управления.

* 1. TRACE – Trace

Трасса — это некоторая последовательность инструкций, а точнее basic blocks, за трассу можно принимать много чего, например, одну команду на языке высокого уровня, либо же 1 команду в файле .obj

* 1. RTN – Routine

Это разделы машинного кода, которые не выполняются последовательно. Машина прыгает туда через команду вызова, при этом состояние компьютера сохраняется в стеке, а затем прыгает обратно через команду возврата, которая восстанавливает состояние компьютера. Это на машинном уровне. На программном уровне это называется: Процедура, функция, метод, подпрограмма, задача, процесс, плагин, файл cookie, расширение, обработчик прерываний (который называется по-разному) и, вероятно, еще много других имен используются.

* 1. SEC – Section

Разделы — это не более чем независимые последовательности памяти.

* 1. IMG – Binary image
  2. Так же могут быть инструментированы процессы, потоки исключения и системные вызовы

------------------------------------- Слайд № 11

Данная иллюстрация приведена для трассы, для других частей программы происходит аналогично, смотря на каком уровне мы работаем. Так вот на этом этапе мы берём очередную трассу приложения для инструментирования (ВАЖНО! До её выполнения)

1. Транслирование кода – тут всё просто – переводим это в промежуточный код, для того чтобы легче было инструментировать
2. Основной этап – инструментирование трассы. То есть модифицирования, добавление анализирующих средств
3. Помещение трассы в кэш – понятно из названия – инструментированную трассу помещаем в кэш для дальнейшего выполнения
4. Выполнение содержимого кэша

Имея все эти знания давайте рассмотрим простейший пример использования PIN API

------------------------------------- Слайд № 13

**#include "pin.h"**

**UINT64 icount = 0;**

**void docount() { icount++; }**

**void Instruction(INS ins, void \*v)**

**{**

**INS\_InsertCall(ins, IPOINT\_BEFORE,**

**(AFUNPTR)docount, IARG\_END);**

**}**

**void Fini(INT32 code, void \*v)**

**{**

**std::cerr << "Count " << icount << endl;**

**}**

**int main(int argc, char \* argv[]) {**

**PIN\_Init(argc, argv);**

**INS\_AddInstrumentFunction(Instruction, 0);**

**PIN\_AddFiniFunction(Fini, 0);**

**PIN\_StartProgram(); // Never returns**

**return 0;**

**}**

Видим подключение библиотеки, инициализацию счётчика, значит будем что-то считать. Функция инкрементации, почему именно функция для такого простого действия – так требует APIшка. Какая-то функция Instruction, очевидно, связана как-то с инструкциями. Следующая функция для вывода результатов, это видно по содержанию. И наконец main функция. Что же делает эта программа – считает количество инструкций целевой программы, быстро пройдёмся по функциям, используемыми нами.

PIN\_Init – инициализирует PIN систему

INS\_AddInstrumentFunction – добавляет функцию, используемую для инструментации на гранулированности инструкции (то есть функция инструментации каждой инструкции)

PIN\_AddFiniFunction – Вызывает функцию непосредственно перед выходом приложения. Функция не является функцией инструментирования – она не может вставлять инструментарий. Может быть несколько функций Fini.

PIN\_StartProgram – как не сложно догадаться – запускает программу.

INS\_InsertCall – вставляет вызов docount относительно инструкции ins, второй аргумент это флаги, возможны 3 флага – до инструкции, после и для неветвящихся, не будем углубляться. Последний параметр – аргументы для функции docount, которые кончаются IARG\_END, то есть в нашем примере отсутствуют аргументы.

Если смотреть на реализацию, то будет примерно следующее:

**inc icount**

**sub $0xff, %edx**

**inc icount**

**cmp %esi, %edx**

**save eflags**

**inc icount**

**restore eflags**

**jle <L1>**

**inc icount**

**mov 0x1, %edi**

Сама же функция Instruction работает с инкрементированием регистра, следовательно, чтобы не портить логику работы программы она должна перед исполнением своих инструкций сохранить состояние стэка, флагов и регистров,переключить на свой собственный PIN-стэк, а после выполнения своего тела восстановить всё обратно.

------------------------------------- Слайд № 14

Сообственно, вызов данного чуда будет выглядеть так:

pin [pin\_options] -t pintool.so [pintool\_options] –app\_name.exe [app\_args]

Запуск app\_name.exe приложения с [app\_args] аргументами, под управлением PIN с [pin\_options] аргументами, используя pintool.dll, в которую подаются [pintool\_options]

Считаю довольно удобным данный инструмент, простым для написания своих собственных библиотек, и, как ни странно, поражающим функционалом, способным на очень многое, ещё и разными способами, например, рассмотрим ту же задачу, но решим её по-другому:

------------------------------------- Слайд № 15

**#include "pin.H"**

**UINT64 icount = 0;**

**void PIN\_FAST\_ANALYSIS\_CALL docount(INT32 c) { icount += c; }**

**void Trace(TRACE trace, void \*v){// Pin Callback**

**for(BBL bbl = TRACE\_BblHead(trace);**

**BBL\_Valid(bbl);**

**bbl = BBL\_Next(bbl))**

**BBL\_InsertCall(bbl, IPOINT\_ANYWHERE,**

**(AFUNPTR)docount, IARG\_FAST\_ANALYSIS\_CALL,**

**IARG\_UINT32, BBL\_NumIns(bbl),**

**IARG\_END);**

**}**

**void Fini(INT32 code, void \*v) {// Pin Callback**

**fprintf(stderr, "Count %lld\n", icount);**

**}**

**int main(int argc, char \* argv[]) {**

**PIN\_Init(argc, argv);**

**TRACE\_AddInstrumentFunction(Trace, 0);**

**PIN\_AddFiniFunction(Fini, 0);**

**PIN\_StartProgram();**

**return 0;**

**}**

Тут мы делаем тот же подсчёт инструкций, только оперируем не мелкими составляющими – инструкциями, а уже работаем с трассой.

------------------------------------- Слайд № 16

Иногда, занимаясь реверсом двоичных файлов, нам нужно как-то измерить или просто иметь представление о том, насколько конкретное выполнение покрывает код нашей цели. Это может быть для целей фаззинга, например, у нас есть набор входных, и мы хотим узнать покрытие. Или, может быть, нас интересуют различия в охвате между двумя исполнениями: например, чтобы определить, где ваша программа обрабатывает определенные входные данные. Сейчас проиллюстрируем.

Вообще говоря, это не тривиальная проблема, в хороших программах использование статических методов, таких как IDA, не даёт нужного результата.

Мы можете четко зарегистрировать, что наша программа делает с вводом A, что она делает с вводом B, а затем проанализировать различия. Таким образом, мы получим информацию о том, какой вход имеет лучший охват, чем другой, а также какие инструкции выполняются при определенных входных данных.

#include <pin.h>

#include <map>

#include <string>

#include <iostream>

#include <set>

typedef std::map<std::string, std::pair<**ADDRINT**, **ADDRINT**> > **MODULE\_BLACKLIST\_T**;

typedef **MODULE\_BLACKLIST\_T MODULE\_LIST\_T**;

typedef std::map<**ADDRINT**, UINT32> **BASIC\_BLOCKS\_INFO\_T**;

// Количество всех инструкций, выполненных программой

**UINT64** instruction\_counter = 0;

// Количество потоков

**UINT64** thread\_counter = 0;

// Список модулей, внесенных в черный список

**MODULE\_BLACKLIST\_T** modules\_blacklisted;

// Для каждого выполненного базового блока мы храним его адрес и номер инструкции

**BASIC\_BLOCKS\_INFO\_T** basic\_blocks\_info;

// Для каждого загруженного модуля мы сохраняем его имя и начальные/конечные адреса

**MODULE\_LIST\_T** module\_list;

VOID image\_instrumentation(IMG img, VOID \* v)

{

ADDRINT module\_low\_limit = **IMG\_LowAddress(img)**, module\_high\_limit = **IMG\_HighAddress(img)**;

if(**IMG\_IsMainExecutable(img)**)

return;

const std::string image\_path = IMG\_Name(img);

std::pair<std::string, std::pair<ADDRINT, ADDRINT> > module\_info = std::make\_pair(

image\_path,

std::make\_pair(

module\_low\_limit,

module\_high\_limit

)

);

module\_list.insert(module\_info);

module\_counter++;

if(is\_module\_should\_be\_blacklisted(image\_path))

modules\_blacklisted.insert(module\_info);

}

VOID trace\_instrumentation(TRACE trace, VOID \*v)

{

for(BBL bbl = TRACE\_BblHead(trace); BBL\_Valid(bbl); bbl = BBL\_Next(bbl))

{

if(is\_address\_in\_blacklisted\_modules(BBL\_Address(bbl)))

continue;

BBL\_InsertCall(

bbl,

IPOINT\_ANYWHERE,

(AFUNPTR)handle\_basic\_block,

IARG\_FAST\_ANALYSIS\_CALL,

IARG\_UINT32,

BBL\_NumIns(bbl),

IARG\_ADDRINT,

BBL\_Address(bbl),

IARG\_END

);

}

}

**VOID PIN\_FAST\_ANALYSIS\_CALL** handle\_basic\_block(UINT32 number\_instruction\_in\_bb, ADDRINT address\_bb)

{

basic\_blocks\_info[address\_bb] = number\_instruction\_in\_bb;

}

------------------------------------- Слайд № 19

С помощью питон скрипта, который я нашёл на одном французском форуме посвященному PIN и использующий IDAPython, можно получить визуализацию результата работы нашей программы.



Слева мы просто пингуем гугл утилитой ping. Посередине пингуем с ключом -n 10. Затем сравниваем и путём вычитания на правой картинке видим блоки кода, которые выполнялись при втором вызове, но не выполнялись при первом.

------------------------------------- Слайд № 20

**Обнаружение уязвимостей**

Мы можем обнаруживать уязвимости программы с помощью наших инструментов

* Перезапись обратного вызова

Это наш вариант, далее покажу, как обнаружить такие уязвимости.

* Повреждение мета-данных

Эта уязвимость связана с кучей, то есть heap. Которая используется для динамического выделения памяти. И на неё есть несколько уязвимостей, которые я в данной курсовой не рассматриваю, но справедливости ради нужно упомянуть, что именно эта уязвимость довольно давняя и давно исправленная во многих системах.

* Выполнение данных пользователя

Этот тип уязвимостей знают все, кто хоть немного интересовался хакингом, сюда входят все инъекции

* Перезапись указателей функций

Очень похоже на первый вариант по использованию, но это мы тоже не будем рассматривать

------------------------------------- Слайд № 21

Давайте посмотрим, как можно использовать PIN для обнаружения уязвимости перезаписи обратного адреса. Допустим, в программе есть возможность переполнения буфера и получается повредить стэк программы, что может привести к очень неприятным последствиям, например, к перезаписи обратного адреса. В реальном эксплойте обратный адрес будет перезаписан значимым адресом, который нужен злоумышленникам, например, указывающим на область стека, в которой уже хранится вредоносный код, или на системные библиотеки или другие уязвимые приложения/программы, доступные в системе.

Последовательность обнаружения перезаписи адреса возврата для функций в определенном двоичном файле

* Перед функцией: сохраняем ожидаемый обратный адрес
* После функции: проверяем, что обратный адрес не был изменен

Делаем свой собственный теневой стэк. Это стэк — это механизм защиты сохраненного адреса возврата процедур. Сам теневой стэк — это второй отдельный стек, который "затеняет" стек вызовов программы. Используется как раз для предотвращения уязвимостей переполнения стэка.

------------------------------------- Слайд № 22

**#include <stdio.h>**

**#include "pin.H"**

**#include <stack>**

**typedef struct**

**{**

**ADDRINT address;**

**ADDRINT value;**

**} pAddr;**

**stack<pAddr> protect; //addresses to protect**

**FILE \* logfile; //log file**

**// called at end of process**

**VOID Fini(INT32 code, VOID \*v)**

**{**

**fclose(logfile);**

**}**

**// Save address to protect on entry to function**

**VOID RtnEntry(ADDRINT esp, ADDRINT addr)**

**{**

**pAddr tmp;**

**tmp.address = esp;**

**tmp.value = \*((ADDRINT \*)esp);**

**protect.push(tmp);**

**}**

**// check if return address was overwritten**

**VOID RtnExit(ADDRINT esp, ADDRINT addr) {**

**pAddr orig = protect.top();**

**ADDRINT cur\_val = (\*((ADDRINT \*) addr));**

**if (orig.value != cur\_val) {**

**fprintf(logfile, "Overwrite at: %x old value: %x, new value: %x\n",**

**orig.address, orig.value, cur\_val );**

**}**

**protect.pop();**

**}**

**//Called for every RTN, add calls to RtnEntry and RtnExit**

**VOID Routine(RTN rtn, VOID \*v) {**

**RTN\_Open(rtn);**

**SEC sec = RTN\_Sec(rtn);**

**IMG img = SEC\_Img(sec);**

**if ( IMG\_IsMainExecutable(img) && (SEC\_Name(sec) == ".text") )**

**{**

**RTN\_InsertCall(rtn, IPOINT\_BEFORE, (AFUNPTR)RtnEntry, IARG\_REG\_VALUE,**

**REG\_ESP, IARG\_INST\_PTR, IARG\_END);**

**RTN\_InsertCall(rtn, IPOINT\_AFTER ,(AFUNPTR)RtnExit , IARG\_REG\_VALUE,**

**REG\_ESP, IARG\_INST\_PTR, IARG\_END);**

**}**

**RTN\_Close(rtn);**

**}**

**// Tool main function - initialize and set instrumentation callbacks**

**int main(int argc, char \*argv[])**

**{**

**// initialize Pin + symbol processing**

**PIN\_InitSymbols();**

**if (PIN\_Init(argc, argv)) return Usage();**

**// open logfile**

**logfile = fopen("protection.out", "w");**

**// set callbacks**

**RTN\_AddInstrumentFunction(Routine, 0);**

**PIN\_AddFiniFunction(Fini, 0);**

**// Never returns**

**PIN\_StartProgram();**

**return 0;**

**}**

Вывод

Средства инструментирования создавались для тестирования кода на разные уязвимости и поломки, для определения покрытия кода, визуализации программы и просто для анализа её выполнения. Напомню, что исходный код мало того, что не всегда доступен, так ещё и не даёт столько полезной информации о регистрах и состоянии программы. Но, как это всегда бывает, используют эти средства не только для защиты.

**DynamoRIO**

Теперь рассмотрим другой фреймворк для инструментации.

DynamoRIO экспортирует пользователю богатый API для создания клиента DynamoRIO. Клиент DynamoRIO — это библиотека, которая соединена с DynamoRIO для совместной работы с двоичной входной программой:

Для взаимодействия с клиентом DynamoRIO предоставляет определенные события, которые клиент может перехватить. Функции перехвата событий, если они предоставляются пользователем-клиентом, вызываются DynamoRIO в соответствующее время.

**Общие события**

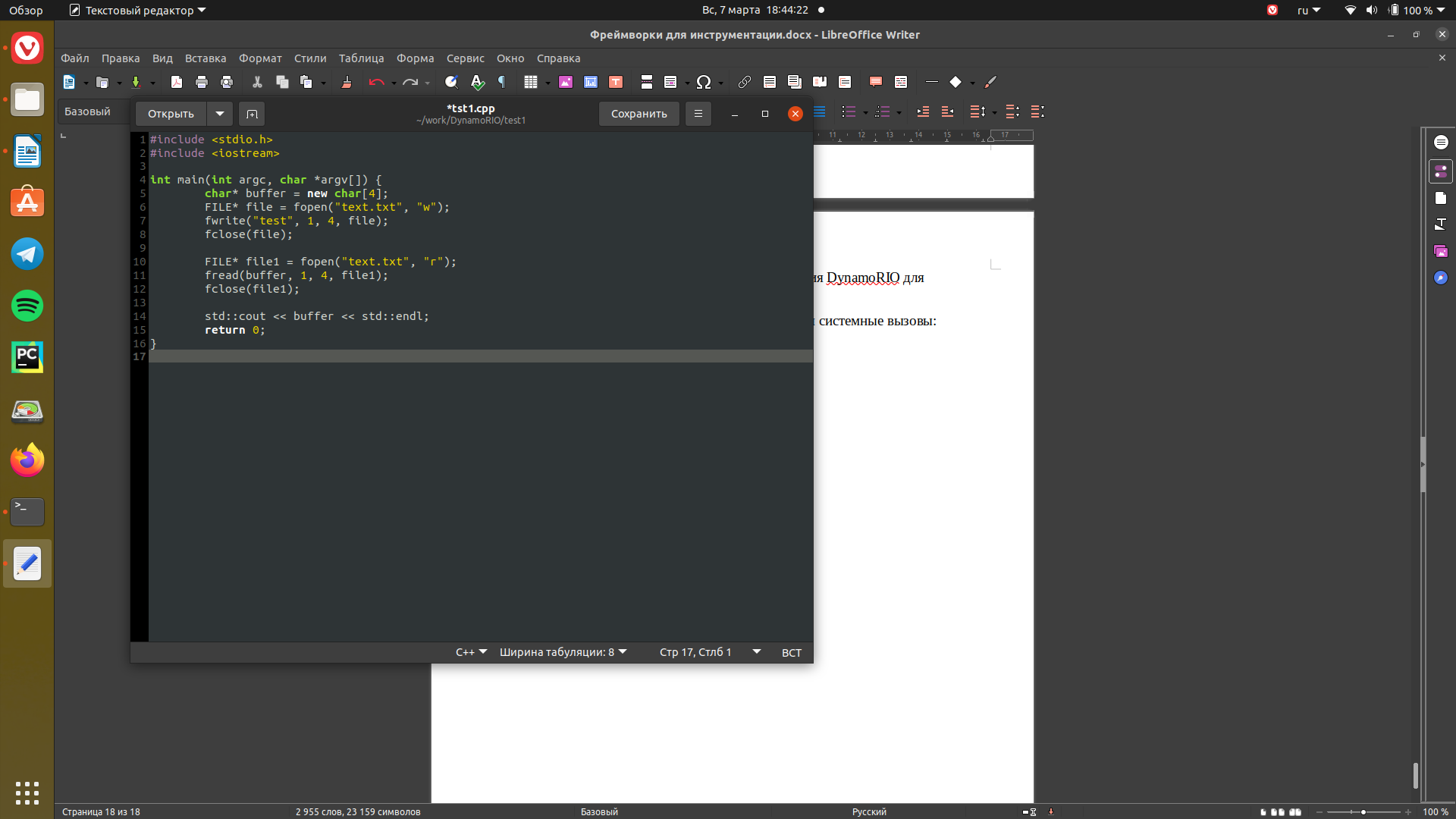
Основное взаимодействие клиента с системой DynamoRIO осуществляется через набор обратных вызовов событий. Эти события включают в себя следующее:

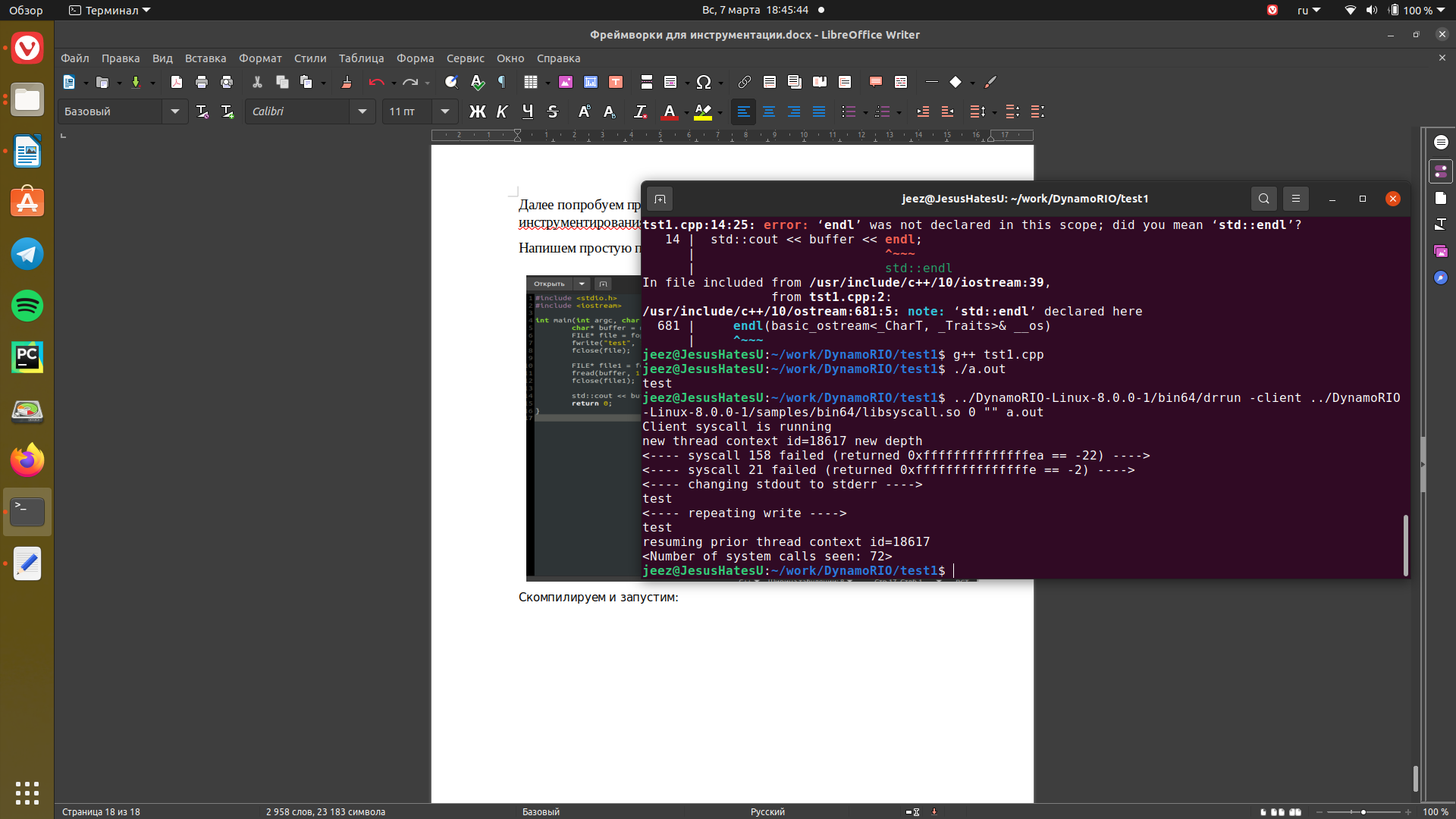
* Создание или удаление базовых блоков и трасс
* Инициализация и выход процесса
* Инициализация и выход потока
* Fork child initialization (только для Linux); предназначен для повторной инициализации структур данных и создания новых файлов журнала
* Загрузка и выгрузка библиотеки приложений
* Ошибка приложения или исключение (сигнал в Linux)
* Перехват системных вызовов: предсистемный вызов, постсистемный вызов и фильтрация системных вызовов по номеру
* Перехват сигнала (только для Linux)
* Nudge received

Как правило, клиент регистрирует нужные события при инициализации в своей подпрограмме dr\_init (). Затем DynamoRIO вызывает зарегистрированные функции в соответствующее время. Каждое событие имеет определенную процедуру регистрации (например, dr\_register\_thread\_init\_event()) и связанную с ней процедуру отмены регистрации.

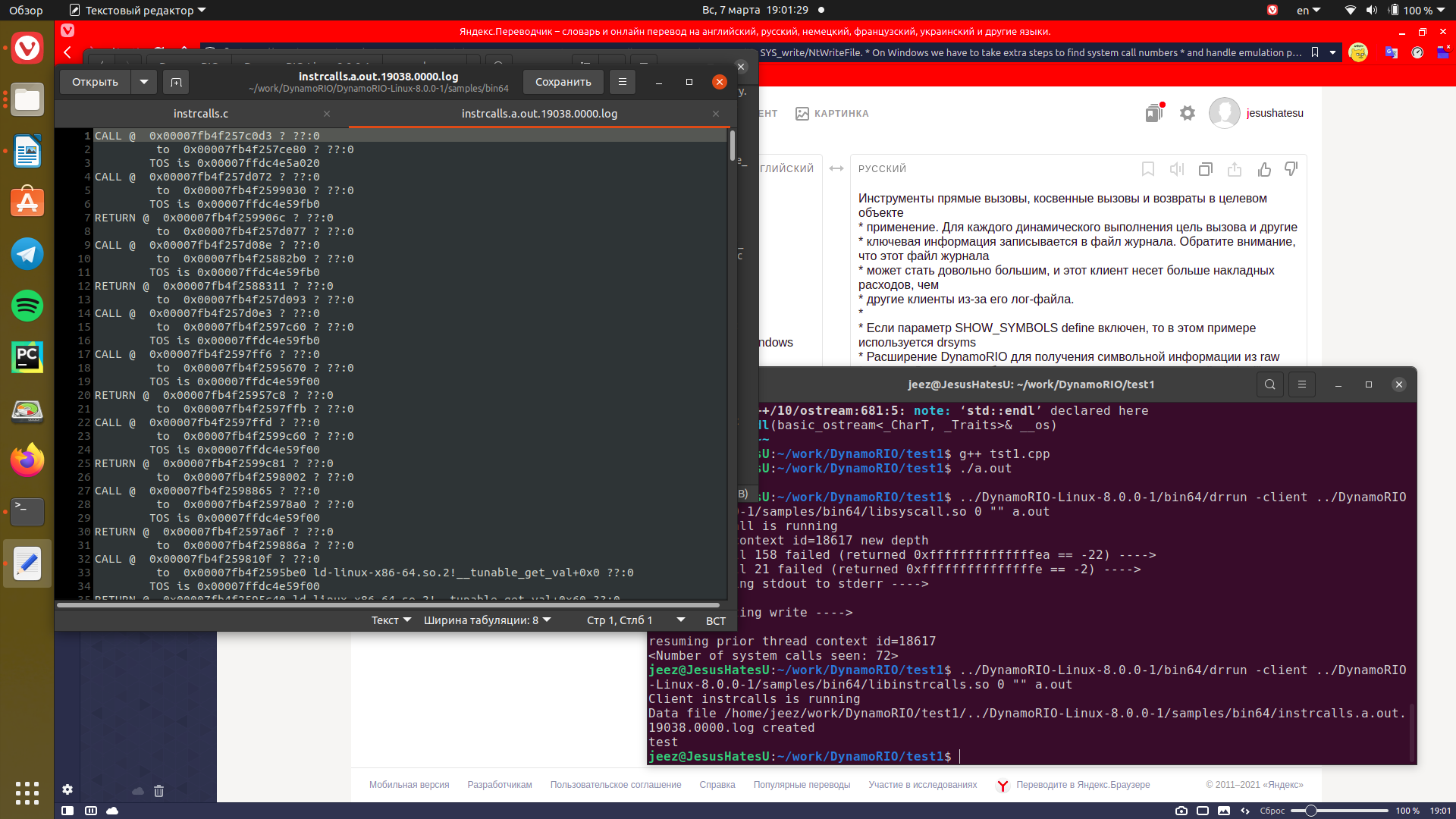
DynamoRIO, как и Intel PIN, хорошо задокументирована и даёт содержательное описание поведения DynamoRIO API, но это скучные мануалы, давайте же перейдём к практике и рассмотрим аналогичные задачи.

Код библиотек-клиентов в DynamoRIO очень сложен для чтения, особенно неподготовленным людям, лично я возился с разбором самого инструмента и одного примера около месяца, при этом до сих пор ничего не понимаю. Поэтому предлагаю пропустить этот этап и посмотреть на саму работу.

Итак, напишем простенькую, максимально незатейливую программку:

Скомпилируем и запустим:

Как видим, всё работает, хоть и написана она довольно криво. Теперь воспользуемся «клиентом» из коробки, который считает инструкции:

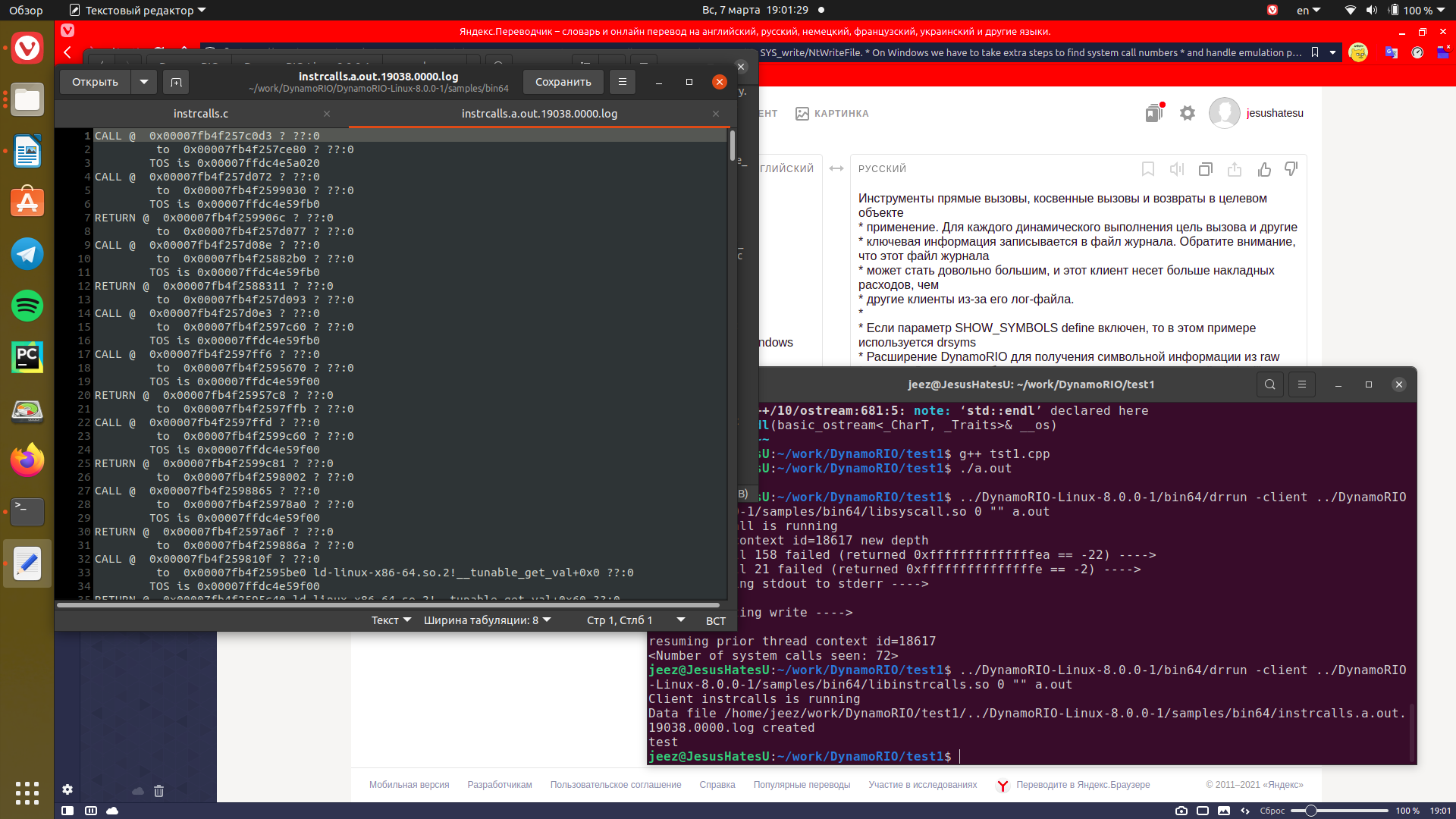


Сразу отмечаем, как запускать DynamoRIO с библиотекой-клиентом:

Запускается, как отмечалось ранее, скрипт drrun, ему подаётся через опцию -client наш клиент, который выполняет логгирование инструкций, затем идут аргументы к скрипту и клиенту, после чего сам исследуемый исполняемый файл.

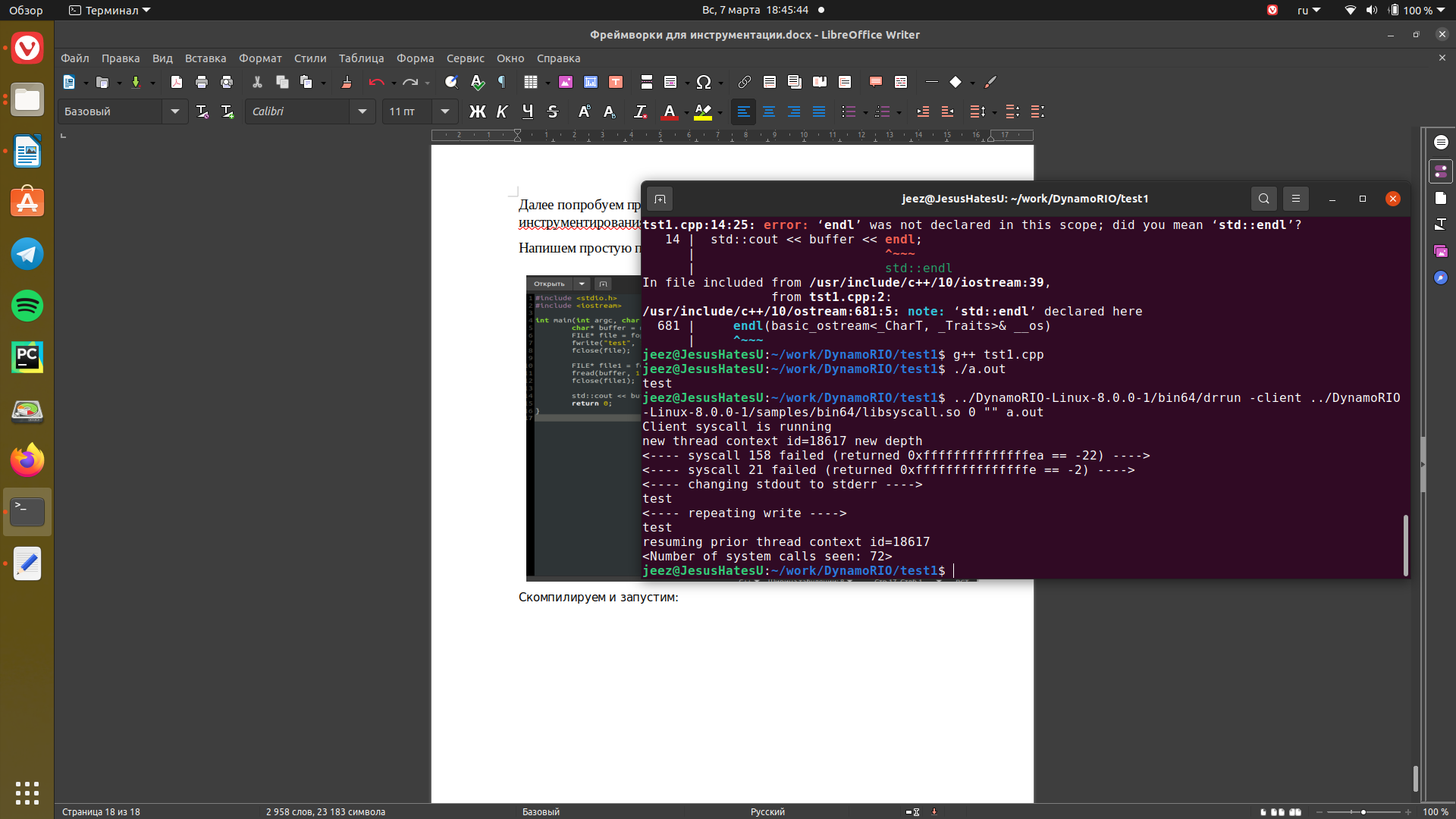
Заметим, что строка «test» вывелась, это как раз и говорит о том, что наша программа отработала нормально под управлением нашего инструмента, так же вывела результат в стандартный поток. При этом создала файл с расширением .log.

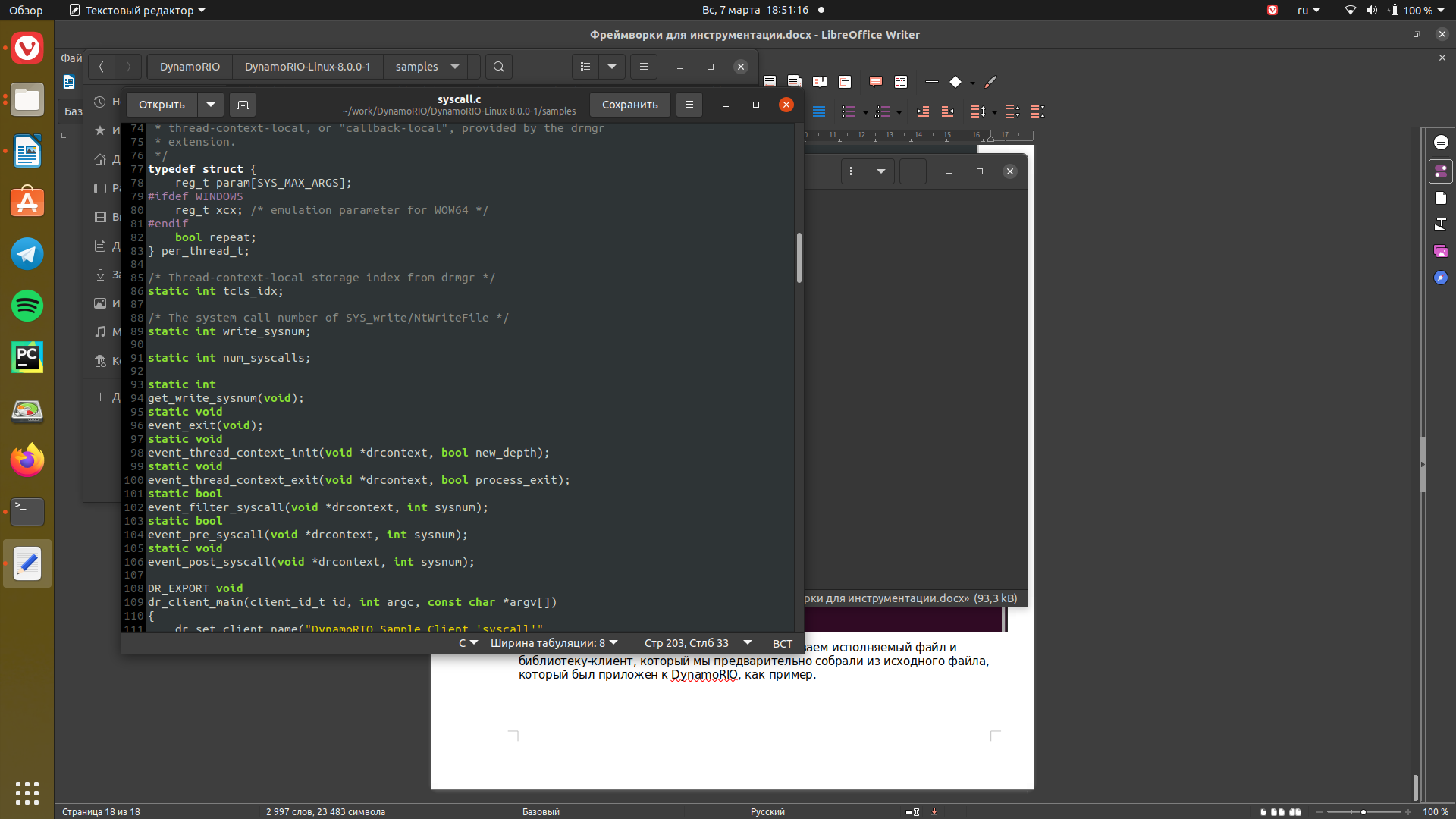
По его названию видно, что там 19038 инструкций, при этом файл с инструкциями выглядит следующим образом:



Это не чистый ассемблерный код, но представление вполне понятное, за исключением некоторых регистров.

Теперь посмотрим на клиента, который считает системные вызовы.

Видим немного другой вывод, сообщения о неудачных завершениях, а так же то, что оказывается наш вывод сообщения «test» был направлен в стандартный вывод ошибок, вероятно из-за маленького размера буфера, после чего он выводится снова. Так же видим количество системных вызовов, что нам и нужно было.

Будет не совсем честным не показать реализацию, поэтому взглянем одним глазком:

Сразу нас встречает множество определений каких-то функций, в которых можно заметить переменную контекста, которая передаёт контекст потока, который принимает сигнал, нужен он для обработки состояния потока, с которым мы сейчас работаем. В этом файле более 3 сотен строк кода, поэтому я показываю лишь часть.



А так выглядят функции регистрации и отмены регистрации потоков, довольно страшно. Сразу понятно, что DynamoRIO не настолько дружелюбен к разработчику, но предоставляет больше средств для контроля и более точной настройки вашей библиотеки.

**Эпилог**