Динамический анализ кода (также известный как «профилирование») наиболее популярен в областях, где требуется выявить hotspots приложения (пиковые с точки зрения производительности участки программы), гонки потоков (race conditions), найти ошибки при работе с памятью в приложении, оценить фактический расход оперативной памяти, а техники Taint Analysis и In-Memory Fuzzing (техники динамического анализа) позволяют определить какие участки программы наиболее подвержены exploit-ам.

В отличие от более распространенного (и простого в исполнении) статического анализа, динамический анализ обладает рядом преимуществ:

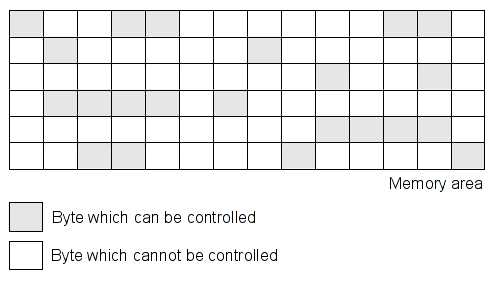
* Для проведения анализа исходный код программы не обязателен
* Наличие runtime информации (содержимое регистров, содержимое ячеек памяти, известны значения переменных окружения)
* Диагностика ошибок в многопоточном коде (таких, как борьба потоков за доступ к общим ресурсам, наличие deadlock-ов)
* Измерение потребляемых программой ресурсов (время исполнения программы или её отдельных частей, число обращений к внешним ресурсам, например, базе данных или файлу)
* Большая точность в диагностике false-positive ошибок: динамический анализатор не пытается предсказывать поведение программы, как это делает статический анализатор, а при запуске программы просто фиксирует наличие ошибки.

К очевидным недостаткам динамического анализа можно отнести:

* Невозможность покрыть 100% кода программы (покрывает только ту часть кода, которая выполняется)
* При наличии исходного кода программы и необходимости внести исправления, найти точное место в коде будет проблематично

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 при сбое — это может быть полезно.



Например,

void foo1 ( const char \* av [])

{

uint32\_t a, b;

a = atoi ( av [ 1 ]);

b = а ;

foo2 ( b );

}

void foo2 ( const char \* av [])

{

uint8\_t \* buffer ;

if ( ! ( buffer = ( uint8\_t \* ) malloc ( 32 \* sizeof ( uint8\_t ))))

return ( - ENOMEM );

buffer [ 2 ] = av [ 1 ] [ 4 ];

buffer [ 12 ] = av [ 1 ] [ 8 ];

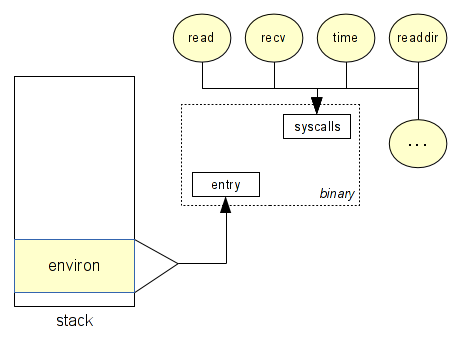
buffer [ 30 ] = av [ 1 ] [ 12 ];

}

В первой функции в начале переменные «a» и «b» не испорчены. Когда вызывается функция atoi, переменная «a» испорчена. Тогда «b» испорчена, если ей присвоено значение «a». Теперь мы знаем, что аргумент функции foo2 может управляться пользователем.

Во второй функции, когда буфер выделяется через malloc, содержимое не испорчено. Затем, когда выделенная область инициализируется пользовательским вводом, нам нужно испортить байты «буфер + 2», «буфер + 12» и «буфер + 30». Позже, когда один из этих байтов читается, мы знаем, что он может контролироваться пользователем.

Для динамического анализа в основном нам нужно определить все вводимые пользователем данные, такие как окружение и системные вызовы. Мы начинаем заражать эти входные данные и распространять/удалять это заражение, когда у нас есть такие инструкции, как GET/PUT, LOAD/STORE.



Для этого нам нужна структура динамического бинарного инструментария. Назначение DBI - добавить обработчик pre / post для каждой инструкции. Когда вызывается обработчик, мы можем получить всю необходимую информацию об инструкции или среде (памяти).

Подготовка

Для грамотного использования бинарной динамической инструментации, а главное использования в нужных местах, необходимо знать устройство программ (kernel и user space 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. Обычно это изучают в университетах на таких предметах, как «Операционные системы», «Компиляторы», «Язык ассемблера» и прочих. Но если не повезло с университетом или преподавателями, в интернете есть множество бесплатного материала по данным темам, например:

* в общих чертах про инструментацию, как статическую, так и динамическую

https://xakep.ru/2013/09/11/61232/

* немного поглубже в DBI

https://habr.com/ru/company/dsec/blog/142575/

* документация по использованию фреймворка для DBI Intel Pin

https://software.intel.com/sites/landingpage/pintool/docs/98314/Pin/html/index.html

* презентация одной конференции с основными терминами

https://docs.huihoo.com/blackhat/usa-2011/BH\_US\_11\_Diskin\_Binary\_Instrumentation\_Slides.pdf

* «если есть проблема, её уже решил какой-нибудь индус с ютуба»

https://youtube.com/playlist?list=PLKwUZp9HwWoDXHo51cWvYz1GRlM4dI5F2

* так как мы инструментируем приложения под линукс, неплохо прочитать книгу обо всём в этой ОС (у меня в программе обучения была эта книга, поэтому её и рекомендую)

Операционная система UNIX, 2-е издание  
Автор: Робачевский А., Немнюгин С., Стесик О.  
Год: 2010

* так же любые видео на ютубе с запросами «как работает компилятор», «как устроена программа на языке С/С++»
* курс по разработке модулей ядра, чтобы научиться самому с ним взаимодействовать

<https://stepik.org/course/2051/promo#toc>

* статьи по ядру GNU/Linux

<https://rtfm.co.ua/linux-tipy-pamyati/> - виртуальная память

<https://rtfm.co.ua/linux-arxitektura-yadra-linux-obshhaya-arxitektura-sistemy/> - архитектура ядра

[https://www.ibm.com/developerworks/ru/library/l-linux-kvm/?mhsrc=ibmsearch\_a&mhq=%D1%8F%D0%B4%D1%80%D0%BE%20linux](https://www.ibm.com/developerworks/ru/library/l-linux-kvm/?mhsrc=ibmsearch_a&mhq=ядро%20linux) - KVM Linux

<https://www.ibm.com/developerworks/ru/library/l-linux_kernel_05/index.html?ca=drs-> - системные вызовы

<https://m.habr.com/ru/company/smart_soft/blog/185226/> - организация памяти процесса

<https://m.habr.com/ru/company/smart_soft/blog/226315/> - как ядро управляет памятью

<https://m.habr.com/ru/company/smart_soft/blog/234239/> - про стек

**Dynamic instrumentation tools**

Существует достаточно много инструментов для DBI, который позволяют в runtime манипулировать программой, её сегментами и используемыми регистрами. 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 каналов. 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.

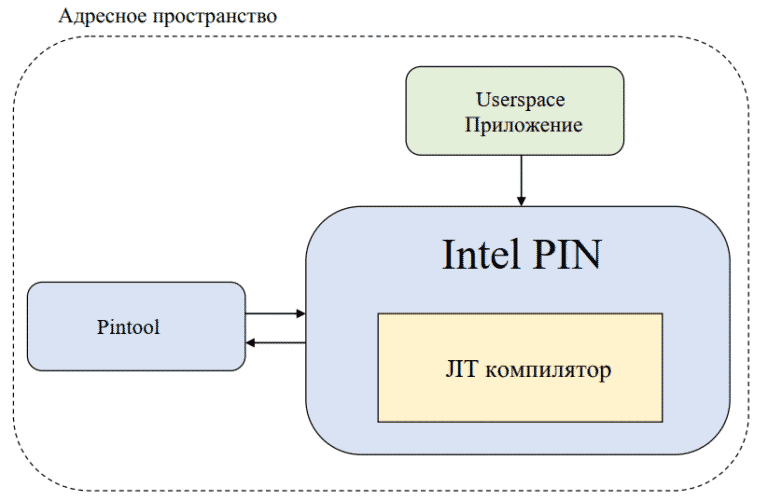
**Intel Pin**

Первое решение, найденное мной и подходящее под задачу (инструментирование приложений Linux без исходного кода) 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, то есть исходный код и перекомпиляция программы не требуются. Pin доступен для ОС Linux, Windows и OS X, также в официальных источниках можно встретить версию Pin для Android.

Инструменты анализа приложений, которые используют Pin API называются Pintools. Для разработки собственного инструмента Intel предоставляет набор разработчика, называемый Pin kits, куда также входят и исходные коды множества инструментов.

Pintool представляет из себя скомпилированный бинарный файл. Для Linux-систем — это разделяемая библиотека с расширением .so, для Windows – динамическая библиотека с расширением .dll. Взаимодействие Intel Pin, Pintool и анализируемой программы можно изобразить в виде схемы:



Intel Pin умеет работать в двух режимах — JIT и Probe. JIT (just-in-time) режим более функциональный, но менее производительный (наблюдается существенное замедление относительно исходной программы). Probe режим имеет более ограниченную функциональность, но производительность (связки ‘исходная программа’ + ‘Pintool’) приближается к производительности исходной программы.

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

##### **Применение Intel PIN для различных сценариев диагностики уязвимостей ПО**

Intel Pin позволяет реализовать различные техники динамического анализа кода с целью диагностики потенциальных уязвимостей ПО. Приведу несколько сценариев из опыта моей команды:

* Taint Analysis: техника выявления потенциально уязвимых к пользовательскому вводу участков кода с целью обезопасить ПО от использования этих участков в интересах злоумышленников (защита ПО от так называемых exploit-ов). Базируется на идее присвоения каждому объекту в коде (переменной) специальной метки (тега), при условии, что значение этого объекта получено из ненадежного источника, т.е. от пользователя, из сети, из файла. Во время исполнения программы строится так называемое taint tree – метки сливаются, распространяются на другие объекты или удаляются в зависимости от инструкций GET/PUT, LOAD/STORE. Intel Pin позволяет добавить обработчики для каждой такой инструкции. На основе анализа taint tree возможно определить какие части программы потенциально подвержены exploit-ам, что позволяет лучше защитить такие участки от вредоносного воздействия. Если пользовательский ввод — это конфиденциальные сведения, то возможно сделать вывод о жизненном цикле такого типа данных внутри программы.
* Dynamic Symbolic Execution (или Consolic Execution): техника, применяемая для динамического покрытия кода для подтверждения того, что все ветви кода достижимы и работоспособны. Графически эту задачу можно представить как обход бинарного дерева, где узлы — это условные инструкции (conditional statements, т.е. if операторы), а ребра – последовательность не-условных инструкций (non-conditional statements), т.е. ветви кода. Очевидно, что при первом проходе любой из ветвей кода, значения входных параметров приведут либо к выполнению (true) или к невыполнению условия (false). Для следующего прохода вычисляются значения переменных, при которых условие будет противоположным. Задача Intel Pin добавить соответствующие обработчики для условных инструкций в коде. Фактически, применив данную технику вы покрываете весь код вашей программы с помощью test suite, который сгенерирован автоматически.
* In-Memory Fuzzing: техника тестирования потенциально уязвимых к пользовательскому вводу участков кода. Основная идея техники базируется на представлении, что почти любая программа будет сломана, если входные данные для неё будут подготовлены случайным образом. В начале выбирается участок кода программы, который необходимо протестировать. Затем определяется множество (диапазон) возможных значений входных параметров. Далее, в случае blackbox тестирования (когда ничего не известно об объекте тестирования) генерируется случайный набор входных параметров из возможного диапазона, для graybox (известна только часть) или whitebox (известно все) тестирования модифицируются известные входные наборы. Теперь участок кода исполняется и, если он приводит к непредсказуемому поведению, т.е. вызывает exception, то входной набор параметров запоминается и результаты анализируются. Далее шаги повторяются для нового набора входных данных. С помощью Intel Pin задача решается довольно просто: добавляются breakpoints до и после тестируемого участка кода; после прохождения первого breakpoint сохраняется контекст и отслеживаются все exceptions в коде. В случае, если произошел exception, набор входных параметров сохраняется и контекст восстанавливается, в случае если exception не произошел — просто восстанавливается контекст. Данная техника позволяет получить устойчивый к пользовательскому вводу код. Один из вариантов применения техники на практике – тестирование parser-а бинарного файла.

**Простой пример использования Pin**

В этом первом примере мы испортим область памяти "чтение" и увидим краткий обзор API Pin. Для этого первого испытания мы будем:

1. Ловить sys\_read syscall.

2. Получим второй и третий аргумент для taint области.

3. Вызовем хэндлер, когда у нас есть такая инструкция, как LOAD или STORE в этой области.

Мы будем работать над этим следующим кодом, функция foo выполняет простые LOAD и STORE инструкции.

void foo(char \*buf){

char a;

a = buf[0];

a = buf[4];

a = buf[8];

a = buf[10];

buf[5] = 't';

buf[10] = 'e';

buf[20] = 's';

buf[30] = 't';

}

int main(int ac, char \*\*av)

{

int fd;

char \*buf;

if (!(buf = malloc(256)))

return -1;

fd = open("./file.txt", O\_RDONLY);

read(fd, buf, 256), close(fd);

foo(buf);

}

С помощью Pin можно добавить обработчик pre и post при возникновении syscall. Для этого нам просто нужно инициализировать функцию обратного вызова.

typedef VOID(\* LEVEL\_PINCLIENT::SYSCALL\_ENTRY\_CALLBACK)(THREADID threadIndex, CONTEXT \*ctxt, SYSCALL\_STANDARD std, VOID \*v);

typedef VOID(\* LEVEL\_PINCLIENT::SYSCALL\_EXIT\_CALLBACK)(THREADID threadIndex, CONTEXT \*ctxt, SYSCALL\_STANDARD std, VOID \*v);

VOID LEVEL\_PINCLIENT::PIN\_AddSyscallEntryFunction(SYSCALL\_ENTRY\_CALLBACK fun, VOID \*val);

VOID LEVEL\_PINCLIENT::PIN\_AddSyscallExitFunction(SYSCALL\_EXIT\_CALLBACK fun, VOID \*val);

В нашем случае мы используем клиент LEVEL\_PINCLIENT::PIN\_AddSyscallEntryFunction. Когда происходит системный вызов, мы проверяем, является ли системный вызов доступным для чтения. Затем мы сохраняем второй и третий аргументы, которые описывают нашу область памяти.

struct range

{

UINT64 start;

UINT64 end;

};

std::list<struct range> bytesTainted;

VOID Syscall\_entry(THREADID thread\_id, CONTEXT \*ctx, SYSCALL\_STANDARD std, void \*v)

{

struct range taint;

if (PIN\_GetSyscallNumber(ctx, std) == \_\_NR\_read){

/\* Get the second argument \*/

taint.start = static\_cast<UINT64>((PIN\_GetSyscallArgument(ctx, std, 1)));

/\* Get the third argument \*/

taint.end = taint.start + static\_cast<UINT64>((PIN\_GetSyscallArgument(ctx, std, 2)));

/\* Add this area in our tainted bytes list \*/

bytesTainted.push\_back(taint);

/\* Just display information \*/

std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" << taint.start \

<< " to 0x" << taint.end << " (via read)"<< std::endl;

}

}

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

{

/\* Init Pin arguments \*/

if(PIN\_Init(argc, argv)){

return Usage();

}

/\* Add the syscall handler \*/

PIN\_AddSyscallEntryFunction(Syscall\_entry, 0);

/\* Start the program \*/

PIN\_StartProgram();

return 0;}

Теперь нам нужно поймать все инструкции, которые читают (ЗАГРУЖАЮТ) или записывают (ХРАНЯТ) в зараженной области. Для этого мы добавим функцию, вызываемую каждый раз, когда делается доступ к этой области.

Для этого мы добавим мастер-обработчик, вызываемый для каждой инструкции.

typedef VOID(\* LEVEL\_PINCLIENT::INS\_INSTRUMENT\_CALLBACK)(INS ins, VOID \*v); VOID LEVEL\_PINCLIENT::INS\_AddInstrumentFunction(INS\_INSTRUMENT\_CALLBACK fun, VOID \*val);

Затем в главном обработчике нам нужно найти инструкцию LOAD / STORE, например mov rax, [rbx] или mov [rbx], rax.

if (INS\_MemoryOperandIsRead(ins, 0) && INS\_OperandIsReg(ins, 0)){

INS\_InsertCall(

ins, IPOINT\_BEFORE, (AFUNPTR)ReadMem,

IARG\_ADDRINT, INS\_Address(ins),

IARG\_PTR, new string(INS\_Disassemble(ins)),

IARG\_MEMORYOP\_EA, 0,

IARG\_END);

}

else if (INS\_MemoryOperandIsWritten(ins, 0)){

INS\_InsertCall(

ins, IPOINT\_BEFORE, (AFUNPTR)WriteMem,

IARG\_ADDRINT, INS\_Address(ins),

IARG\_PTR, new string(INS\_Disassemble(ins)),

IARG\_MEMORYOP\_EA, 0,

IARG\_END);

}

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

{

...

INS\_AddInstrumentFunction(Instruction, 0);

...

}

Как вы можете видеть выше, мы делаем некоторые проверки, прежде чем вставить вызов. Если инструкции второго операнда считываются в память и если первый операнд является регистром. То есть если инструкция выглядит как " mov reg, [r/imm]", она вызывает функцию ReadMem. При вызове этой функции она передает несколько сведений, таких как: адрес инструкции, дизассемблирование и адрес считываемой памяти. То же самое с инструкциями STORE.

Теперь нам просто нужно проверить, находится ли доступная память в зараженной области. Для нашего обратного вызова чтения памяти у нас есть что-то вроде этого:

VOID ReadMem(UINT64 insAddr, std::string insDis, UINT64 memOp)

{

list<struct range>::iterator i;

UINT64 addr = memOp;

for(i = bytesTainted.begin(); i != bytesTainted.end(); ++i){

if (addr >= i->start && addr < i->end){

std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr

<< ": " << insDis<< std::endl;

}

}

}

Конечный результат выглядит примерно так:

$ ../../../pin -t ./obj-intel64/Taint\_ex1.so -- ./test\_ex1

[TAINT] bytes tainted from 0x665010 to 0x665110 (via read)

[READ in 665010] 400620: movzx eax, byte ptr [rax]

[READ in 665014] 40062a: movzx eax, byte ptr [rax+0x4]

[READ in 665018] 400635: movzx eax, byte ptr [rax+0x8]

[READ in 66501a] 400640: movzx eax, byte ptr [rax+0xa]

[WRITE in 665015] 40064f: mov byte ptr [rax], 0x74

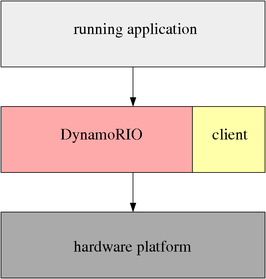
[WRITE in 66501a] 40065a: mov byte ptr [rax], 0x65

[WRITE in 665024] 400665: mov byte ptr [rax], 0x73

[WRITE in 66502e] 400670: mov byte ptr [rax], 0x74

**DynamoRIO**

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.

DynamoRIO также можно использовать в качестве сторонней библиотеки дизассемблинга.

После распаковки содержимого дистрибутива DynamoRIO (см. раздел Содержимое дистрибутива) настройка и выполнение приложений в рамках DynamoRIO обрабатываются набором библиотек и инструментов. В Linux инструментами являются скрипты drconfig, drrun и drinject.

При использовании DynamoRIO в качестве сторонней библиотеки дизассемблирования развертывание не требуется, так как DynamoRIO не управляет целевым приложением при использовании в качестве обычной библиотеки.

Общие Мероприятия

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

* Создание или удаление базовых блоков и трассировок
* 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. Заголовочный файл dr\_events.h содержит объявления для всех процедур регистрации и отмены регистрации.

Обратим внимание, что клиентам разрешено регистрировать несколько обратных вызовов для одного и того же события. DynamoRIO также поддерживает несколько клиентов, каждый из которых может зарегистрироваться на одно и то же событие. В этом случае DynamoRIO выстраивает в последовательность обратные вызововы событий в обратном порядке, когда они были зарегистрированы. Другими словами, первый зарегистрированный обратный вызов получает уведомление о событии последним. Эта схема отдает приоритет обратному вызову, зарегистрированному ранее, поскольку она может переопределять или изменять действия клиентов, зарегистрированных позже. Так же DynamoRIO вызывает процедуру dr\_init() каждого клиента в соответствии с приоритетом клиента.

Системы, регистрирующие несколько обратных вызовов для одного события, должны знать, что изменения клиента видны в последующих обратных вызовах. DynamoRIO не делает никаких попыток уменьшить помехи между функциями обратного вызова. Клиент несет ответственность за обеспечение совместимости между своими функциями обратного вызова и функциями обратного вызова других клиентов.

Клиенты также могут отменить регистрацию обратного вызова с помощью соответствующей процедуры отмены регистрации (см. dr\_events.h). Хотя это необычно, одна процедура обратного вызова может отменить регистрацию другой. В этом случае DynamoRIO по-прежнему вызывает подпрограммы, которые были зарегистрированы до события. Отмена регистрации вступает в силу до следующего события.

В Linux exec (SYS\_execve) не приводит к событию exit, но это приводит к перезагрузке клиентской библиотеки и повторному вызову ее подпрограммы dr\_init (). События системного вызова можно использовать для уведомления о SYS\_execve.

DynamoRIO предоставляет клиентам мощную библиотеку утилит для пользовательских преобразований кода среды выполнения. Интерфейс включает в себя явную поддержку создания прозрачных клиентов. DynamoRIO предоставляет общие ресурсы, которые клиенты могут использовать, чтобы избежать зависимости от общих библиотек, которые могут использоваться приложением. Клиент должен использовать внешние ресурсы только через собственный API DynamoRIO, через расширения DynamoRIO, через прямые системные вызовы или через внешнего агента в отдельном процессе, который взаимодействует с клиентом. Сторонние библиотеки можно использовать, если они связаны статически или загружены частным образом и нет возможности глобальных конфликтов ресурсов (например, выделение памяти сторонней библиотеки должно быть обернуто).

Dynamorio API предоставляет:

* Распределение памяти: как thread-private (быстрее, так как оно не требует затрат на синхронизацию), так и thread-shared
* Локальное хранилище потоков
* Поток-локальный стек, отдельный от стека приложений
* Простые мьютексы
* Создание, чтение и запись файлов
* Запрос адресного пространства
* Итератор прикладного модуля
* Идентификация функций процессора
* Создание дополнительной нити

Другой класс утилит, предоставляемых DynamoRIO, — это структуры и процедуры для декодирования, кодирования и манипулирования инструкциями IA-32 и AMD64.

Чтобы упростить доступ к 64-битному адресному пространству, DynamoRIO гарантирует, что все его code caches и heap находятся в одной и той же области памяти объемом 2 ГБ. DynamoRIO также загружает клиентские библиотеки в пределах 32-битной достижимости своих code caches и heap (что в настоящее время требует, чтобы клиентские библиотеки имели предпочтительные базы в нижних 2 ГБ адресного пространства). Это означает, что любые статические данные или код в клиентской библиотеке или любые данные, выделенные с помощью Dynamorio API, гарантированно будут доступны непосредственно из code cache.

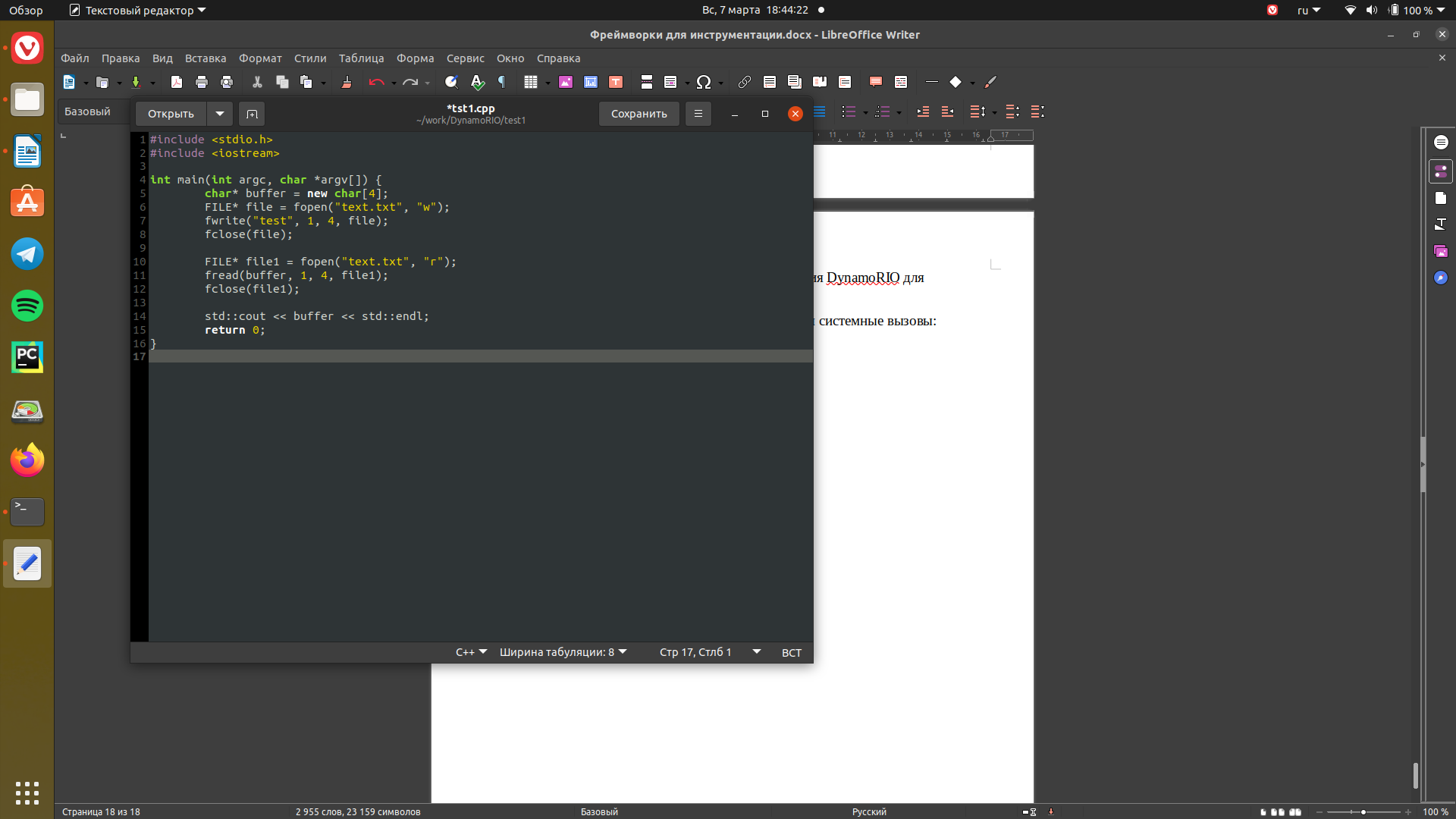
Поведение DynamoRIO может быть точно настроено с помощью параметров времени выполнения. Параметры задаются через скрипты drconfig и drrun в Linux.

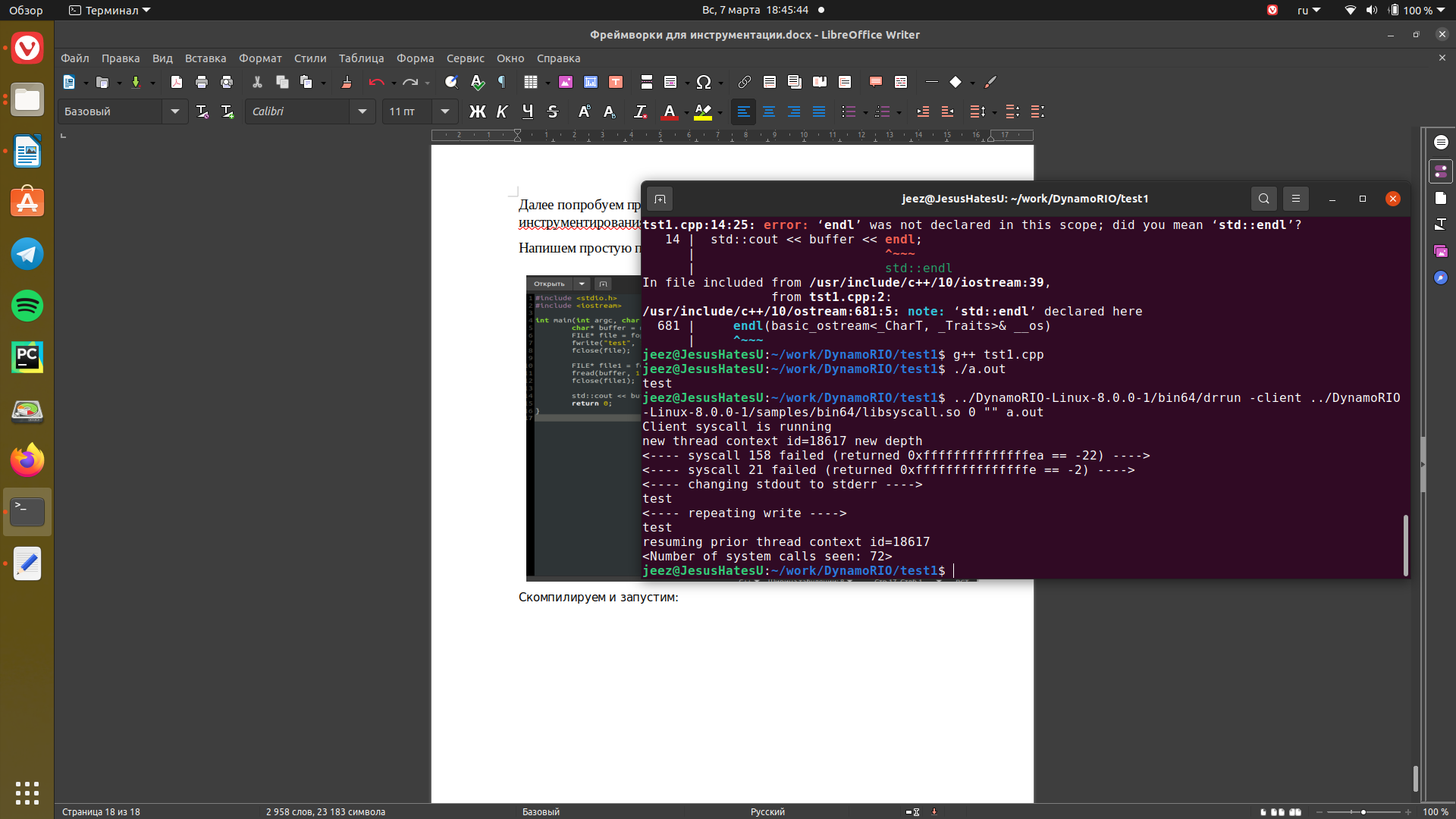
Так же есть примеры от создателей по использованию DynamoRIO API для создания клиентов.

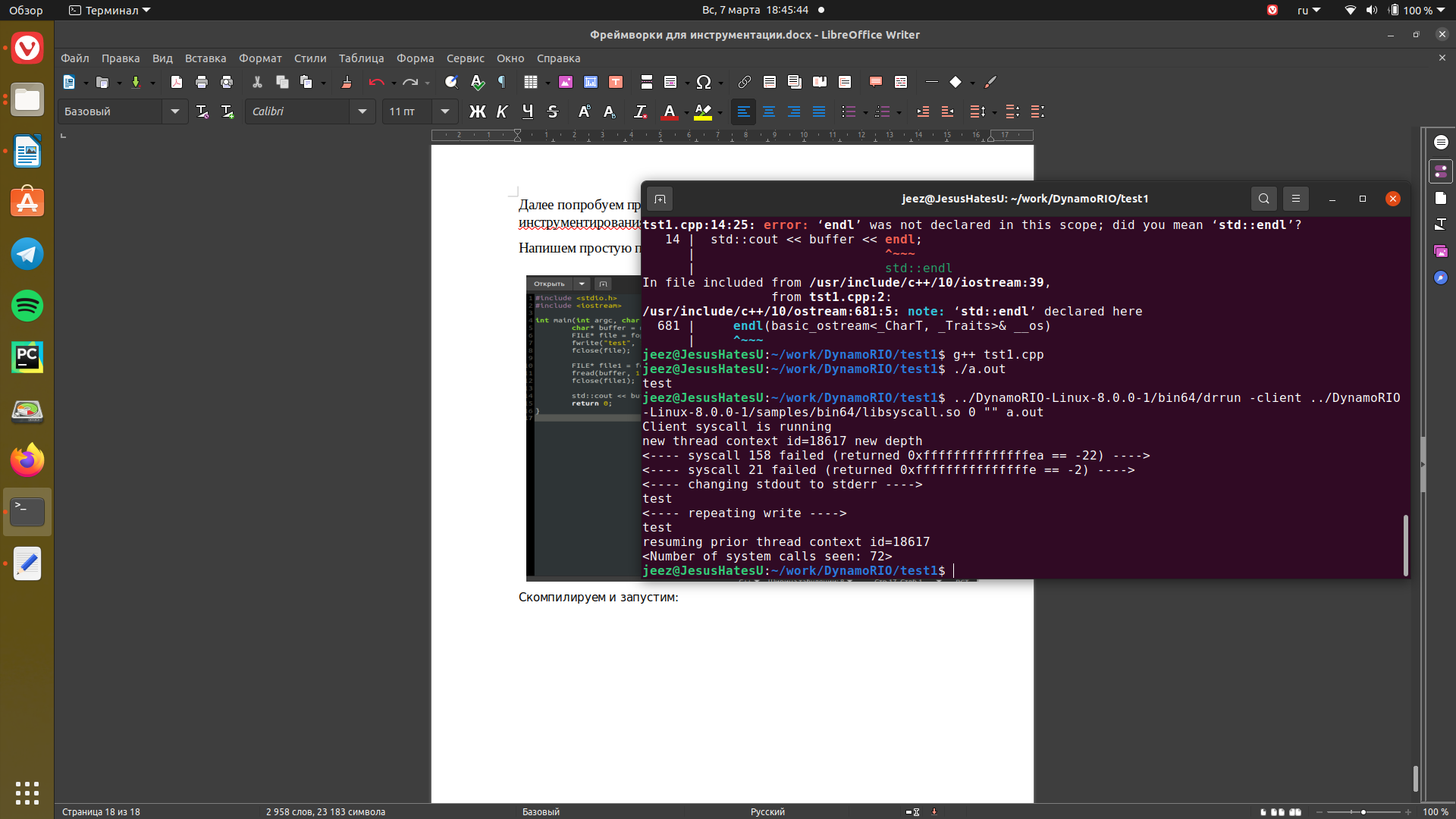
<https://dynamorio.org/docs/API_samples.html>

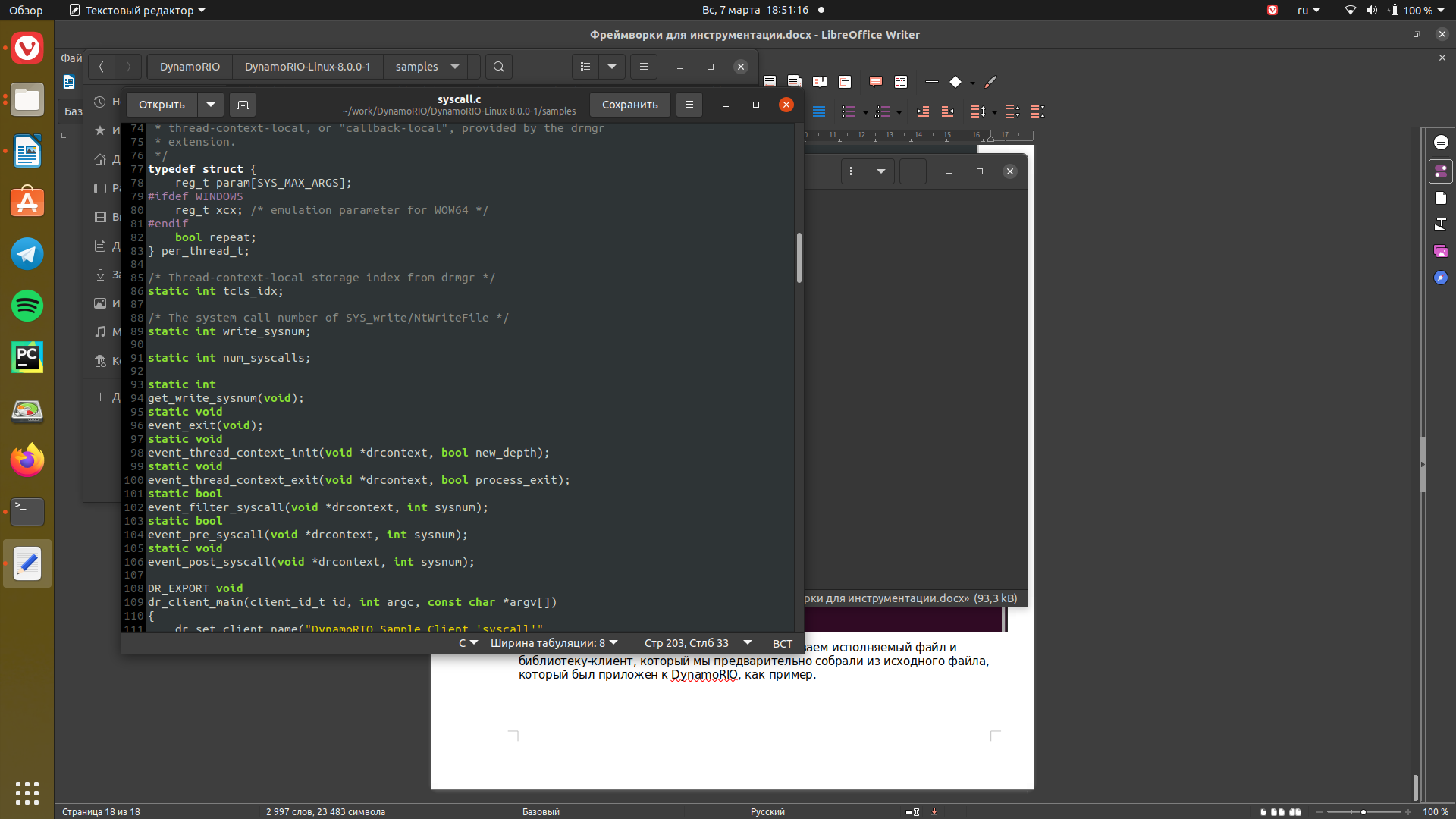
Далее попробуем привести пример использования DynamoRIO для инструментирования.

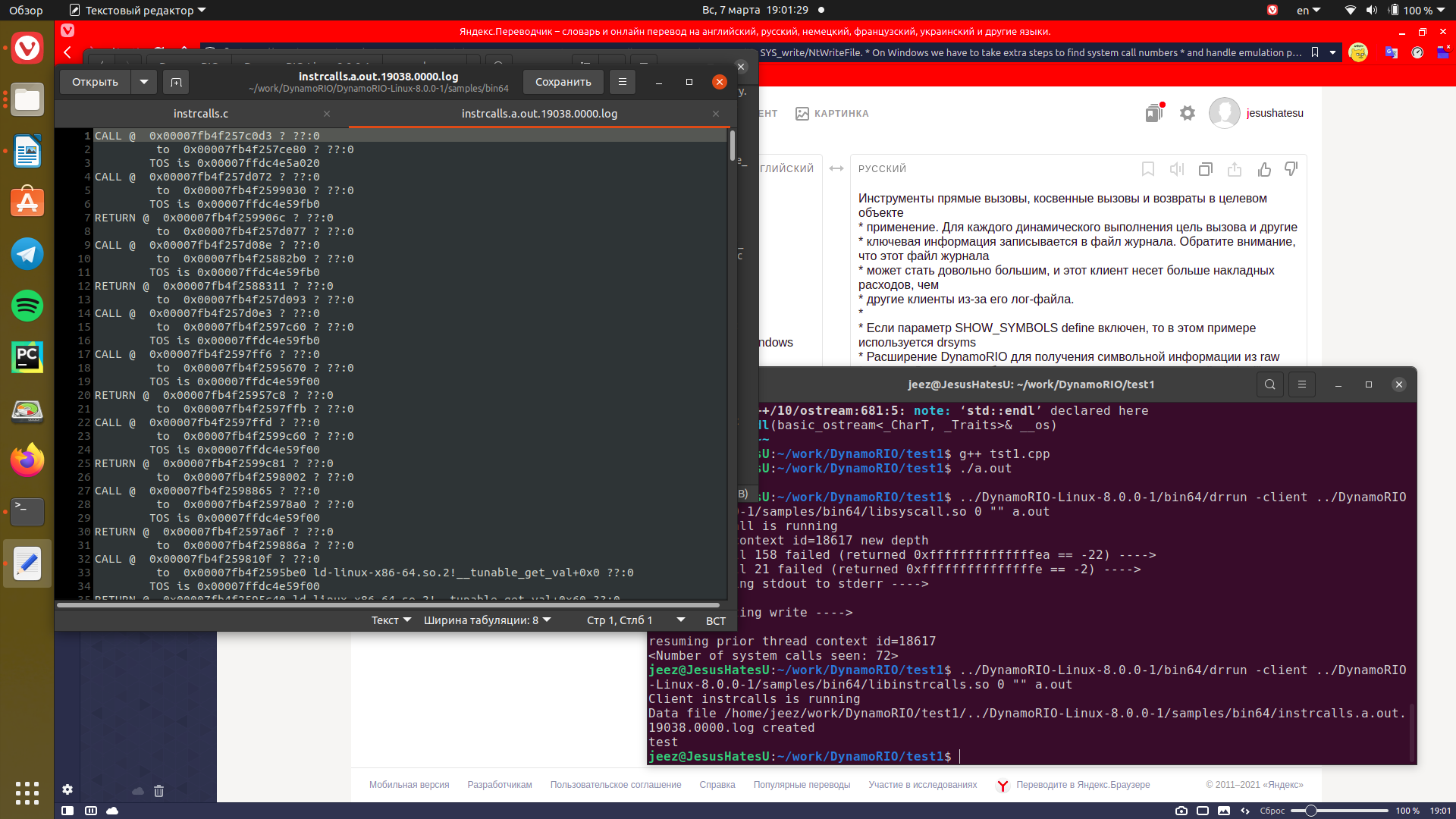
Напишем простую программу, где используются системные вызовы:

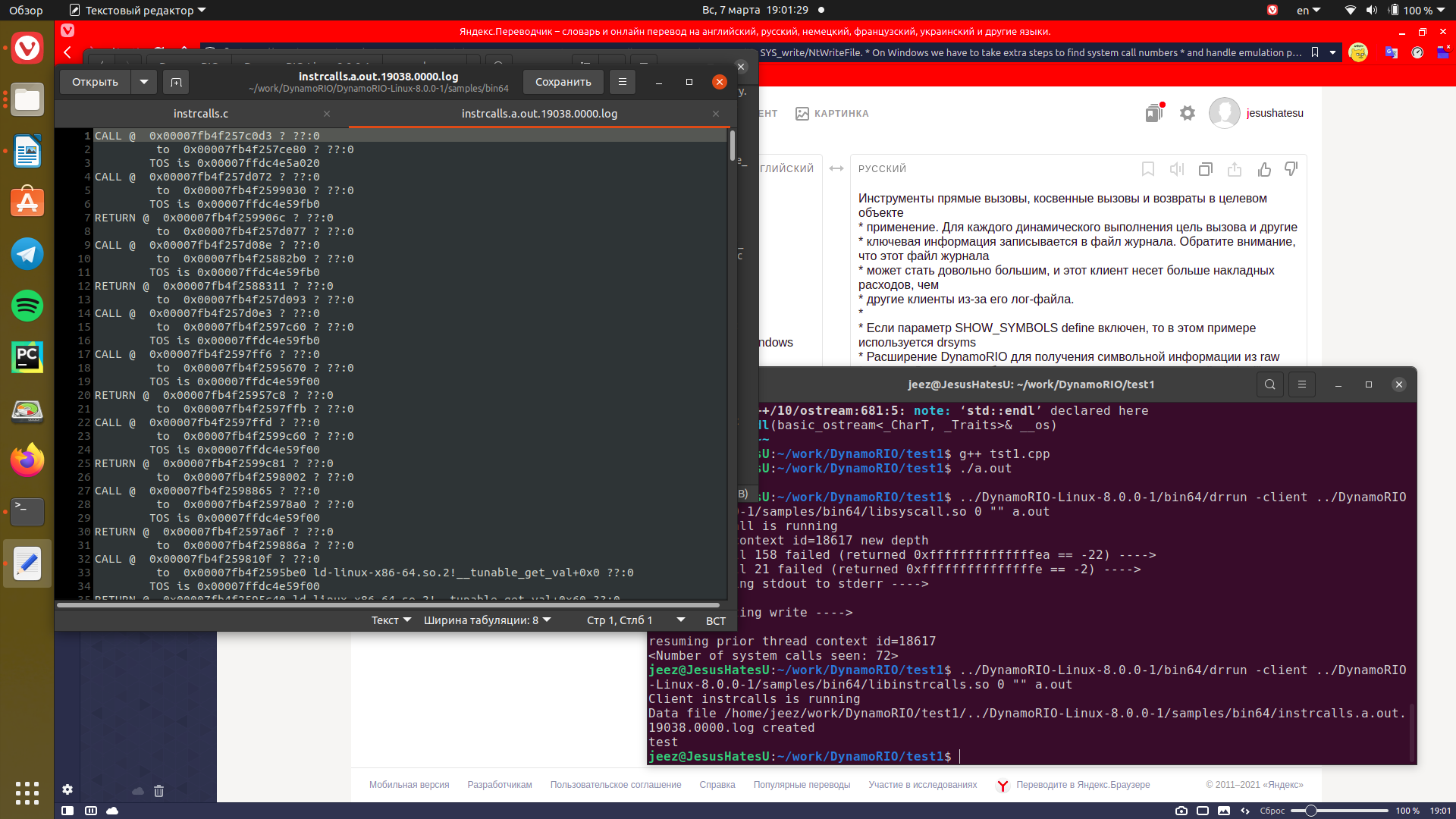
Скомпилируем и запустим:

Как видим, всё работает. Теперь воспользуемся «клиентом» из коробки, который считает количество системных вызовов:

Как видно, он считает системные вызовы. Запускать надо через файл drrun, в который закидываем исполняемый файл и библиотеку-клиент, который мы предварительно собрали из исходного файла, который был приложен к DynamoRIO, как пример. Выглядит он следующим образом.

Так же есть пример клиента, ведущего лог вызова всех инструкций:

В итоге получается такой очень большой файл с логами:



**Syzcaller**

Фаззинг-это автоматизированный способ ввода случайных значений в программное обеспечение в надежде обнаружить ошибки, вызванные проблемными комбинациями входных данных, которые разработчики ранее не пробовали.

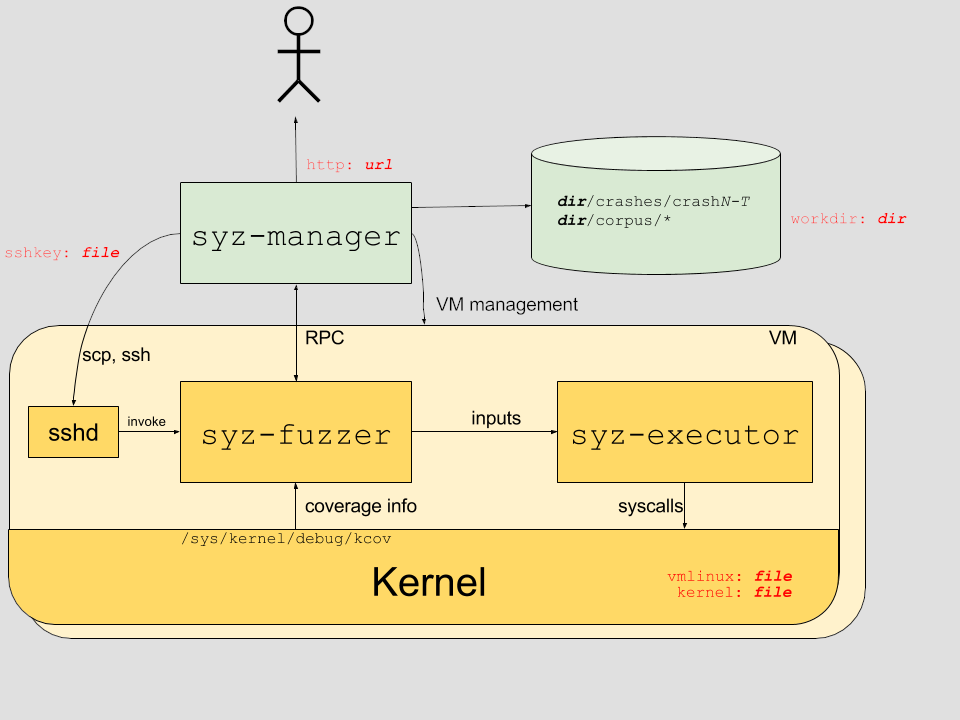
Слепая генерация тестовых входов также приводит к очень большому пространству возможных значений для проверки, большинство из которых можно тривиально доказать, что они правильно обработаны, поэтому вместо случайного тестирования из набора всех возможных входов мы можем ограничить наш набор, предоставив описание того, как предполагается использовать аргументы системных вызовов. Делая это, мы можем обеспечить рандомизированное тестирование по большему набору вызовов с помощью доступных ресурсов.

Другой метод оптимизации фаззинга называется управляемым фаззингом покрытия. Он измеряет охват кода, достигаемый каждым входным сигналом, который он дает, и пытается максимизировать охват кода входными сигналами, которые он генерирует. Каждый вход, который увеличивает охват, мутируется, чтобы увеличить общий охват тестирования (т. е. получить доступ к большему количеству функций, ветвей). Набор входных данных для цели фаззинга называется корпусом.

Syzkaller-это неконтролируемый фаззер ядра, который использует оба описанных выше метода для применения фаззинга к системным вызовам. Рассмотрим, как его использовать и изменить для тестирования приложений.

В упрощенном виде syscaller имеет “manager software” для управления тестовой системой. Он порождает множество виртуальных машин с фаззерами внутри них, которые генерируют и запускают небольшие программы, которые вызывают системные вызовы, используя вышеупомянутые рандомизированные значения, чтобы фаззировать их. Используя RPC, виртуальные машины передают достигнутое покрытие и любую информацию трассировки, полученную в результате фаззинга, хост-машине. Менеджер хранит эту информацию в локальной базе данных и предоставляет веб-интерфейс, в котором вы можете перемещаться по этой информации. Вместо виртуальных машин можно использовать и реальные.

Вероятно, самым особенным богатством syzkaller является коллекция определений системных вызовов, поскольку это ядро инструмента. Он предоставляет простой язык для описания их в шаблонах системных вызовов. Например, давайте посмотрим, как работает системный вызов open()на man-страницах:



int open(const char \*pathname, int flags, mode\_t mode);

И как в исходникак syzcaller:

open(file ptr[in, filename], flags flags[open\_flags], mode flags[open\_mode]) fd

Теперь рассмотрим каждый параметре:

* **fileptr[in, filename]:** первый аргумент, называемый file, является входным указателем, содержащим строку filename.
* **flags flags[open\_flags]:** аргумент flags — это любой из флагов, определенных в массиве open\_flags

**open\_flags** = O\_WRONLY, O\_RDWR, O\_APPEND, ...

* **mode flags[open\_mode]:** аргумент mode — это любой из флагов, определенных в массиве open\_mode

**open\_mode** = S\_IRUSR, S\_IWUSR, S\_IXUSR, ...

* **fd**: возвращаемое значение будет сохранено здесь, чтобы позже использоваться в других описаниях syscall. fd — это специальный тип, используемый для файловых дескрипторов. Он содержит значение, возвращаемое open, но, если ему не присвоено значение, он будет содержать -1.

Возвращаемое значение, хранящееся в fd, затем может быть использовано следующими системными вызовами:

read(fd fd, buf buffer[out], count len[buf])

write(fd fd, buf buffer[in], count len[buf])

Можно так же, вместо использования **fd**, фаззить случайными целыми положительными значениями в диапазоне от 0 до 5000000 с int64[0:5000000]. Но это может быть неэффективным, так как это, скорее всего, может достигнут большего количества чтения/записи кода ядра из/в фактически открытый файл, а не случайный несуществующий файловый дескриптор. **Buf** — это массив произвольного размера случайных значений int8. Это может быть входной аргумент или выходной, в зависимости от значения in/out. Наконец **len[buf]** будет соответствовать длине аргумента **buf**.

Такие определения будут использоваться фаззером для создания небольших программ. Он также будет экспортировать текстовые представления вызова и аргументы, подобные этому:

r0 = open(&(0x7f0000000000)="./file0", 0x3, 0x9)

read(r0, &(0x7f0000000000), 42)

close(r0)

**&(0x7f0000000000)** - это адрес памяти, используемый в этом аргументе указателя, а "**./file0**" - это строковая интерпретация данных в этом адресе.

Затем эти описания будут использоваться фаззером для выполнения системных вызовов внутри виртуальных машин. Инструмент будет генерировать множество таких маленьких программ, всегда стремясь увеличить охват кода.

Теперь давайте установим инструмент и начнем использовать его для улучшения нашей кодовой базы.

Ожидается, что исходный код ядра будет найден в $KSRC директории.

Ядро должно быть специально настроено для повышения производительности syzkaller и для того, чтобы его было должным образом фаззить. Необходимы следующие конфигурации:

CONFIG\_KCOV=y

CONFIG\_KCOV\_INSTRUMENT\_ALL=y

CONFIG\_KCOV\_ENABLE\_COMPARISONS=y

CONFIG\_DEBUG\_FS=y

CONFIG\_CONFIGFS\_FS=y

CONFIG\_SECURITYFS=y

Чтобы иметь возможность перемещаться по покрытию кода в веб-интерфейсе, CONFIG\_DEBUG\_INFO также должен быть включен. После изменения конфигурации снова собираем ядро, так как на следующих шагах нам понадобится скомпилированное ядро. Кроме того, необходимо иметь следующие инструменты:

* Go
* QEMU
* git
* OpenSSH
* Компилятор C с поддержкой покрытия (должны работать как текущие версии GCC, так и Clang)

Для получения набора инструментов Go и исходного кода syzkaller:

go get golang.org/dl/go1.12

go get -u -d github.com/google/syzkaller

Теперь Syzkaller находится в не очень удобном месте, но это можно решить с помощью символической ссылки или переменной PATH. В этом примере он будет определяться следующей переменной:

export SYZPATH=~/go/src/github.com/google/syzkaller

Базовый rootfs для виртуальных машин может быть создан с помощью скрипта, предоставленного syzkaller, который требует пакета debootstrap. Выберите каталог, в котором будут храниться образы rootfs, а затем запустим:

export IMAGES=$(pwd)

bash $SYZPATH/tools/create-image.sh

Таким образом, он создаст базовый образ Debian с необходимыми инструментами и SSH-сервером для копирования и запуска команд на виртуальных машинах.

Просто чтобы убедиться, что все работает нормально и избежать некоторых неполадок в будущем, проверьте, правильно ли работает ssh. Во - первых, загрузите машину:

qemu-system-x86\_64 \

-kernel $KSRC/arch/x86/boot/bzImage \

-append "console=ttyS0 root=/dev/sda debug earlyprintk=serial slub\_debug=QUZ"\

-hda $IMAGES/stretch.img \

-net user,hostfwd=tcp::10021-:22 -net nic \

-enable-kvm \

-nographic \

-m 2G \

-smp 2 \

-pidfile vm.pid \

2>&1 | tee vm.log

В хост-системе попробуем получить к ней доступ:

ssh -i $IMAGES/stretch.id\_rsa -p 10021 -o "StrictHostKeyChecking no" root@localhost

Когда закончии, выключаем:

kill $(cat $IMAGES/vm.pid)

Теперь у нас есть рабочий rootfs, мы почти готовы. Соберём syzkaller и его инструменты:

cd $SYZPATH/

make -j8

Нам нужно создать рабочий каталог, где инструмент создаст базу данных и сохранит результаты. Для этого просто создаем разные каталоги и устанавливаем тот, который хотим использовать в конфигурационном файле.

Чтобы определить поведение инструмента, создим где-нибудь файл конфигурации, называемый здесь config.cfg. Этого примера должно быть достаточно для начального запуска:

{

"target": "linux/amd64",

"http": "127.0.0.1:56741",

"workdir": "$SYZPATH/workdir",

"kernel\_obj": "$KSRC",

"image": "$IMAGES/stretch.img",

"sshkey": “$IMAGES/stretch.id\_rsa",

"syzkaller": "$SYZPATH",

"procs": 8,

"type": "qemu",

"vm": {

"count": 2,

"kernel": "$KSRC/arch/x86/boot/bzImage",

"cpu": 2,

"mem": 2048

}

}

Содержимое вышеприведенного файла выглядит следующим образом:

* target: операционная система/архитектура, для фаззинга
* http: IP-адрес и порт, где будет открыт веб-интерфейс syzkaller
* workdir: рабочий каталог, который будет использоваться, как описано выше
* kernel\_obj: каталог исходников ядра
* image: образ дистрибутива bootstrap
* sshkey: SSH-ключ, используемый для доступа к виртуальным машинам
* syzkaller: путь к исходникам syzkaller
* procs: количество параллельных тестов внутри каждой виртуальной машины
* type: гипервизор виртуальной машины/используемое устройство
* vm: конфигурация виртуальной машины:
* count: количество виртуальных машин
* kernel: образ ядра для фаззинга
* cpu: количество ядер, имитируемых в каждой виртуальной машине
* mem: размер оперативной памяти виртуальной машины

По умолчанию будут фаззиться все системные вызовы (потом это изменим). Теперь мы можем запустить фаззер, указав конфигурационный файл:

./bin/syz-manager -config=config.cfg

2020/02/17 14:22:01 loading corpus...

2020/02/17 14:22:01 serving http on http://127.0.0.1:56741

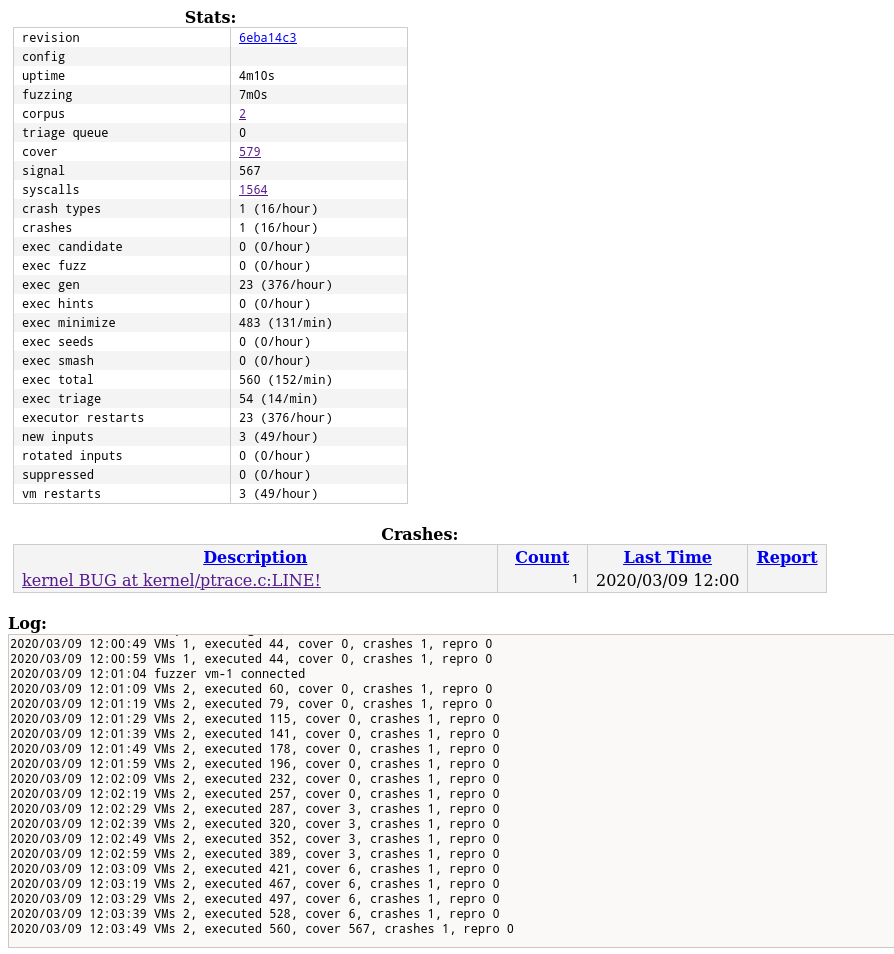
2020/02/17 14:22:01 serving rpc on tcp://[::]:46009

2020/02/17 14:22:01 booting test machines...

2020/02/17 14:22:01 wait for the connection from test machine...

...

Откроем указанный URL-адрес, видим веб-интерфейс. Это удобный способ навигации по функциям, проверки отчетов о покрытии, сбоях и корпусах. Обратим внимание, что сервер может не отвечать в течение нескольких секунд, пока инструмент выполняет некоторые тяжелые операции.



Вот как выглядит главная панель управления. В первом разделе можно увидеть текущее состояние инструмента, например, время безотказной работы, количество включенных системных вызовов, сбои и статистику выполнения фаззера. Во втором у вас есть список крашей и сколько раз они возникали. Он также может экспортировать отчет. Последний раздел — это журнал, тот же самый журнал, который инструмент выводит в терминал с живой информацией о ходе работы инструмента.

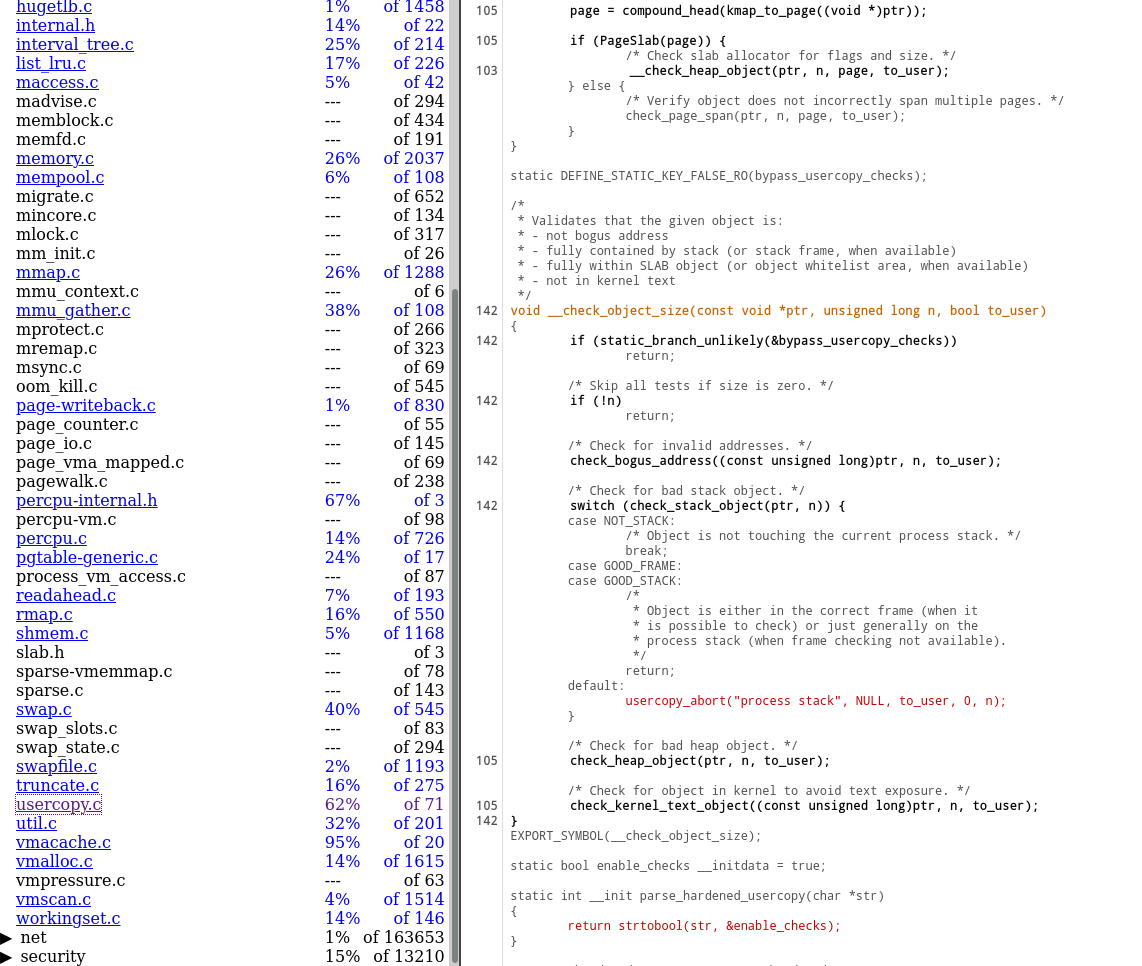
Кликнув на корпус, можно увидеть набор системных вызовов, которые создают текущее покрытие. Если системный вызов не может увеличить покрытие, он не будет сохранен в этой базе данных.



В списке отображается ссылка на покрытие, которое этот системный вызов производит, а также имя описания этого системного вызова. Нажав на название, можно увидеть инпут этого корпуса. Например, для syz\_mount\_image$ext4:

syz\_mount\_image$ext4(0x0, &(0x7f0000000040)='./file0\x00', 0x0, 0x0, 0x0, 0x0, 0x0)

Клик по отчету о покрытии на странице корпуса покажет покрытие для этого конкретного входного сигнала. Клик по ссылке покрытия на главной панели мониторинга покажет глобальное покрытие для текущего корпуса.



Это текущее покрытие файла usercopy.c для текущего корпуса. Объяснение цветов покрытия можно найти в документации.

<https://github.com/google/syzkaller/tree/master/docs>

Это и было основное использование syzkaller. Более подробную информацию можно найти в той же документации. Например, в файл конфига можно добавить следующее:

* "enable\_syscalls": [ "eventfd", "read$eventfd", ... ]: только эти системные вызовы будут фаззиться. Если они зависят от работы другого системного вызова (например, open ()), syzkaller попросит включить его. Использование read вызовет все определения для этого системного вызова, включая eventfd. Использование read$eventfd вызовет только один eventfd.
* "disable\_syscalls": ["ioctl$int\_int", “mmap”, ... ]: все системные вызовы будут фаззиться, за исключением тех, которые определены здесь.