

# **Natch v.2.1 Documentation**

# Содержание

[1. Начало работы с Natch](#)

[2. Основы работы с Natch](#)

[3. Конфигурационные файлы Natch](#)

[4. Запуск Natch](#)

[5. Функциональные возможности Natch](#)

[6. Графический интерфейс для анализа SNatch](#)

[Приложение 1. Настройка окружения для использования лицензированного Natch](#)

[Приложение 2. Командная строка эмулятора Qemu](#)

[Приложение 3. Формат списка исполняемых модулей \(module\\_config.cfg\)](#)

[Приложение 4. Формат конфигурационного файла для секции Tasks \(task\\_struct\\_offsets.ini\)](#)

[Приложение 5. Команды монитора Qemu для работы с Natch](#)

[Приложение 6. История релизов Natch](#)

# 1. Начало работы с Natch

*Natch* - это инструмент для определения поверхности атаки, основанный на полносистемном эмуляторе QEMU. С помощью этого раздела можно освоить основные принципы использования системы определения поверхности атаки [Natch](#) (разработчик - [ИСП РАН](#)):

- создание виртуализированной среды выполнения объекта оценки (далее - ОО) в формате [QEMU](#)
- запуск виртуализированной среды под контролем *Natch*
- анализ информации о движении помеченных данных в контролируемой виртуализированной среде

*Natch* поддерживает анализ только бинарного кода - таким образом анализ задеирования кода интерпретируемых скриптов, а также "распространения" помеченных данных по коду интерпретируемых скриптов, возможен только в опосредованном виде - в формате анализа задеирования нативных функций интерпретаторов, выполняющих указанные скрипты.

## 1.1. Комплект поставки

Комплект поставки *Natch* доступен в двух форматах:

- **основной** - защищенный бинарный дистрибутив, требующий наличие аппаратного ключа (персональный "черный" ключ, сетевой "красный" ключ или иные версии ключа) с лицензией с идентификатором "6":

[Natch v.2.0](#)

- **резервный** - .ova-образ Ubuntu 20 для VirtualBox с предустановленным защищенным *Natch*, необходимым ПО (pip3, vim) и доступом к VPN-серверу, раздающему лицензии:

[Natch v.2.0](#)

## 1.2. Подготовка виртуализированной среды в формате QEMU

Подготовка виртуализированной среды выполнения ОО в общем случае состоит из следующих последовательных шагов:

- создание образа эмулируемой операционной системы в формате диска [qcow2](#) на основе базового дистрибутива ОС. Формат *qcow2* позволяет эффективно формировать снимки состояния файловой системы в произвольный момент выполнения виртуализированной среды функционирования;
- сборка дистрибутива ОО с требуемыми параметрами, в частности, с генерацией и сохранением отладочных символов;
- помещение собранного дистрибутива ОО в виртуализованную среду выполнения;
- подготовка команд запуска QEMU, обеспечивающих эмуляцию аппаратной составляющей среды функционирования, загрузку и выполнение компонент *Natch*.

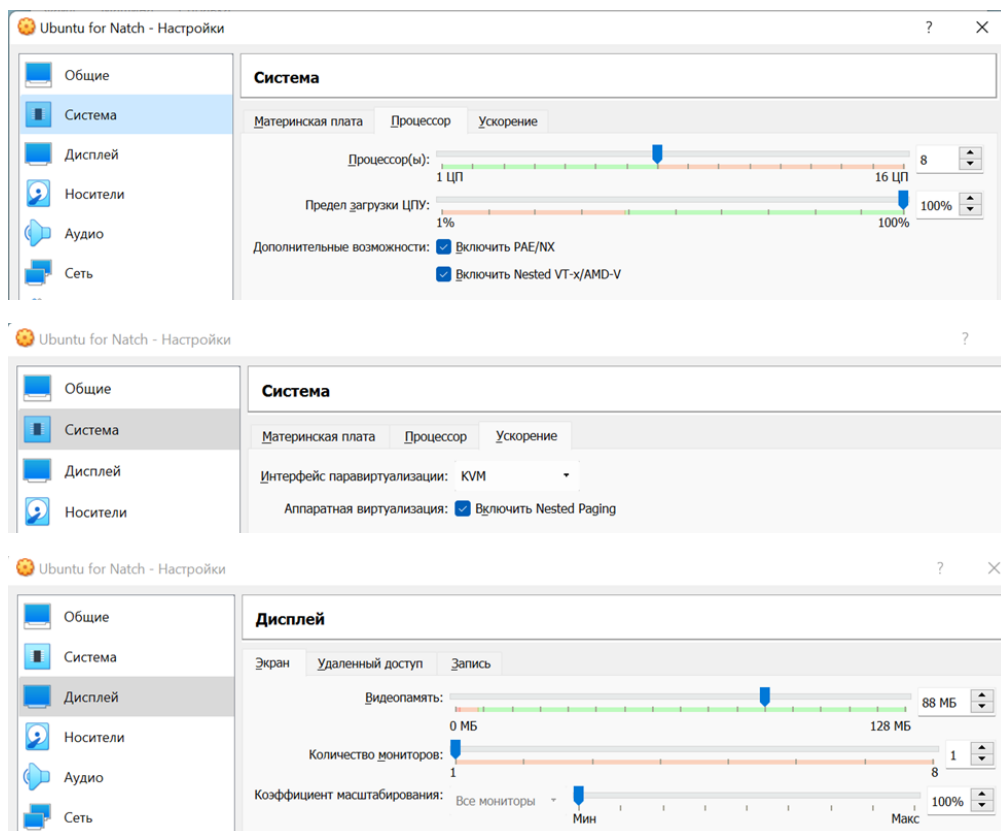
Подготовка виртуализированной среды выполнения ОО в значительной степени совпадает с подготовкой среды для анализа с помощью инструмента динамического анализа помеченных данных [Блесна](#) (разработчик - [ИСП РАН](#)), с точностью до подготовки команд запуска QEMU.

Создавать виртуализованную среду выполнения ОО **рекомендуется** в хостовой системе, допускающей запуск QEMU в режиме пользовательской виртуализации (ключ `-enable-kvm`) - это существенно ускорит процесс, скорость работы в режиме аппаратной виртуализации более чем на порядок превосходит работу в режиме полносистемной эмуляции. Проверить доступность данного режима в вашей хостовой системе (равно как и установить *kvm*-модули в вашу систему) можно опираясь на следующую [статью](#) с помощью команды:

```
sudo kvm-ok
```

Примерный алгоритм проброса виртуализации для трехуровневого стенда: *Windows 11+AMD Процессор (хостовая ОС рабочей станции)* -> *VirtualBox (хостовая ОС рабочей станции)* -> *ubuntu+kvm+qemu (хостовая ОС Natch)* -> *lubuntu (гостевая ОС)* приведен ниже.

Перед установкой KVM в гостевой ОС необходимо выполнить следующие настройки среды виртуализации VirtualBox в хостовой ОС Natch (для нормально функционирования системы, параметры должны быть в два раза больше чем на скринах, видеопамяти, по возможности вашей системы, кратно больше):



### *Настройки машины в VBox*

Перед установкой KVM необходимо определить, поддерживает ли процессор эту функцию: `egrep -c '(vmx|svm)' /proc/cpuinfo`

В результате будут следующие варианты ответа системы:

- 0 – процессор не поддерживает функции KVM;
- 1 и более – процессор поддерживает функции KVM.

Следующий этап – установка KVM: `sudo apt install qemu qemu-kvm libvirt-daemon libvirt-clients bridge-utils virt-manager`. Этой командой выполнена установка утилиты `kvm`, библиотеки `libvirt` и менеджера виртуальных машин.

Далее необходимо добавить своего пользователя в группу `libvirt`, так как только `root` и пользователи этой группы могут использовать виртуальные машины KVM: `sudo gpasswd -a $user libvirt`

Затем необходимо убедиться, что сервис `libvirt` запущен и работает: `sudo systemctl status libvirtd`

После выполнения этой команды выполнить: `reboot`

Далее, проверка установки `kvm`: `kvm-ok`

Если вы получили ответ:

INFO: `dev/kvm` exists

KVM acceleration can be used

Значит настройка выполнена правильно, и вы молодец :) Ваша QEMU-виртуализированная гостевая ОС будет работать быстро, что позволит быстро сформировать в ней исследуемую среду.

При этом важно помнить, что собственно запись трассы в любом случае необходимо выполнять без использования данного ключа, так как только полносистемная эмуляция позволяет собрать полный лог действий процессора.

### 1.2.1. Подготовка хостовой системы

Рекомендации по подготовке хостовой системы приведены [здесь](#) (для получения доступа к репозиторию сообщества ознакомьтесь с информацией в описании телеграм-канала [Орг. вопросы: Доверенная разработка](#)).

Подготовим Linux-based рабочую станцию (далее - хост), поддерживающую графический режим выполнения (QEMU демонстрирует вывод эмулируемой среды выполнения в отдельном графическом окне, следовательно нам необходим графический режим). Хост может быть реализован в формате виртуальной машины. В примерах ниже описаны действия пользователя, работающего в виртуальной машине VirtualBox (4 ядра, 8 ГБ ОЗУ) с установленной ОС [Ubuntu20:04](#) (desktop-конфигурация, обновить пакеты при установке).

Установим требуемое системное ПО, в т.ч. QEMU:

```
sudo apt install -y curl qemu-system gcc g++
```

*Подсказка: данная инсталляция требуется не для запуска Natch, но для создания образов VM на произвольном хосте. Natch содержит в своём составе требуемую для работы версию QEMU, поэтому если вы планируете создавать образ VM на том же хосте, на котором уже установили Natch, отдельно QEMU можно не ставить*

Скачаем на хост выбранный базовый дистрибутив ОС. Лучше использовать минимальный образ – уменьшение числа установленных служб, стартующих при запуске, значительно сокращает нагрузку на процессор и ускоряет запись и анализ трасс в режиме полносистемной эмуляции. В нашем примере используется легковесный образ Ubuntu - [lubuntu](#). Создадим каталог на хосте и скачаем туда дистрибутив:

```
cd ~ && mkdir natch_quickstart && cd natch_quickstart
curl -o lubuntu-18.04-alternate-amd64.iso 'http://cdimage.ubuntu.com/lubuntu/releases/18.04/release/
lubuntu-18.04-alternate-amd64.iso'
```

Проверим, работает ли эмулятор:

```
qemu-system-x86_64 --version
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.19)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
```

Для установки гостевой ОС создадим образ жесткого диска в формате qcow2, с именем lubuntu.qcow2 и размером 20 Гбайт. Установка параметра preallocation=metadata позволит сразу выделить весь объем виртуального диска, таким образом qemu не придется динамически наращивать его и выполнение сценария будет более быстрым.

```
qemu-img create -f qcow2 -o preallocation=metadata lubuntu.qcow2 20G
Formatting 'lubuntu.qcow2', fmt=qcow2 size=21474836480 cluster_size=65536 lazy_refcounts=off
refcount_bits=16
ll
total 8100768
drwxrwxr-x  4 user user      4096 янв 30 20:40 ./
drwxr-xr-x 25 user user      4096 янв 30 20:40 ../
-rw-rw-r--  1 user user 751828992 янв 30 20:34 lubuntu-18.04-alternate-amd64.iso
-rw-r--r--  1 user user  196928 янв 30 20:08 lubuntu.qcow2
```

Создадим скрипт запуска нашей VM run.sh. Он достаточно объемный, поэтому желательно именно сохранять его в виде отдельного файла, допускающего удобное внесение изменений. Для тех, кто сталкивается с синтаксисом QEMU впервые, настоятельно рекомендуется ознакомиться с основными командами, подробно расписанными в официальной [документации QEMU](#). Важным для

ускорения работы виртуализированной среды qemu, за счет проброса аппаратной [виртуализации](#), является установка ключей `-enable-kvm` и `-cpu host,nx`.

```
qemu-system-x86_64 \  
-hda lubuntu.qcow2 \  
-m 4G \  
-enable-kvm \  
-cpu host,nx \  
-monitor stdio \  
-netdev user,id=net0 \  
-device e1000,netdev=net0 \  
-cdrom lubuntu-18.04-alternate-amd64.iso
```

Запустим скрипт:

```
./run.sh
```

после чего увидим знакомое нам графическое окно установки *lubuntu*. Выполним установку *lubuntu* – желательно выполнять её с минимальным набором параметров – для ускорения процесса установки и минимизации потенциального “шума” избыточных процессов и сетевых служб в записях сетевого трафика и трасс выполнения.

*Подсказка: чтобы вывести курсор мыши из открытого графического окна VM QEMU нажмите Ctrl+Alt+G*

После завершения установки удалим из скрипта запуска `run.sh` указание подключения `cdrom` – для дальнейшей работы он нам не потребуется

```
#-cdrom lubuntu-18.04.3-desktop-amd64.iso
```

Наш образ среды функционирования готов к работе – в частности к установке в него пресобранного **с символами** прототипа объекта оценки.

### 1.2.2. Сборка прототипа объекта оценки

Рекомендации по подготовке исполняемого кода приведены [здесь](#).

В общем случае к подлежащему анализу исполняемому коду выставляется два требования:

- для исполняемого кода должна быть представлена отладочная информация в формате символов в составе исполняемых файлов, отдельно прилагаемых символов или тар-файлов. Предоставление символов непосредственно в составе исполняемых файлов является основной и рекомендуемой стратегией – начиная с версии *Natch v.1.3* инструмент умеет самостоятельно доставать информацию об отладочных символах из исполняемых файлов, собранных *как минимум* компиляторами *gcc* и *clang* с сохранением отладочной информации (ключ компилятора `-g`, также рекомендуется сборка без оптимизаций в режиме `-O0`). Начиная с версии *Natch v.2.1* будет внедрен функционал автоматической подгрузки символов для наиболее популярных сборок операционных систем;
- в случае, если процессы сборки и анализа будут выполняться в различных средах функционирования (как правило сборка осуществляется на отдельном высокопроизводительном сборочном сервере), требуется обеспечить совместимость версий разделяемых динамических библиотек, в первую очередь *glibc*, из состава среды функционирования.

В данном разделе в качестве прототипа объекта оценки рассмотрим популярную программу *wget*, сборку которой осуществим в хостовой системе (условная “сборочница”) с последующим помещением собранного дистрибутива в виртуализированную гостевую среду *lubuntu*.

Для выполнения [классического](#) подготовительного скрипта `configure`, входящего в комплект поставки *wget*, генерирующего `make`-файл, потребуется установить ряд дополнительных зависимостей (скрипт выведет их наименования в случае неудачного завершения), например:

```
sudo apt install -y gnutls-dev gnutls-bin curl
```

*Подсказка: поскольку мы собираем wget из исходников, потребуется комплект заголовочных файлов, доступный как раз в dev-версии пакета gnutls*

Скачаем исходные тексты wget с репозитория в файловую систему хоста:

```
curl -o wget-1.21.2.tar.gz 'https://ftp.gnu.org/gnu/wget/wget-1.21.2.tar.gz'
tar -xzf wget-1.21.2.tar.gz && cd wget-1.21.2
```

Скрипт configure запустим с ключами, устанавливающими параметры компилятора для сохранения информации об отладочных символах. После этого выполним команду make для сборки проекта.

```
CFLAGS='-g -O0' ./configure
make
```

**Важное замечание - следующие два подраздела оставлены в качестве пособия для специфических случаев, когда возможность сборки исполняемого файла из исходных текстов с произвольными параметрами отсутствует, либо явно требуется получение отдельных [тар-файлов](#).**

### 1.2.3. Генерация тар-файлов средствами компилятора

Выполним скрипт:

```
CFLAGS='-g -O0 -Xlinker -Map=output.map' ./configure && \
make
```

и проверим, что тар-файлы создались:

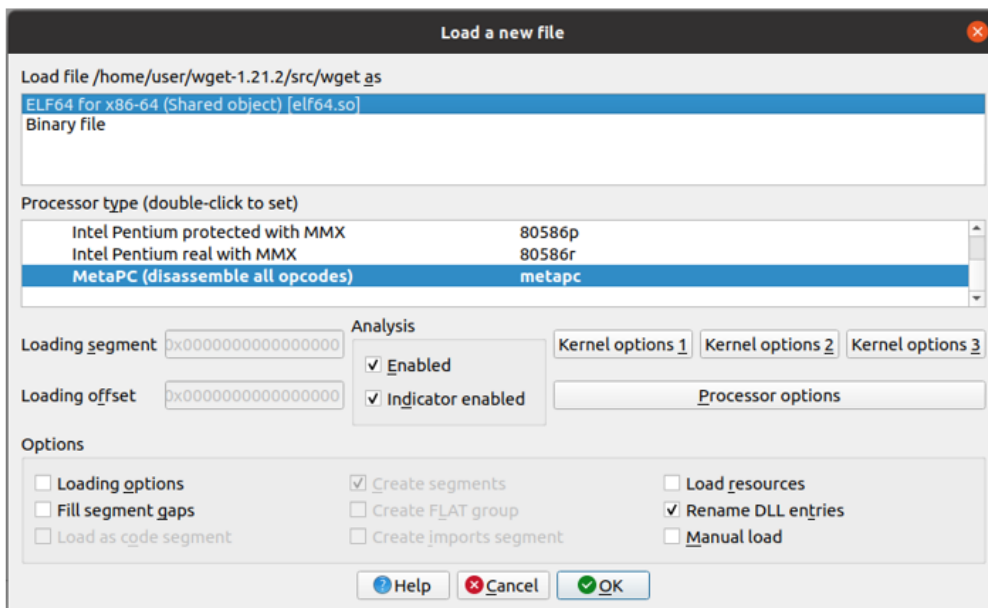
```
find . -name *.map
./src/output.map
./output.map
```

Полученные тар-файлы можно поместить на хосте в явно обозначенный Natch`у каталог, содержащий исполняемые файлы wget – тогда Natch будет опираться на данные тар-файлы при символизации соответствующих процессов. **Важное замечание: название бинарного файла и соответствующего ему тар-файла должны совпадать.**

### 1.2.4. Генерация тар-файлов сторонними инструментами

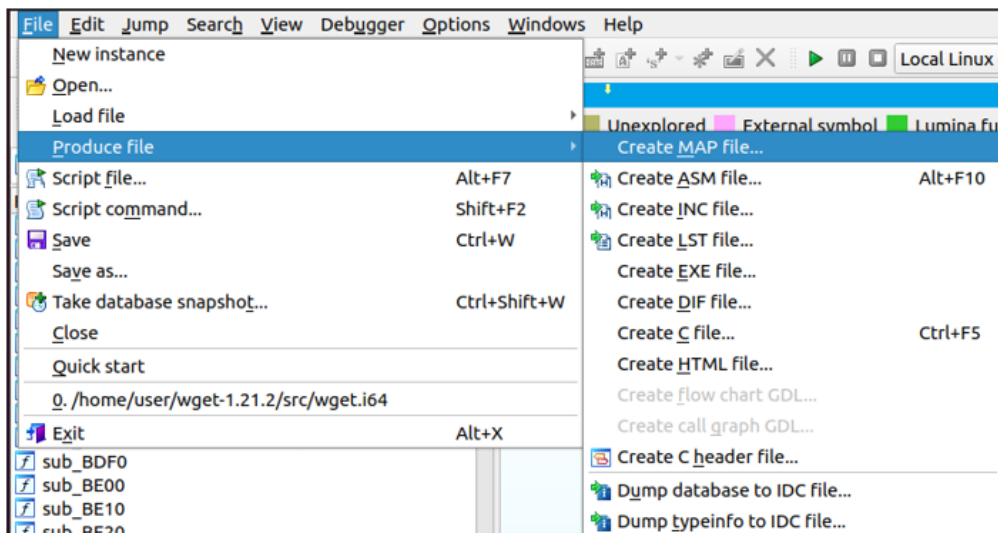
Получение тар-файлов для исполняемого файла, собранного с отладочными символами, возможно с помощью сторонних инструментов. Это может быть актуально в тех случаях, когда сборочный конвейер недоступен, либо получение от сборочного конвейера тар-файлов в поддерживаемом Natch формате невозможно (например, использование специфического компилятора/компоновщика). Сгенерируем тар-файлы с использованием бесплатной версии дизассемблера [IDA Pro](#) – необходимо скачать установочный комплект по указанной ссылке, возможно в систему придётся доустановить библиотеки Qt apt install -y qt5-default.

После установки IDA необходимо запустить её, открыть интересующий нас исполняемый файл (в нашем случае это wget)



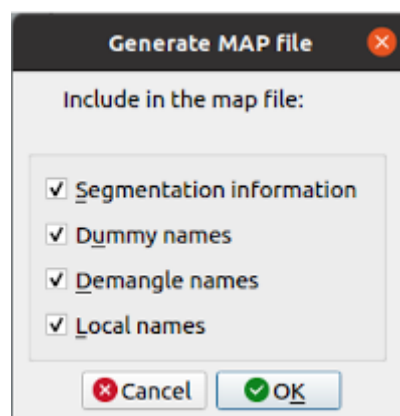
*Загрузка бинарного файла в IDA Pro*

пройти процедуру генерации тар-файла



*Генерация тар-файла*

обязательным пунктом является только *Segmentation information*, остальные по желанию (хотя, например, локальные имена дизассемблера вряд ли сделают вывод понятнее).



*Выбор опций тар-файла*

после чего убедиться, что тар-файл появился в файловой системе



```
ll src | grep .map
-rw-rw-r-- 1 user user 564686 фев 3 16:11 wget.map
```

### 1.2.5. Перенос прототипа объекта оценки в образ VM

Чтобы поместить собранный `wget` в виртуальную машину, воспользуемся [nbd-сервером QEMU](#), позволяющим [смонтировать](#) созданный ранее `qcow2`-диск VM в файловую систему хостовой ОС. Для монтирования диск не должен быть задействован (виртуальная машина должна быть выключена).

Загрузим NBD-драйвер в ядро хостовой ОС:

```
modprobe nbd max_part=8
```

Смонтируем наш образ диска как сетевое блочное устройство:

```
sudo qemu-nbd --connect=/dev/nbd0 lubuntu.qcow2
```

Определим число разделов на устройстве:

```
fdisk /dev/nbd0 -l
Disk /dev/nbd0: 20 GiB, 21474836480 bytes, 41943040 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe6ea1316
```

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/nbd0p1	*	2048	41940991	41938944	20G	83	Linux

Смонтируем раздел в какой-либо каталог хостовой ОС (например, традиционно, в `mnt`)

```
sudo mount /dev/nbd0p1 /mnt/
ls /mnt
bin dev home          initrd.img.old lib64      media  opt   root  sbin  swapfile tmp  var
vmlinuz.old
boot etc initrd.img lib          lost+found mnt     proc  run   srv   sys   usr  vmlinuz
```

Поместим прототип ОО на смонтированный раздел:

```
sudo cp -r wget-1.21.2 /mnt/ && ls /mnt
bin dev home          initrd.img.old lib64      media  opt   root  sbin  swapfile tmp  var
vmlinuz.old
boot etc initrd.img lib          lost+found mnt     proc  run   srv   sys   usr  vmlinuz
wget-1.21.2
```

Отмонтируем диск:

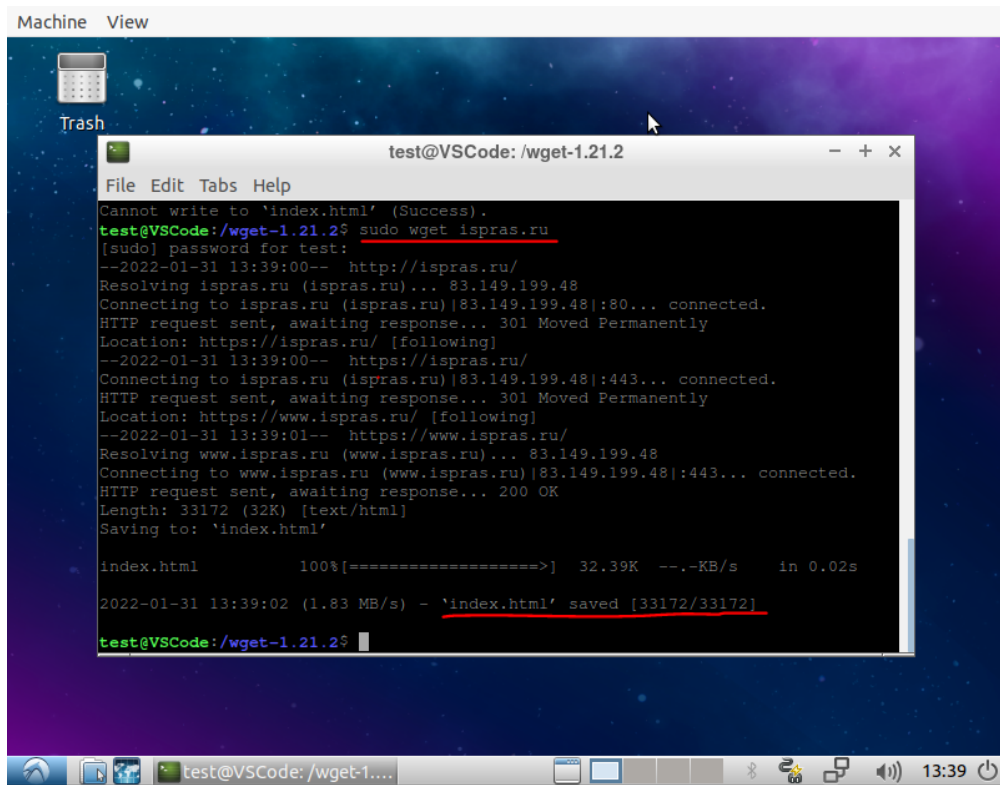
```
sudo umount /mnt/
sudo qemu-nbd --disconnect /dev/nbd0
sudo rmmod nbd
```

### 1.2.6. Тестирование виртуализированной среды функционирования ОО

Запускаем ВМ скриптом `run.sh` с учетом отключенного ранее `cdrom`, дожидаясь загрузки ОС ВМ, авторизуемся в ОС, пробуем выполнить обращение к произвольному сетевому ресурсу с помощью собранной нами версии `wget`:

```
cd /wget-1.21.2 && sudo ./wget ispras.ru
```

В результате вы должны увидеть приблизительно следующую картину в графическом окне QEMU, свидетельствующую о том, что ОО корректно выполняется в среде функционирования и сетевая доступность для ВМ обеспечена:



Пример подготовленного ОО в QEMU

## 1.3. Обучающие примеры

### 1.3.1. Анализ образа системы, содержащего тестовые комплекты пресобранных исполняемых файлов

Для выполнения данного примера потребуется:

- рабочая станция под управлением ОС Linux (традиционно Ubuntu 20.04). Отдельная установка пакета **qemu-system** не требуется, нужная версия входит в дистрибутив *Natch*;
- актуальный [дистрибутив Natch](#);
- подготовленный разработчиком [тестовый набор](#), включающий в себя минимизированный образ гостевой операционной системы Debian (размер `qcow2`-образа около 1 ГБ), а также два комплекта бинарных файлов (`Sample1_bins` и `Sample2_bins`), собранных с символами, к которым дополнительно прилагаются `map`-файлы.

Сценарий использования тестового комплекта `Sample1_bins`

Исполняемый файл `test_sample` читает помеченный файл (таковым при конфигурировании *Natch* нужно установить файл `sample.txt` в гостевой ОС), в первой строке которого записан адрес Google, выдергивает эту первую (уже помеченную в процессе чтения файла) строку и вызывает

исполняемый файл `test_sample_2` – в качестве параметра используется эта строка. Программа `test_sample_2` “курлит гугл” в файл `curl.txt`.

#### Сценарий использования тестового комплекта `Sample2_bins`

Процесс сервера `redis-server` следует запустить командой `redis-server --port 5555 --protected-mode no` (разумеется при конфигурировании `Natch` следует настроить проброс порта 5555, чтобы он был доступен из хостовой системы), после чего соединиться с ним из хостовой системы клиентской утилитой `redis-cli -h localhost -p 15555` (её можно поставить например так `sudo apt install redis-tools`) и выполнить какие-нибудь действия, например `SET b VeryBigValue`.

### 1.3.1.1. Получение образа и дистрибутива

В случае выполнении действий в подготовленной виртуальной машине, содержащей `Natch`, самостоятельные скачивание и установка бинарного комплекта не требуются.

В случае установки в формате бинарного комплекта следует скачать его и распаковать – команда для скачивания тестового комплекта с помощью `curl` выглядит так `curl -o materials.zip 'https://nextcloud.ispras.ru/index.php/s/testing_2.0/download'`. Состав комплекта бинарной поставки в облачном хранилище выглядит примерно так:



#### Комплект поставки

После скачивания дистрибутива и обучающих материалов их следует распаковать – традиционно (но не обязательно, реальное размещение файлов тестовых материалов не принципиально и зависит от ваших предпочтений) после распаковки содержимое каталога будет выглядеть примерно так:

```
user@natch1:~/natch_quickstart$ ll
total 24
drwxrwxr-x 6 user user 4096 сен 14 14:26 ./
drwxr-xr-x 17 user user 4096 сен 14 14:10 ../
drwxr-xr-x 2 user user 4096 сен 9 14:20 libs/
drwxr-xr-x 4 user user 4096 сен 14 11:24 Natch_testing_materials/
drwxr-xr-x 9 user user 4096 сен 9 14:20 qemu_plugins_2004_tainting_x64_natch/
drwxr-xr-x 5 user user 4096 сен 9 14:21 snatch/
```

#### Комплект поставки после распаковки

В каталоге `libs` размещаются используемые `Natch` библиотеки (подключаются с использованием стандартного механизма [preload](#) при запуске `qemu-system` и иных `qemu`-процессов). В каталоге `qemu_plugins...` помещаются собственно исполняемые файлы `Natch`. Каталог `docs`, содержащий веб-страницу руководства, располагается внутри каталога `qemu_plugins...`.

Учётные записи пользователей гостевой ОС: `user/user` и `root/root`.

### 1.3.1.2. Установка `Natch` и `Snatch`

Для работы `Natch` следует установить `python`-библиотеки, обеспечивающие работоспособность скриптов:

```
user@natch1:~/natch_quickstart$ pip3 install -r qemu_plugins_2004_tainting_x64_natch/bin/natch_scripts/requirements.txt
```

Для работы *Snatch* следует запустить установочный скрипт и дождаться его успешного выполнения (сообщения об ошибках сборки некоторых python-пакетов можно игнорировать при условии того, что скрипт в целом завершается успешно):

```
user@natch1:~/natch_quickstart/snatch$ ./snatch_setup.sh
```

### 1.3.1.3. Настройка Natch для работы с тестовым образом ОС

Процесс настройки состоит из двух этапов – автоматизированно (обязательный) и ручного (дополнительный, при необходимости тонкой настройки). Предназначение [файлов конфигурации и их параметров](#) подробно расписано в документации):

#### 1.3.1.3.1. Автоматизированная настройка

Автоматизированная настройка выполняется интерактивным скриптом *natch\_run.py*, выводимые которым вопросы и примеры ответов на которые приведём далее. Запуск скрипта (**не забываем про необходимость прелоада библиотек**):

```
user@natch1:~/natch_quickstart$ LD_LIBRARY_PATH=/home/user/natch_quickstart/libs/ ./
qemu_plugins_2004_tainting_x64_natch/bin/natch_scripts/natch_run.py Natch_testing_materials/
test_image_debian.qcow2
Image: /home/user/natch_quickstart/Natch_testing_materials/test_image_debian.qcow2
OS: Linux
```

Вводим имя проекта - будет создан каталог с таким именем:

```
Enter path to directory for project (optional): test1
Directory for project files '/home/user/natch_quickstart/test1' was created
Directory for output files '/home/user/natch_quickstart/test1/output' was created
```

Сколько памяти выдать гостевой виртуальной машине (постфикс указывать обязательно. G или M):

```
Common options
Enter RAM size with suffix G or M (e.g. 4G or 256M): 4G
```

Если наш сценарий предполагает передачу помеченных данных со сети (далее мы рассматриваем в качестве основного как раз сценарий №2 – взаимодействие с redis-сервером, слушающим tcp-порт 5555), нам следует передавать пакеты помеченных данных в гостевую ОС извне её - **перехват пакетов, отправитель и получатель которых “находятся” внутри гостевой ОС (localhost <-> localhost), в настоящий момент работает не стабильно и не рекомендуется к использованию**. Указываем *Natch*, какой порт мы хотим опубликовать в гостевую ОС:

```
Network option
Do you want to use ports forwarding? [Y/n] Y
Write the ports you want separated by commas (e.g. 7777, 8888, etc) 5555
Your port for connecting outside: 15555
```

Далее нам нужно указать пути к каталогам на хосте, содержащим копии бинарных файлов, размещенных в гостевой ОС - это как раз те самые файлы (собранные с символами, или с отдельными тар-файлами), которые мы получили в ходе выполнения пункта [Сборка прототипа объекта оценки](#). Нам нужно увидеть численное подтверждение того, что все помещенные в каталог файлы найдены (в данном комплекте их 2, *Natch* ищет все ELF-файлы и соответствующие им по названию тар-файлы на всю глубину вложенности каталогов):

```
Modules part
Do you want to create module config? [Y/n] Y
```

```
Enter path to maps dir: Natch_testing_materials/Sample2_bins
Your config file 'module_config.cfg' for modules was created
ELF files found: 2
Map files found: 2
```

Финальная стадия - конфигурирование технических параметров *Natch*, требующая тестового запуска виртуальной машины. В ходе данного запуска выполняется получение информации о параметрах ядра и заполнение ini-файла (данный файл не следует корректировать вручную, если только вы не абсолютно уверены в том, зачем вы это делаете). Вы можете отказаться от данного шага, в случае если этот файл уже был ранее создан для данного образа гостевой виртуальной машины – тогда вам потребуется указать к нему путь, однако, в большинстве случаев вы вероятно будете создавать таковые файлы с нуля:

```
Do you have a config file task_config.ini for your image? [N/y] n
```

```
Now will be launch tuning. Don't close emulator
```

```
Three...
```

```
Two..
```

```
One.
```

```
Go!
```

```
QEMU 6.2.0 monitor - type 'help' for more information
```

```
(qemu)
```

```
Natch v.2.0
```

```
(c) 2020-2022 ISP RAS
```

```
Reading Natch config file...
```

```
[Tasks] No such file '/home/user/natch_quickstart/test1/task_config.ini'. It will be created.
```

```
Now tuning will be launched.
```

```
Tuning started. Please wait a little...
```

```
Generating config file: /home/user/natch_quickstart/test1/task_config.ini
```

```
Trying to find 12 kernel-specific parameters
```

```
[01/12] Parameter - task_struct->pid           : Found
[02/12] Parameter - task_struct->comm           : Found
[03/12] Parameter - task_struct->parent          : Found
[04/12] Parameter - files_struct fields          : Found
[05/12] Parameter - vm_area_struct size         : Found
[06/12] Parameter - vm_area_struct->vm_start    : Found
[07/12] Parameter - vm_area_struct->vm_end      : Found
[08/12] Parameter - vm_area_struct->vm_flags    : Found
[09/12] Parameter - mm->map_count                : Found
[10/12] Parameter - mm_struct fields             : Found
[11/12] Parameter - task_struct->mm             : Found
[12/12] Parameter - task_struct->state          : Found
```

```
Tuning completed successfully! Now you can restart emulator and enjoy! :)
```

Отлично, автоматизированная настройка и создание базовых скриптов завершены успешно, всё готово к записи трассы, о чём *Natch* сообщил нам дополнительно:

```
File '/home/user/natch_quickstart/test1/natch_config.cfg' created. You can edit it before using Natch.
```

```
After checking config file you can launch:
```

```
just Natch with help 'run.sh'
```

```
Natch in record mode with help 'run_record.sh'
Natch in replay mode with help 'run_replay.sh'
Qemu without Natch with help 'run_qemu.sh'
```

Обратите внимание на файл настроек `natch_config.cfg` – именно его мы будем редактировать при необходимости выполнения ручной настройки, а также на файл `natch_log.log` - в нём логируются основные результаты работы подпрограмм, входящих в комплект поставки *Natch*.

### 1.3.1.3.2. Дополнительная ручная настройка

Отредактируем сгенерированный основной конфигурационный файл *Natch* `natch_config.cfg` в соответствии с рекомендациями. *Не забываем, что необходимо раскомментировать также названия секций в квадратных скобках, а не только сами параметры..* Раскомментируем следующие секции (подробнее об их предназначении см. пункт [Основной конфигурационный файл](#) документации):

Логирование сетевых пакетов, поступающих из источников, указанных в секции [Ports], в рсар-файл:

```
[NetLog]
on=true
log=packets
```

Пометка файла в гостевой ОС (пригодится для выполнения тестового сценария №1):

```
[TaintFile]
list=sample.txt
```

Сбор покрытия по базовым блокам для просмотра покрытия в *IDA Pro*:

```
[Coverage]
file=coverage
taint=true
```

### 1.3.1.4. Запись трассы

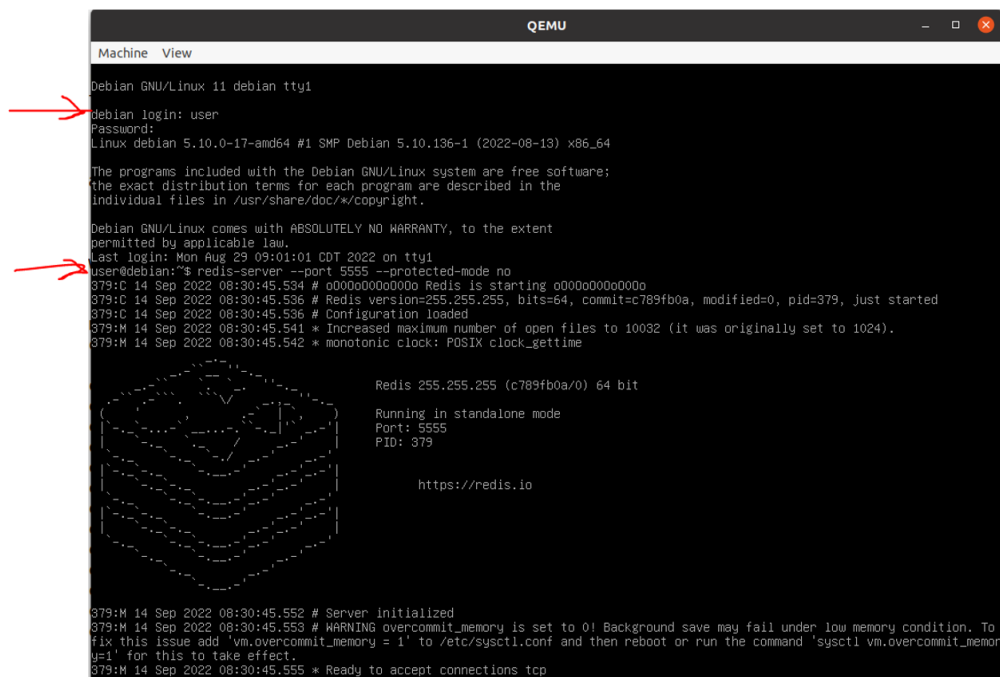
Выполним запись трассы с интересующим нас сценарием выполнения:

```
user@natch1:~/natch_quickstart/test1$ ./run_record.sh
```

**И получим ожидаемую ошибку - забыли про прелоад :)** Выполним запуск по правилам:

```
user@natch1:~/natch_quickstart/test1$ LD_LIBRARY_PATH=./libs/ ./run_record.sh
```

Введём логин и пароль учетной записи пользователя - `user/user` и запустим `redis`-сервер:



### Запуск redis-сервера

Тестово соединимся с ним из хостовой ОС чтобы убедиться, что система в комплексе работает как надо:

```
user@natch1:~/natch_quickstart$ redis-cli -h localhost -p 15555
```

```
localhost:15555> SELECT 0
OK
localhost:15555> SET a b
OK
localhost:15555> get a
"b"
localhost:15555> exit
```

Теперь всё готово к записи трассы. **Важный момент - трасса длинная, включает в себя в том числе этап загрузки ОС – но помеченные данные появятся практически в самом конце, когда мы собственно обратимся к redis-серверу.** Соответственно, для существенного сокращения времени на анализ (последующее выполнение `./run_replay.sh`) нам желательно и рекомендуется сделать снимок в точке, максимально приближенной к точке начала поступления помеченных данных в системе. То есть сейчас, когда от порождения помеченных данных нас отделяет только повторное соединение с redis-сервером из хостовой ОС и повторная отправка в него тех или иных уже знакомых нам команд.

Нажмем `Ctrl+Alt+G`, выйдем в монитор QEMU (bash-терминал хостовой ОС в котором мы запустили `run_record.sh`) и выполним команду генерации снимка (займёт несколько секунд, в зависимости от размера образа и производительности компьютера в целом) - я назвал его `ready` - **команда `savevm ready`:**

```
user@natch1:~/natch_quickstart/test1$ LD_LIBRARY_PATH=../libs/ ./run_record.sh
```

```
QEMU 6.2.0 monitor - type 'help' for more information
```

```
(qemu)
```

```
Natch v.2.0
```

```
(c) 2020-2022 ISP RAS
```

```
Reading Natch config file...
```

Config is loaded.

You can make system snapshots with the command: `savevm <snapshot_name>`

Natch works in network packet logging mode

Network logging plugin started with pcap log file: `"/home/user/natch_quickstart/test1/packets.pcap"`

```
(qemu) savevm ready
```

```
(qemu)
```

После того, как снэпшот был сгенерирован, снова отправим какие-нибудь данные из хостовой ОС в redis-сервер. Теперь завершим QEMU, закрыв графическое окно эмулятора.

Поздравляю, трасса записана!

### 1.3.1.5. Воспроизведение трассы

Для воспроизведения трассы нужно запустить скрипт `run_replay.sh`. Скрипт может принимать параметр с именем снэпшота, в нашем случае команда будет выглядеть так:

```
user@natch1:~/natch_quickstart$ LD_LIBRARY_PATH=/home/user/natch_quickstart/libs/ ./test1/run_replay.sh
ready
```

Если по какой-то причине вы не хотите использовать параметр, то скрипт можно запустить без, но при этом надо будет внести изменения в сам скрипт. Перед воспроизведением трассы следует заменить значение параметра `SNAPSHOT` на имя нашего снэпшота, например, используя редактор `vim`. По умолчанию значение параметра `init`. Заменяем:

```
SNAPSHOT="ready"
```

Начнём воспроизведение трассы (приблизительно на порядок медленнее, чем базовой выполнение - вы моментально оцените пользу создания снэпшота):

```
user@natch1:~/natch_quickstart$ LD_LIBRARY_PATH=/home/user/natch_quickstart/libs/ ./test1/run_replay.sh
```

Если в скрипт не был передан параметр и скрипт не был отредактирован - воспроизведение начнется с начала загрузки ОС.

В ходе выполнения, в случае если вам требуются какие-то диагностики, имеющие отношение к конкретному моменту, в мониторе можно вводить команды, в частности команды плагинов *Natch*, указанные в разделе [Приложение 5. Команды монитора Qemu для работы с Natch](#). К примеру, команда `show_tasks` показывает дерево процессов в гостевой ОС в текущий момент:

```
(qemu) show_tasks tasks.txt
```

Task struct tree:

task\_struct: 0xffffffff9dc13740

ctx: 0x137e6a000

img: swapper/0

pid: 0x0

state: 0x0

stack: 0

task\_struct: 0xfffff8e977af88dc0

ctx: 0x13910e000

img: systemd

pid: 0x1

state: 0x1

stack: 0



```
task_struct: 0xfffff8e977e977a329b80
ctx: 0x13alec000
img: containerd
pid: 0x165
state: 0x1
stack: 0
```

Через какое-то время выполнение сценария завершится, графическое окно закроется, и вы должны будете увидеть сообщение наподобие приведённого ниже, свидетельствующее о том, что интересующие нас модули гостевой ОС (в данном случае это один модуль - redis-сервер) были распознаны успешно, и, следовательно, мы получим в отчетах корректную символьную информацию.

QEMU 6.2.0 monitor - type 'help' for more information

(qemu)

Natch v.2.0

(c) 2020-2022 ISP RAS

Reading Natch config file...

Task graph enabled

Taint enabled

Config is loaded.

Module binary log file /home/user/natch\_quickstart/test1/output/log\_m\_b.log created successfully

Modules: started reading binaries

Modules: finished with 2 of 2 binaries for analysis

thread\_monitor: identification method is set to a complex developed at isp approach

Started thread monitoring

Process events binary log file /home/user/natch\_quickstart/test1/output/log\_p\_b.log created successfully

Tasks: config file is open.

Binary log file /home/user/natch\_quickstart/test1/output/log\_t\_b.log created successfully

Detected module /home/user/natch\_quickstart/Natch\_testing\_materials/Sample2\_bins/redis-server execution

Если работа системы завершилась успешно, и вы не словили, например, core dumped (о чём стоит немедленно сообщить в [трекер](#) с приложением всех артефактов), можно переходить к собственно анализу трассы.

### 1.3.1.6. Анализ трассы в ручном режиме

Основные виды диагностики, предоставляемые *Natch*, расписаны в разделе [Функциональные возможности Natch](#). Следует ознакомиться со списками:

- задействованных модулей
- задействованных функций

а также:

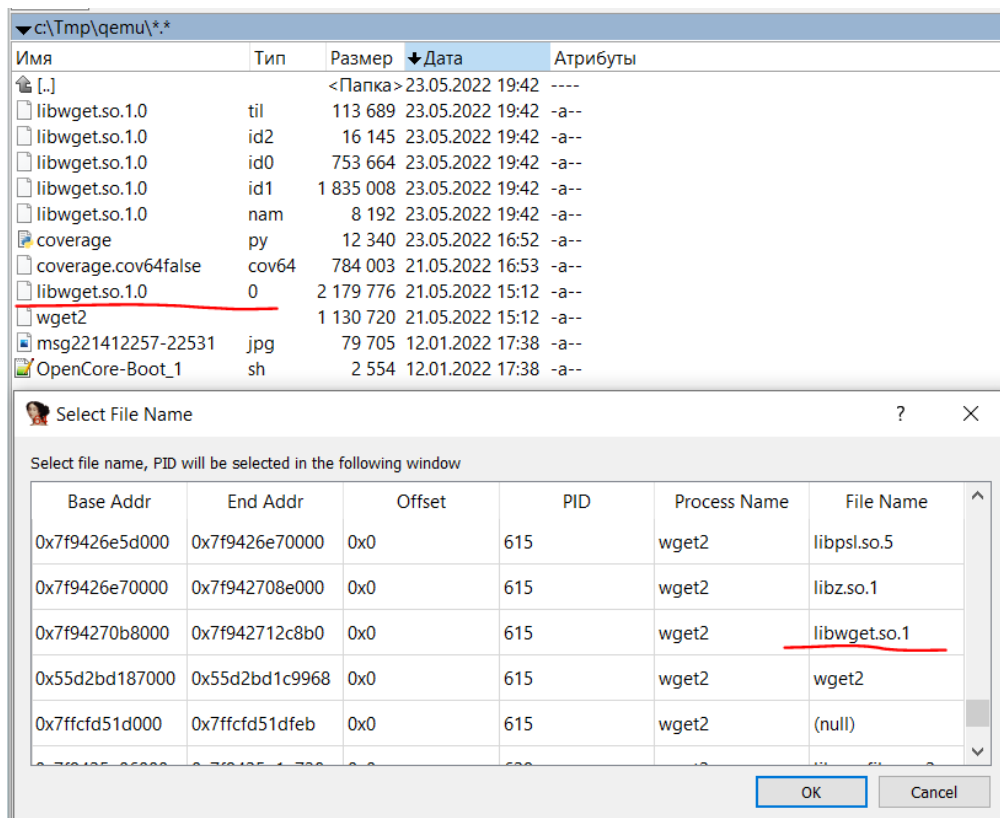
- графом движения помеченных данных
- сетевым трафиком, доступным в рсар-файле on.pсар, а также в файле лога помеченных данных tnetpackets.log

а также проанализировать:

- какие функции в наибольшей степени взаимодействовали с помеченными данными
- покрытие по базовым блокам функций, взаимодействовавших с помеченными данными

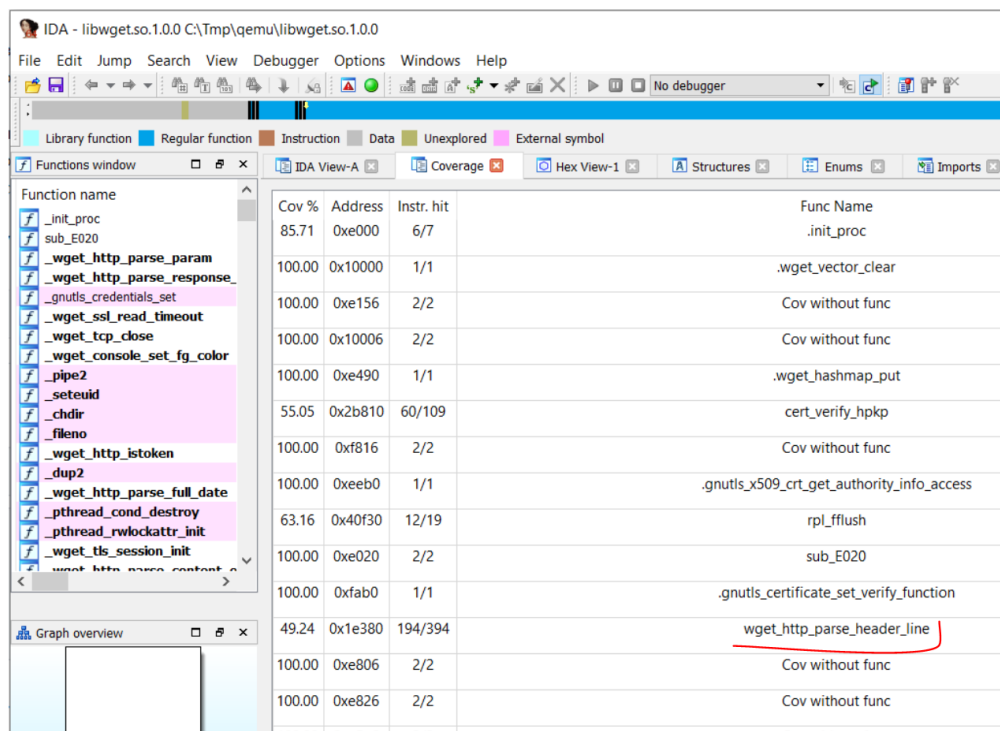
Анализ покрытия по базовым блокам выполняется с использованием *IDA Pro* (протестировано на версиях 7.0, 7.2), общий алгоритм действий описан в пункте [Анализ покрытия бинарного кода](#). В ходе его выполнения может потребоваться ручное сопоставление модуля, для которого собрано

покрытие, с модулем, загруженным в *IDA Pro*. Наиболее явная причина – несовпадение имён исполняемого файла и файла, распознанного *Natch*. Пример такового несовпадения приведён на рисунке ниже:



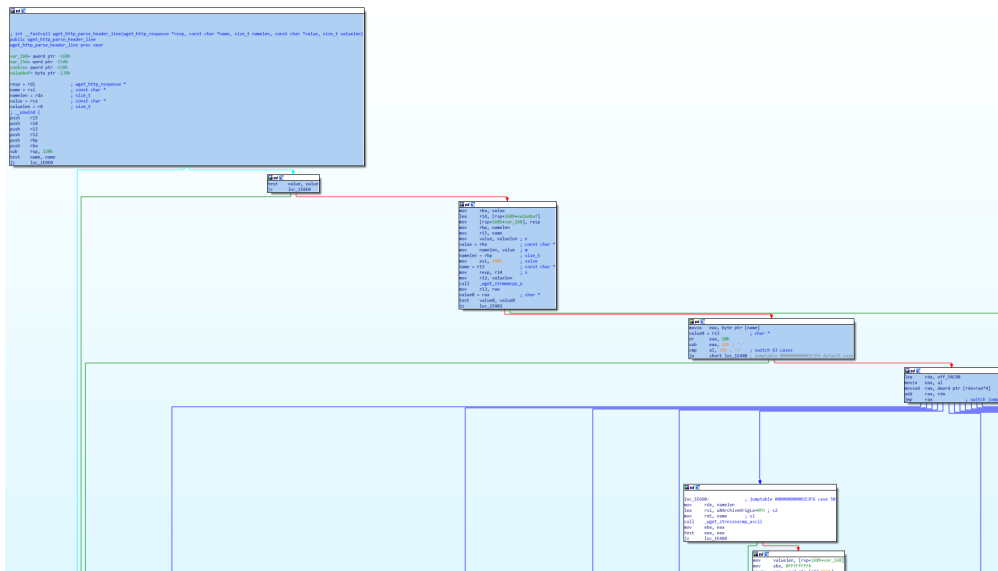
*Пример несовпадения имен модулей*

После выполнения маппинга в представленном выше меню в ручном режиме мы увидим приблизительно следующие сведения о покрытии:



*Загруженный проект*

Также при выборе функции можно увидеть покрытие непосредственно по ассемблерным инструкциям (голубой цвет):



Пример покрытия по ассемблерным инструкциям

Демонстрация покрытия по декомпилированному коду в настоящий момент не поддерживается.

Также можно открыть и изучить записанный файл сетевого трафика wireshark packets.pcap:

packets.pcap									
No.	Time	Source	Destination	Protocol	Length	Info			
61	541.378822663	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=77507 Win=65535 Len=0		
62	541.378822664	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=78967 Win=65535 Len=0		
63	541.378822665	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=80427 Win=65535 Len=0		
64	541.378822666	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=81887 Win=65535 Len=0		
65	541.378822667	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=83347 Win=65535 Len=0		
66	541.378822668	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=84807 Win=65535 Len=0		
67	541.378822669	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=86267 Win=65535 Len=0		
68	541.378822670	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=87727 Win=65535 Len=0		
69	541.378822671	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=89187 Win=65535 Len=0		
70	541.378822672	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=90647 Win=65535 Len=0		
71	541.378822673	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=18 Ack=90662 Win=65535 Len=0		
72	542.968719360	10.0.2.2	10.0.2.15	TCP	68	47012 → 5555	[PSH, ACK] Seq=18 Ack=90662 Win=65535 Len=0		
73	543.022452736	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=32 Ack=93582 Win=65535 Len=0		
74	543.022452737	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=32 Ack=95042 Win=65535 Len=0		
75	543.022452738	10.0.2.2	10.0.2.15	TCP	60	47012 → 5555	[ACK] Seq=32 Ack=95611 Win=65535 Len=0		
76	543.908972800	10.0.2.2	10.0.2.15	TCP	93	47012 → 5555	[PSH, ACK] Seq=32 Ack=95611 Win=65535 Len=0		

Frame 76: 93 bytes on wire (744 bits), 93 bytes captured (744 bits)

Ethernet II, Src: 52:55:0a:00:02:02 (52:55:0a:00:02:02), Dst: RealtekU\_12:34:56 (52:54:00:12:34:56)

Internet Protocol Version 4, Src: 10.0.2.2, Dst: 10.0.2.15

Transmission Control Protocol, Src Port: 47012, Dst Port: 5555, Seq: 32, Ack: 95611, Len: 39

Data (39 bytes)

```

0000  52 54 00 12 34 56 52 55 0a 00 02 02 08 00 45 00  RT...4VU...E-
0010  00 4f 00 49 00 00 40 06 62 50 0a 00 02 02 0a 00  0I...0 DP.....
0020  02 0f b7 a4 15 b3 00 60 ae 21 af 4e 37 ab 50 18  .....!N7P...
0030  ff ff b7 a2 00 00 2a 33 0d 0a 24 33 0d 0a 73 65  .....*3...$3...se
0040  74 0d 0a 24 31 0d 0a 62 0d 0a 24 31 32 0d 0a 56  t...$1..b...$12..V
0050  65 72 79 42 69 67 56 61 6c 75 65 0d 0a          eryBigVa lue...

```

Исследование трафика в Wireshark

### 1.3.1.7. Анализ трассы с использованием Snatch

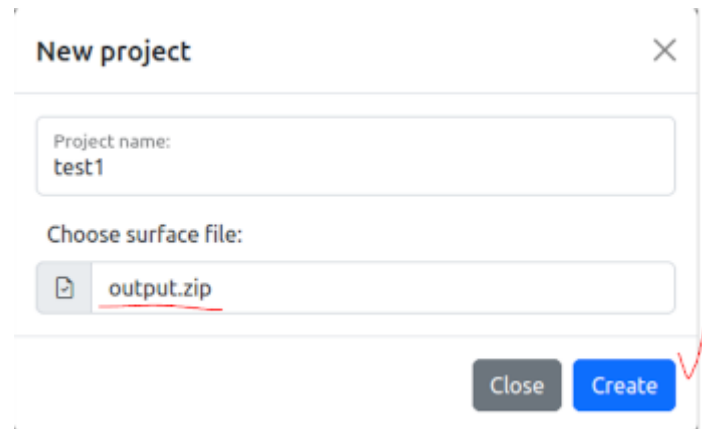
*SNatch* – это подсистема визуализации данных и статистик, полученных в ходе анализа записанной *Natch* трассы. *SNatch* реализован в формате веб-службы с браузерным интерфейсом просмотра.

В комплект поставки *SNatch* входят скрипты `snatch_run.sh` и `snatch_stop.sh` для запуска и остановки *SNatch* соответственно. Скрипт `snatch_run.sh` запускает необходимые для работы службы, а также открывает браузер с интерфейсом. В терминал, из которого был запущен скрипт, будут приходить сообщения от сервера, однако, он свободен для использования, поэтому по окончании работы из него же можно запустить скрипт `snatch_stop.sh` для остановки служб. Запускать `snatch_stop.sh` следует всегда, в противном случае процессы останутся висеть в памяти вашего компьютера до перезагрузки.

Запустим *SNatch*:

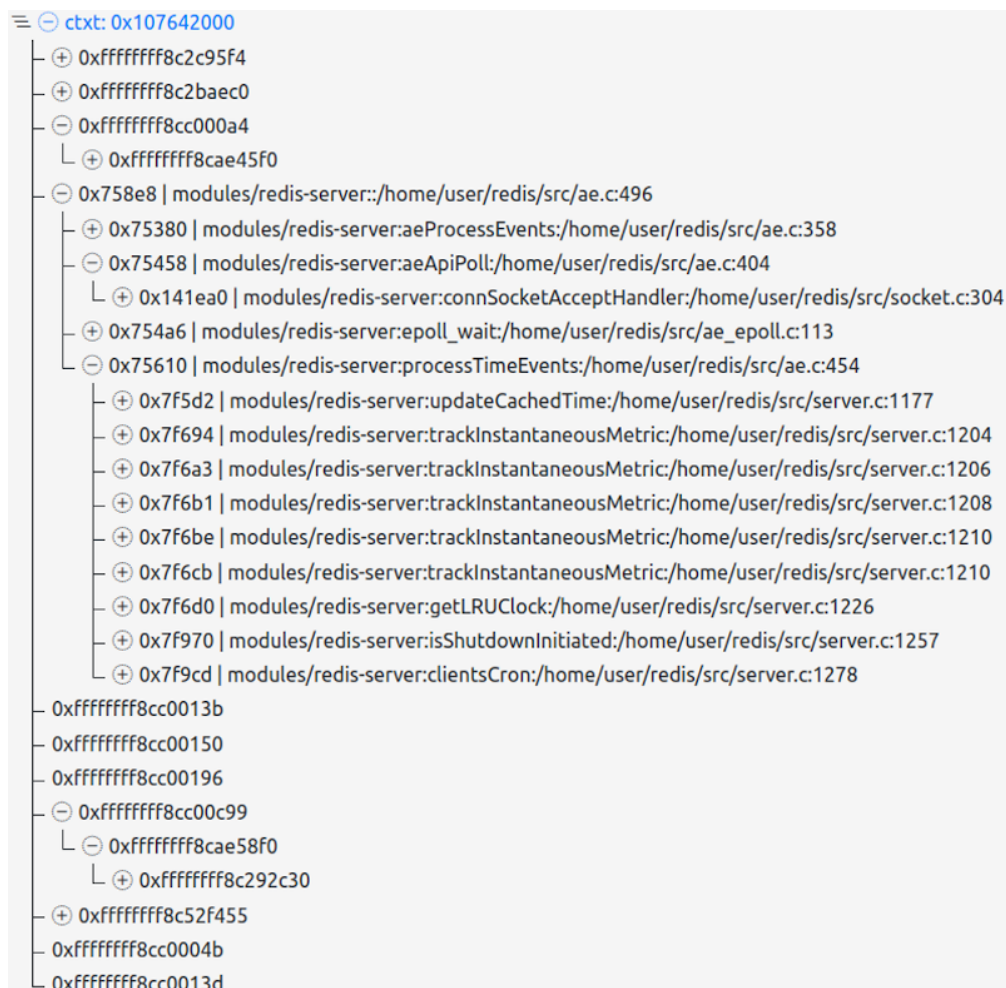
```
user@natch1:~/natch_quickstart$ ./snatch/snatch_run.sh
```

Создадим проект на основе результатов анализа трассы (необходимо указывать zip-архив, формируемый *Natch* в каталоге проекта по результатам выполнения `run_replay.sh`):



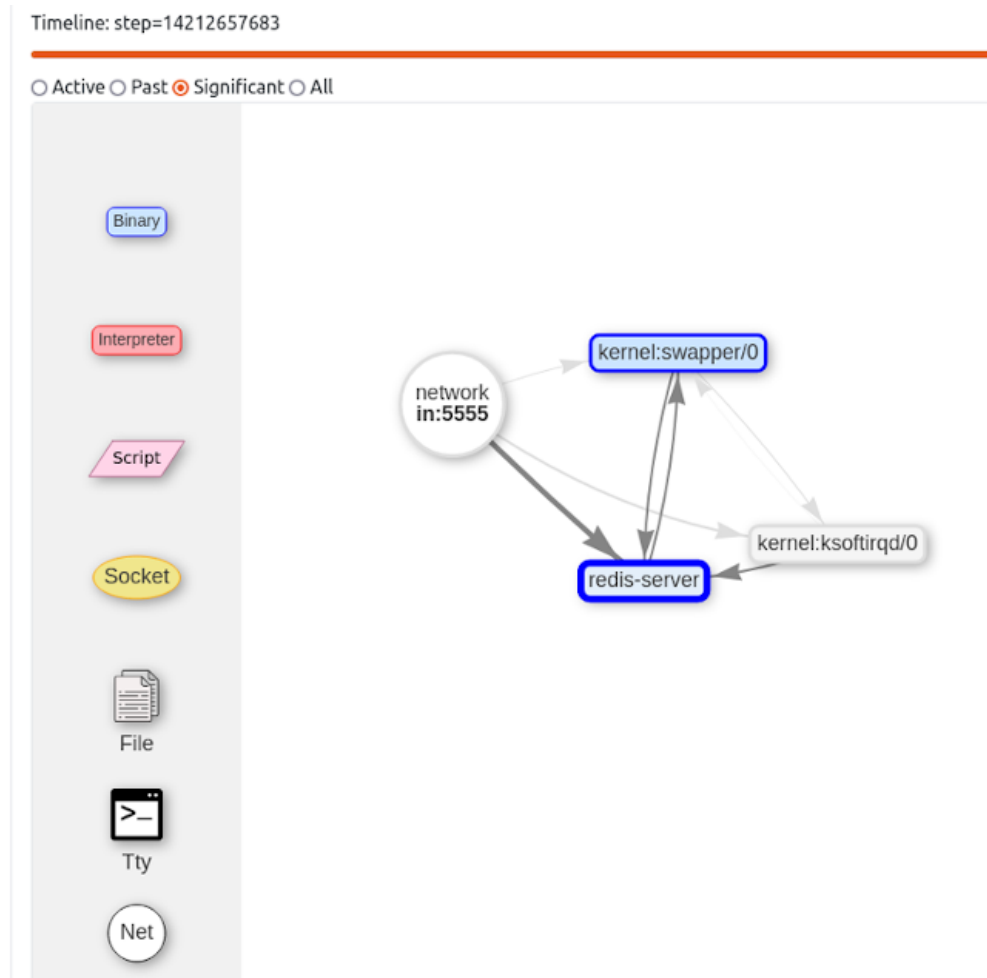
*Создание Snatch проекта*

Через некоторое время процесс загрузки проекта завершится и станут доступны различные виды (**их число и возможности постоянно нарастают**) аналитик, такие как просмотр стека вызовов обработки помеченных данных:



*Стек вызовов*

а также основное окно динамической визуализации распространения помеченных данных:



*Граф процессов, работавших с помеченными данными*

Полное руководство пользователя *SNatch* доступно в соответствующем разделе [Графический интерфейс для анализа SNatch](#). Основное внимание нужно обратить на то, что ярким цветом на каждом шаге *Timeline* выделяются сущности, взаимодействующие на данном конкретном шаге *Timeline*.

## 2. Основы работы с Natch

Инструмент *Natch* для определения поверхности атаки отслеживает потоки данных с помощью пометок. Обнаруженные участки кода средствами интроспекции соотносятся с программными модулями и функциями.

*Natch* представляет собой набор плагинов для Qemu, которые инструментируют выполняемый код и существенно замедляют работу эмулятора. В связи с этим, анализ предлагается проводить с использованием детерминированного воспроизведения. Детерминированное воспроизведение - это технология, которая позволяет записывать, а затем многократно воспроизводить и анализировать сценарий работы виртуальной машины. В нашем случае на записанный сценарий уже не будут оказывать влияния задержки от плагинов анализа, что очень важно для корректной работы часов реального времени и при взаимодействии с сетью.

Примерный алгоритм получения поверхности атаки может выглядеть следующим образом:

- Запуск Qemu с плагином *natch*
- Ожидание загрузки до интересующего момента
- Включение анализа помеченных данных (если он не был включен автоматически через конфигурационный файл)
- Выполнение тестового сценария
- Загрузка полученных данных в веб-интерфейс
- Анализ и оценка результатов

## 3. Конфигурационные файлы Natch

Для работы инструмента *Natch* требуются конфигурационные файлы. Основной конфигурационный файл отдается непосредственно инструменту на вход, остальные являются источниками настроек для плагинов, входящих в состав *Natch*.

### 3.1. Основной конфигурационный файл

Пример файла конфигурации приведен ниже, но пользователю может потребоваться внести в него изменения. Содержимое файла конфигурации:

```
# Natch settings

# Section Version since Natch v.2.0
[Version]
version=1

# Section for path to work directory
[OutputData]
workdir=/home/user/workdir

# Section for loading task_struct offsets
[Tasks]
config=task_config.ini

# Section for loading modules
[Modules]
config=module_config.cfg
log=taint.log
params_log=params.log

[Ports]
# 6 is for tcp
ip_protocol=6
# ports are supported for tcp only yet
in=22;80;3500;5432
out=22;80;3500;5432

[Taint]
# Threshold value for tainting. Should be in decimal number system [0..255]
threshold=50
on=true

# Section for enable generating graphs
[TaintedTask]
task_graph=true
module_graph=false

# Section for tainted network packets. !Only for replay mode!
[NetTaintLog]
log=tnetpackets.log
```

```
# Section for network log in pcap. !Only for record mode!
```

```
[NetLog]
```

```
on=true
```

```
log=netpackets.log
```

```
# Section for add tainted files
```

```
[TaintFile]
```

```
list=file1.txt;file2.txt
```

```
# Section for getting coverage
```

```
[Coverage]
```

```
file=coverage.cov64
```

```
taint=true
```

```
# Section for enabling additional plugins
```

```
[Plugins]
```

```
items=bcqti,broker,addr=:5555;some_plugin
```

```
# Section for loading custom syscall config
```

```
[Syscalls]
```

```
config=custom_x86_64.cfg
```

## Секция Version

- Поле *version*: номер версии конфигурационного файла *Natch*. Генерируется автоматически, редактирование не требуется.

## Секция OutputData

- Поле *workdir*: путь к директории, куда будут записываться все файлы, генерируемые инструментом. Внутри *workdir* будут созданы два каталога *output* и *output\_text*. В первый будут записаны файлы для анализа в графической подсистеме (бинарные логи, графы взаимодействия процессов и модулей, символьная информация), этот же каталог будет заархивирован. В *output\_text* попадут файлы с текстовым представлением поверхности атаки (*surface\_functions.txt* и *surface\_modules.txt*).

## Секция Tasks

- Поле *config*: указывается имя конфигурационного файла для распознавания процессов (подробнее в пункте [Tasks](#)).

## Секция Modules

- Поле *config*: указывается имя конфигурационного файла для распознавания модулей (подробнее в пункте [Modules](#)).
- Поле *log*: содержит название файла, в который в процессе работы будет записываться подробный лог помеченных данных (подробнее в пункте [Подробная трасса помеченных данных](#)).
- Поле *params\_log*: содержит название файла, в который в процессе работы будет записываться лог с помеченными параметрами функций (подробнее в пункте [Получение областей помеченной памяти для функций](#)).

## Секция Ports

- Поле *ip\_protocol* описывает тип протокола 4 уровня. Если не указано, пакеты по этому полю не фильтруются.
- Поле *out* - фильтр по Source Port в заголовке TCP, порты перечисляются через точку с запятой.



- Поле *in* - фильтр по Destination Port в заголовке TCP, порты перечисляются через точку с запятой.

При необходимости отслеживать трафик по всем портам, в полях *in/out* секции *Ports* следует указать значение -1. Если хотя бы в одном поле будет -1, будет отслеживаться весь трафик.

### Секция Taint

- Поле *threshold*: пороговое значение для отслеживания помеченных данных, задается десятичным числом в диапазоне от 0 до 255. Чем больше число, тем пометка будет сильнее, то есть в поверхность атаки будут попадать минимально измененные данные.
- Поле *on*: принимает логическое значение, при установке в true отслеживание помеченных данных будет включено при старте эмулятора. Если это не требуется, следует установить значение false.

Если секция *Taint* не определена, по умолчанию отслеживание помеченных данных будет выключено и пороговое значение будет установлено в 0.

### Секция TaintedTask

- Поле *task\_graph*: принимает логическое значение, при установке в true при завершении работы эмулятора будет создан граф задач и потоков помеченных данных.
- Поле *module\_graph*: принимает логическое значение, при установке в true при завершении работы эмулятора будет создан граф модулей и потоков помеченных данных.

### Секция NetTaintLog

- Поле *log*: содержит название файла, в который в процессе воспроизведения будут записываться помеченные сетевые пакеты.

### Секция NetLog

- Поле *on*: принимает логическое значение, при установке в true осуществляется сохранение сетевых пакетов в файл.
- Поле *log*: опциональное поле, содержит название файла для записи пакетов. Расширение указывать не нужно, оно автоматически будет *.pcap*. Если поле не задано, имя файла по умолчанию *net\_packets\_log.pcap*.

Использование этой секции предусмотрено только в режиме записи журнала и собирается только входящий трафик.

### Секция TaintFile

- Поле *list*: через точку с запятой могут быть перечислены имена файлов, которые требуется пометить. Указываются имена файлов гостевой машины в формате имя + расширение. Для надежности рекомендуется не использовать пути. Пометка произойдет автоматически при запуске эмулятора.

Если включена секция *TaintFile* без указания списка файлов, плагин *taint\_file* все равно будет загружен.

### Секция Coverage

- Поле *file*: указывается имя файла, куда будет записана операция о покрытии кода.
- Поле *taint*: определяет режим сбора покрытия кода (подробнее в пункте [Анализ покрытия бинарного кода](#)).

### Секция Plugins

- Поле *items*: через точку с запятой указываются плагины, не входящие в состав *Natch*, но которые должны быть загружены.

## Секция Syscalls

- Поле *config*: указывается имя конфигурационного файла для перехвата системных вызовов (подробнее в пункте [Syscalls](#)).

## 3.2. Modules

В этой секции можно указать конфигурационный файл, описывающий анализируемые исполняемые модули. *Natch* может находить загруженные модули, на которые передавалось управление, но определить их имена и функции не всегда возможно.

В этом случае можно загрузить образы интересующих модулей через конфигурационный файл. Использование этого конфигурационного файла может понадобиться в следующих случаях:

- Для правильного определения имен бинарных файлов и их экспортируемых функций
- Для подгрузки тар-файла с именами функций
- Для подгрузки дополнительных ELF-файлов с отладочной информацией (подробнее в [Приложение 3. Формат списка исполняемых модулей](#))

Из основного бинарного файла всегда читаются экспортируемые символы и отладочная информация, сгенерированная компилятором при использовании ключа *-g*. Поддерживаются все современные компиляторы. Также можно использовать тар-файлы, сгенерированные с помощью IDA Pro. Это можно сделать через меню File -> Produce file -> Create MAP File. В появившемся после ввода имени файла диалоге нужно выставлять галочку *Segmentation information*.

### 3.2.1. Автоматическая генерация конфигурационного файла

Конфигурационный файл со списком модулей может быть сгенерирован автоматически при помощи входящего в поставку скрипта *module\_config.py* во время выполнения скрипта *natch\_run.py* (подробнее в пункте [Использование скрипта для генерации командных строк запуска](#)).

Перед работой нужно установить зависимости из списка *requirements.txt* с помощью команды

```
pip3 install -r requirements.txt
```

Скрипт принимает ряд параметров:

- Путь к директории, содержащей модули и тар-файлы (обязательный).
- Путь к рабочей директории, в которую будет помещен лог работы инструмента. По умолчанию в месте запуска скрипта.
- Флаг включения диагностических сообщений.

```
module_config.py folder [-h] [-l] [-d]
```

Имена тар-файлов и исполняемых файлов должны соответствовать друг другу: скрипт будет искать тар-файлы, приписывая суффикс *.tar* к полному имени исполняемого файла.

Пример запуска скрипта:

```
./module_config.py <path_to_modules>
```

## 3.2. Tasks

Конфигурационный файл этой секции генерируется автоматически на этапе конфигурирования *Natch*.

Если во время настроечного запуска индикатор прогресса заводится на одном пункте, то вероятно это связано с недостатком количества перехватываемых системных вызовов. Для решения проблемы можно попробовать запустить на гостевой системе какую-либо программу.

### **3.3. Syscalls**

Конфигурационные файлы для перехвата системных вызовов поставляются с инструментом и, как правило, подгружаются автоматически.

В редких случаях следует указывать конкретный файл, но писать самостоятельно его не нужно, лучше обратиться к разработчикам.

## 4. Запуск Natch

В комплект поставки инструмента *Natch* входят документация и исполняемые файлы. Исполняемые файлы разделены на две категории - собственно исполняемые файлы *Natch* и комплект необходимых для работы разделяемых библиотек. При запуске на машине без установленных библиотек может потребоваться явно указать путь к библиотекам, используя механизм `LD_PRELOAD`, а именно определить переменную `LD_LIBRARY_PATH`, указав ей путь к распакованному архиву с библиотеками следующим образом:

```
LD_LIBRARY_PATH=/path_to_libs ./natch_run.py path_to_image
```

### 4.1. Использование скрипта для генерации командных строк запуска

В поставку инструмента входит скрипт *natch\_run.py*, который нужен для генерации командных строк для запуска *Natch*, а так же для первоначальной настройки инструмента.

Перед работой нужно установить зависимости из списка *requirements.txt* с помощью команды:

```
pip3 install -r requirements.txt
```

Зависимости, описанные в файле *requirements.txt* общие для всех python-скриптов инструмента и достаточно уставить их один раз.

В процессе выполнения скрипта пользователь должен уточнять параметры запуска инструмента. На выходе сформируются три скрипта: *run\_record.sh* и *run\_replay.sh* для запуска *Natch* в режиме записи и воспроизведения работы соответственно, а так же *run\_qemu.sh* для запуска Qemu без *Natch*.

Инструмент *Natch* агрегирован в плагине *natch*, который подключается к Qemu.

Прежде чем перейти к описанию скрипта, рассмотрим опцию для подключения конфигурационного файла плагина *natch*. - *config* Задаёт файл конфигурации инструмента (подробнее в пункте [Основной конфигурационный файл](#)).

Пример командной строки, отвечающей за подключение плагина с опцией *config*:

```
-plugin natch,config=natch_config.cfg
```

Скрипт *natch\_run.py* имеет три параметра, один обязательный и два опциональных. Обязательный параметр это путь к образу системы, опциональные - путь к ядру (опция *kernel* эмулятора Qemu, если ядро ОС не загружается из образа системы) и название операционной системы, для которой будет использован инструмент (по умолчанию Linux, предоставляется выбор из следующих вариантов: Linux, FreeBSD). Так же скрипт можно запустить с опцией *-h* и получить справку по доступным опциям.

Описание команды и ее параметров представлено ниже:

```
natch_run.py image [-h] [-k KERNEL] [-o {Linux, FreeBSD}]
```

Пример запуска скрипта:

```
LD_LIBRARY_PATH=/path_to_libs ./natch_run.py <image> -o Linux
```

Список вопросов, которые будут заданы при выполнении скрипта: - *Путь к директории проекта*  
Необходимо указать полный путь или только название папки, которая будет создана для хранения всех скриптов, конфигураций и прочих файлов инструмента. Внутри этой директории будет создана еще одна папка с именем *output*, в которую будут помещены выходные файлы работы инструмента. Параметр является опциональным, в случае пропуска рабочий каталог будет

называться именем запускаемого образа гостевой системы. Если каталог уже существует, будет создан новый с постфиксом \*\_x\*, где x - это число от 1 до максимально возможного.

- **Количество оперативной памяти**

Необходимо указать объем оперативной памяти, выделяемый виртуальной машине. Указывается число в гигабайтах или мегабайтах с соответствующим постфиксом G или M, например 512M.

- **Сетевые опции**

В этом меню пользователь может указать порты, которые необходимо пробросить в виртуальную машину. Запрос на ввод портов появится при выборе ответа Y/y или при нажатии клавиши *Enter*. Порты следует указывать через запятую. Указываемые порты должны лежать в диапазоне [1 .. 9999]. Порт для обращения к машине извне будет формироваться по принципу: *user\_port + 10000*.

После этих вопросов скрипт попытается обратиться к утилите *qemu-img*, входящей в поставку эмулятора Qemu, чтобы создать оверлей для образа, необходимый для хранения состояний системы. Если оверлей создать не удалось, выполнение скрипта будет прервано.

Далее скрипт предложит создать конфигурационный файл для модулей. Если модули есть в наличии, то следует согласиться, в противном случае ввести *n* или *N*. Если конфигурационный файл нужен, будет предложено ввести путь к папке, содержащей модули.

Следующий вопрос коснется конфигурационного файла *task\_config.ini*. Если вы впервые создаете конфигурацию для образа, следует воспользоваться ответом по умолчанию (нет). На этом этапе будет запущен эмулятор для извлечения информации о структурах ядра. Если же вы уже работали с этим образом и получали конфигурационный файл с этого этапа - вопрос можно пропустить, но не забудьте скопировать *task\_config.ini* в рабочую директорию нового проекта. Если не уверены, что файл от нужного образа - лучше ответить нет и дождаться генерации файла.

Вопросы для пользователя закончились, далее при необходимости будет осуществлен настроечный запуск эмулятора, следует подождать пока он отработает и окно эмулятора закроется.

На этом настройка окончена. На экран будут выведены сгенерированные командные строки, а так же описание полученных скриптов.

Кроме командных строк будет сгенерирована заготовка конфигурационного файла *Natch*. В полученном файле описаны все возможные секции и поля с примерами заполнения, часть из которых закомментирована. При необходимости этот файл можно отредактировать - раскомментировать нужные секции и внести актуальные значения параметров. Имя конфигурационного файла задается по умолчанию, а именно *natch\_config.cfg*.

Помимо скрипта *natch\_run.py* в поставку инструмента входит скрипт *module\_config.py*. Этот скрипт запускается внутри основного, если пользователь соглашается создать конфигурационный файл для модулей. При необходимости его можно использовать отдельно, если, например, модули обновились, а проводить заново настройку инструмента не нужно.

Скрипт *natch\_run.py* расположен в директории *natch\_scripts*. Запускать скрипт можно из любого расположения, готовые скрипты для запуска инструмента и все конфигурационные файлы будут помещены в рабочей директории, указанной пользователем или назначенной автоматически.

## 4.2. Подготовка запуска Natch вручную

Скрипт *natch\_run.py* призван упростить пользователю работу с инструментом, однако можно формировать командные строки для запуска самостоятельно. Пример конфигурационного файла *Natch* можно взять из документации.

Пример командной строки для запуска *Natch*:

```
./qemu-system-x86_64 -hda debian.qcow2 -m 6G -monitor stdio -netdev user,id=net0 -device e1000,netdev=net0 -os-version Linux -plugin natch,config=natch_config.cfg
```

При первом ручном запуске *Natch* произойдет генерация конфигурационного файла *task\_config.ini*. Следует дождаться завершения работы эмулятора, после чего можно запускать его для работы.

Однако, рекомендуемым вариантом использования *Natch* является его запуск с использованием детерминированного воспроизведения.

### 4.3. Запуск Natch с использованием детерминированного воспроизведения

Детерминированное воспроизведение состоит из двух фаз: запись и воспроизведение. Во время записи необходимо выполнить сценарий, который будет анализироваться во время следующих запусков в режиме воспроизведения.

При использовании скрипта для генерации командных строк, строки запуска для записи и воспроизведения будут получены автоматически: *run\_record.sh* для записи и *run\_replay.sh* для воспроизведения.

Таким образом для работы с *Natch* нужно записать сценарий работы виртуальной машины с помощью скрипта *run\_record.sh* и воспроизводить его в дальнейшем с помощью скрипта *run\_replay.sh*.

Скрипт *run\_replay.sh* имеет необязательный параметр *snapshot\_name*. Например, если во время записи был сохранен снимок с именем *load*, то запустить воспроизведение с этого момента можно либо передав скрипту параметр *./run\_replay.sh load*, либо отредактировав скрипт путем замены значения переменной *SNAPSHOT* с *init* на *load*.

Непосредственно работа с инструментом *Natch* никак не изменится при использовании детерминированного воспроизведения. Изменения коснутся лишь строки запуска самого эмулятора.

Если по какой-то причине вы не используете скрипт *natch\_run.py*, ниже приведен пример строк запуска эмулятора в режимах записи и воспроизведения.

Пример командной строки для записи работы:

```
./qemu-system-x86_64 -m 4G \  
-icount shift=5,rr=record,rrfile=replay.bin \  
-drive file=debian10.qcow2,if=none,snapshot,id=img-direct \  
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay \  
-device ide-hd,drive=img-blkreplay \  
-netdev user,id=net0 \  
-device e1000,netdev=net0 \  
-object filter-replay,id=replay,netdev=net0 \  
-monitor stdio \  

```

Пример командной строки для воспроизведения работы:

```
./qemu-system-x86_64 -m 4G \  
-icount shift=5,rr=replay,rrfile=replay.bin \  
-drive file=debian10.qcow2,if=none,snapshot,id=img-direct \  
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay \  
-device ide-hd,drive=img-blkreplay \  
-netdev user,id=net0 \  
-device e1000,netdev=net0 \  
-object filter-replay,id=replay,netdev=net0 \  
-monitor stdio \  
-plugin natch,config=natch_config.cfg \  

```

## 5. Функциональные возможности Natch

### 5.1. Получение поверхности атаки

Основным результатом работы инструмента *Natch* является поверхность атаки. Поверхность атаки представлена набором процессов, модулей и функций, которые обрабатывали помеченные данные по время выполнения тестового сценария.

Для получения поверхности атаки можно воспользоваться командой монитора `natch_get_attack_surface <filename>`, либо завершить работу эмулятора и файлы с информацией сгенерируются автоматически.

Поверхность атаки разбита на два файла, в одном находятся модули, во втором функции. В обоих случаях сущности привязаны к процессу. К введенному пользователем имени файла добавляются соответствующие суффиксы: `<filename>_modules.txt` и `<filename>_functions.txt`. Автоматически сгенерированные файлы называются *surface\_modules.txt* и *surface\_functions.txt*.

Фрагмент файла *surface\_modules.txt*:

Task docker

Module /lib/x86\_64-linux-gnu/libpthread.so.0 0x7f6bc94a4000

Module /lib/x86\_64-linux-gnu/libc.so.6 0x7f6bc92de000

Task containerd-shim

Module 0x0

Task wget2

Module /lib/x86\_64-linux-gnu/libpsl.so.5 0x7f921b147000

Module 0x0

Module 0x7f921b419000

Module files/wget2 0x55b58a293000

Module wget2/build/lib/libwget.so.1 0x7f921b3a2000

Module /lib/x86\_64-linux-gnu/libpthread.so.0 0x7f921af77000

Module /lib/x86\_64-linux-gnu/libresolv.so.2 0x7f9219cce000

Module /lib/x86\_64-linux-gnu/libc.so.6 0x7f921adb6000

Module /lib/x86\_64-linux-gnu/libz.so.1 0x7f921b15a000

Фрагмент файла *surface\_functions.txt*:

Task docker

Module /lib/x86\_64-linux-gnu/libpthread.so.0 0x7f6bc94a4000

Function pthread\_create 0x7f6bc94ac280 43

Module /lib/x86\_64-linux-gnu/libc.so.6 0x7f6bc92de000

Function 0x7f6bc94aa390 5

Function 0x556426eee1b0 54

Function 0x556426eee180 1

Task containerd-shim

Module 0x0

Function 0xffffffff94af9700 453

Function 0xffffffff94c001b8 54

Function 0xffffffff94e03000 1

Task wget2

Module files/wget2 0x55b58a293000

Function 0x7f921b3b1210 2

Function process\_response\_header 0x55b58a2a9560 8 wget2/src/wget.c:1665

Function prepare\_file 0x55b58a2a7720 1 wget2/src/wget.c:3214

Function \_host\_hash 0x55b58a2a2400 28 wget2/src/host.c:81

```

Function 0x7f921b3bcbe0 4
Function get_header 0x55b58a2a85a0 5 wget2/src/wget.c:3485
Function my_free 0x55b58a2ad2d0 1 wget2/src/options.c:4232
Function plugin_db_forward_downloaded_file 0x55b58a2a4510 1 wget2/src/plugin.c:556
Function hash_iri 0x55b58a2a1880 56 wget2/src/blacklist.c:171
Function process_response 0x55b58a2aaf40 8 wget2/src/wget.c:1980

```

Число после описания каждой функции обозначает количество ее обращений к помеченным данным. Это позволяет выбирать функции, наиболее интенсивно задействованные в обработке данных тестового сценария.

## 5.2. Подробная трасса помеченных данных

Для более детального анализа может потребоваться больше информации, которую можно получить с помощью опции конфигурационного файла *Modules/log*.

В генерируемом логе на каждое обращение к помеченной памяти формируется расширенный набор данных.

Фрагмент лога для одного обращения:

```

Load:
Process name: wget2 cr3: 0x1b5a5c000
Tainted access at 00007f921af88896
Access address 0x7f921a504b08 size 8 taint 0xfcfcfcfc
icount: 21216379863
Module name: /lib/x86_64-linux-gnu/libpthread.so.0 base: 0x00007f921af77000
Call stack:
  0: 00007f921af88896 in func 00007f921b3b1c40 wget2/build/lib/libwget.so.1::recvfrom
  1: 00007f921b3c5307 wget2/libwget/net.c:861 in func 00007f921b3b0860 wget2/build/lib/libwget.so.1::wget_tcp_read
  2: 00007f921b3bdbda wget2/libwget/http.c:990 in func 000055b58a29f350 files/wget2::wget_http_get_response_cb
  3: 000055b58a2ac0ec wget2/src/wget.c:4017 in func 000055b58a2ac0e0 files/wget2::http_receive_response wget2/src/wget.c:4016
  4: 000055b58a2ac654 wget2/src/wget.c:2266 in func 000055b58a2ac2b0 files/wget2::downloader_thread wget2/src/wget.c:2250
  5: 00007f921af7efa1 in func 00007f921af7eeb0 /lib/x86_64-linux-gnu/libpthread.so.0
  6: 00007f921aeaf4cd

```

Не рекомендуется включать эту опцию по умолчанию, поскольку файл получается ощутимого размера (сотни байт на каждое обращение к помеченным данным).

## 5.3. Получение областей помеченной памяти для функций

Инструмент позволяет получить лог вызовов функций с диапазонами адресов записанных и прочитанных помеченных данных. Для получения лога необходимо использовать опцию конфигурационного файла *Modules/params\_log*. Эта опция задает имя файла, куда будет записан лог с параметрами функций.

Выходной файл содержит диапазоны адресов и типы операций, выполненных с помеченными данными (r=чтение, w=запись). Также выводится стек вызовов на момент выхода из функции.

Фрагмент выходного файла:

```

0xffffffff82dfc6f0 vmlinux:eth_type_trans
  0xffff88800e723840 8 bytes r
  0xffff88800e72384c 2 bytes r

```



```

enter_icount: 58799550862
exit_icount: 58799551027
0: ffffffff82dfc963 in func ffffffff82dfc6f0 vmlinux::eth_type_trans
1: ffffffff827cd3b6 in func ffffffff827ccec0 vmlinux::e1000_clean_rx_irq
2: ffffffff827d544c in func ffffffff827d4c50 vmlinux::e1000_clean
3: ffffffff82d1ea25 in func ffffffff82d1e6c0 vmlinux::net_rx_action
4: ffffffff83a001b0 in func ffffffff83a00000 vmlinux::__do_softirq
5: ffffffff83800f8d
0xffffffff81232e20 vmlinux:lock_acquire
0xffff88806d009a90 8 bytes rw
enter_icount: 58799552881
exit_icount: 58799554333
0: ffffffff81232ffd in func ffffffff81232e20 vmlinux:lock_acquire
1: ffffffff8302c830 in func ffffffff8302c670 vmlinux::inet_gro_receive
2: ffffffff82d200cb in func ffffffff82d1f440 vmlinux::dev_gro_receive
3: ffffffff82d22885 in func ffffffff82d22680 vmlinux::napi_gro_receive
4: ffffffff827cd485 in func ffffffff827ccec0 vmlinux::e1000_clean_rx_irq
5: ffffffff827d544c in func ffffffff827d4c50 vmlinux::e1000_clean
6: ffffffff82d1ea25 in func ffffffff82d1e6c0 vmlinux::net_rx_action
7: ffffffff83a001b0 in func ffffffff83a00000 vmlinux::__do_softirq
8: ffffffff83800f8d

```

## 5.4. Получение графов взаимодействий процессов и модулей

*Natch* позволяет получить историю распространения помеченных данных между процессами. Каждая строка лог-файла описывает передачу данных между двумя процессами, либо между процессом и файлом.

Для определения взаимодействий *Natch* выделяет дополнительную теньюю память объемом в два раза больше, чем объем основной памяти, выделенной гостевой системе. В эту память для каждого физического адреса записывается идентификатор процесса, который последним записал помеченные данные на этот адрес. Взаимодействие определяется, когда процесс читает помеченные данные из ячейки памяти, записанной другим процессом. Взаимодействия также имеют веса, соответствующие количеству передаваемых данных. Однако следует отметить, что случаи, когда процесс читает 100 байт другого процесса и когда читает 1 байт 100 раз, имеют один и тот же вес.

Также *Natch* умеет определять некоторые интерфейсы передачи данных. По системным вызовам отслеживаются взаимодействия процессов с файлами и сокетами. Через структуры ядра определяются области разделяемой и приватной памяти.

Для включения функции построения графов используется секция *TaintedTask* в основном конфигурационном файле. Опция *task\_graph* отвечает за граф процессов, опция *module\_graph* за граф модулей. Граф модулей строится теми же методами, что и граф процессов. Заметим, что включение обеих опций одновременно приведет к значительному увеличению объема потребления памяти. Результаты записываются соответственно в файлы *task\_graph.json* и *module\_graph.json* при завершении работы эмулятора.

Выходной файл на верхнем уровне представляет собой список ребер графа. У каждого ребра есть поля *source*, *destination* и *score*. Первые два описывают узлы, между которыми происходит передача помеченных данных. Поле *score* содержит количество передаваемых байт между процессами. Если граф строился во время воспроизведения, то для каждого ребра присутствует поле *icount*, которое описывает диапазон времени, в которое происходила передача данных. У некоторых ребер присутствует поле *extra*, которое содержит описание способа передачи данных. Описание способа передачи состоит из одного поля *type*, которое может принимать значения *shared-memory* для передач через разделяемую память и *private-memory* для передач через приватную память (Например, передача данных от родительского процесса к дочернему во время вызова *fork*). Описание узла графа имеет поле *type*, описывающее тип узла. Если узел является

источником помеченных данных, то его описание содержит поле *taint\_source* в значении *true*. Далее идет описание каждого типа узлов графа и его параметров:

- *file* - обычный файл. Поле *name* содержит полное имя файла.
- *tcp*, *udp*, *tcpv6*, *udpv6* - сетевые сокет TCP и UDP для IPv4 и IPv6 соответственно. Поля *ip* и *port* содержат соответственно *ip* адрес и порт сокета.
- *unix* - unix сокет. Поле *name* содержит имя файла из параметров сокета, либо строку *pair + число*, если сокет не имеет имени файла (Создан системным вызовом *socketpair*)
- *netlink* - netlink сокет. Поле *name* содержит индивидуальный адрес сокета netlink.
- *socket* - остальные виды сокетов. Поле *name* содержит название типа сокета. Наличие такого узла в графе говорит о том, что обработка параметров данного типа сокетов в настоящее время не реализована.
- *pipe* - неименованный канал. Поле *name* содержит строку *pair + число*.
- *network* - сеть, источник помеченных данных. Поле *protocol* описывает номер *ip* протокола в сетевых пакетах. Поле *port\_in* описывает входящий порт, поле *port\_out* исходящий порт.
- *user-process* - пользовательский процесс. Поле *name* содержит имя процесса. Поле *proc* содержит адрес структуры *task\_struct*.
- *kernel-process* - процесс ядра. Поля совпадают с *user-process*.
- *module* - модуль (Только для *module\_graph*). Поле *name* содержит имя модуля, поле *address* - адрес модуля.

Пример выходного файла:

```
{ "icount": { "start": 16638764870, "final": 16638764870 }, "source": { "name": "client_tcp", "proc": "0xffff941c3bd73700", "type": "user-process" }, "destination": { "name": "server_tcp_s", "proc": "0xffff941c3bd75280", "type": "user-process" }, "score": 2 },
{ "icount": { "start": 16638780181, "final": 16638780181 }, "source": { "name": "server_tcp_s", "proc": "0xffff941c3bd75280", "type": "user-process" }, "destination": { "port": 1234, "ip": "127.0.0.1", "type": "tcp" }, "score": 2068 },
{ "icount": { "start": 16638780181, "final": 16638780181 }, "source": { "port": 1234, "ip": "127.0.0.1", "type": "tcp" }, "destination": { "name": "client_tcp", "proc": "0xffff941c3bd73700", "type": "user-process" }, "score": 2068 },
{ "icount": { "start": 16883848026, "final": 16883848026 }, "source": { "name": "/home/nat/bin/scripts/index.html", "type": "file" }, "destination": { "name": "args1", "proc": "0xffff941c3b4f3700", "type": "user-process" }, "score": 2050 },
{ "icount": { "start": 16883982774, "final": 16883987750 }, "extra": { "type": "private-memory" }, "source": { "name": "args1", "proc": "0xffff941c3b4f3700", "type": "user-process" }, "destination": { "name": "args1", "proc": "0xffff941c3bd76e00", "type": "user-process" }, "score": 4104 },
{ "icount": { "start": 16885074396, "final": 16885511425 }, "extra": { "type": "private-memory" }, "source": { "name": "args1", "proc": "0xffff941c3bd76e00", "type": "user-process" }, "destination": { "name": "args2", "proc": "0xffff941c3bd76e00", "type": "user-process" }, "score": 2155 },
{ "icount": { "start": 16890900062, "final": 16890900062 }, "source": { "name": "args2", "proc": "0xffff941c3bd76e00", "type": "user-process" }, "destination": { "name": "/tty1", "type": "file" }, "score": 100 },
{ "icount": { "start": 16909717359, "final": 16909717359 }, "source": { "name": "wget", "proc": "0xffff941c3bd71b80", "type": "user-process" }, "destination": { "name": "/home/nat/bin/scripts/index.html", "type": "file" }, "score": 2058 },
{ "icount": { "start": 16909717359, "final": 16909717359 }, "source": { "name": "/home/nat/bin/scripts/index.html", "type": "file" }, "destination": { "name": "shm2", "proc": "0xffff941c3b53a940", "type": "user-process" }, "score": 2058 },
{ "icount": { "start": 16915543303, "final": 16915547137 }, "extra": { "type": "shared-memory" }, "source": { "name": "shm2", "proc": "0xffff941c3b53a940", "type": "user-process" }, "destination": { "name": "shm3", "proc": "0xffff941c3b538000", "type": "user-process" }, "score": 100 },
{ "icount": { "start": 16921037754, "final": 16921038372 }, "extra": { "type": "shared-memory" }, "source": { "name": "shm3", "proc": "0xffff941c3b538000", "type": "user-process" }, "destination": { "name": "shm1", "proc": "0xffff941c3b539b80", "type": "user-process" }, "score": 236 },
```

```
{"icount": {"start": 16926559320, "final": 16926559320}, "source": {"name": "shm1", "proc": "0xffff941c3b539b80", "type": "user-process"}, "destination": {"name": "/tty1", "type": "file"}, "score": 100},
```

## 5.5. Пометка файлов

*Natch* может пометать отдельные файлы в гостевой системе и отслеживать модули и функции, которые были затронуты в результате работы с ними. При этом берутся в расчет только операции чтения. Исполняемые файлы пометаться таким способом не будут.

Включить плагин и задать необходимые для пометки файлы можно в основном конфигурационном файле *Natch*. Если секция *TaintFile* не была описана, для пометки файлов необходимо подключить плагин *taint\_file*. Сделать это можно во время работы эмулятора командой монитора `plugins_load taint_file`.

Плагин предоставляет набор функций, в частности, отслеживание файла включается командой `taint_file <filename>`. Имя файла можно указывать без пути или с ним, но во втором случае он может быть не найден из-за особенностей реализации.

После этого необходимо повзаимодействовать с помеченным файлом и оценить результат с помощью команды `natch_get_attack_surface <filename>` или завершив работу эмулятора.

### 5.5.1. Пометка входящих сетевых пакетов

*Natch* способен пометать весь сетевой трафик, который приходит в виртуальную машину извне. Пометка полностью локального трафика пока не поддерживается.

Для управления пометкой пакетов используется секция *[Ports]* конфигурационного файла (подробнее в пункте [Основной конфигурационный файл](#) секция **Ports**).

Пакеты помечаются целиком, вместе с заголовком второго уровня. Возможны следующие варианты работы:

- Помечать все входные пакеты. `ip_protocol=-1 + (in=-1 или out=-1)`
- Помечать все UDP-пакеты. `ip_protocol=17`
- Помечать все TCP-пакеты. `ip_protocol=6 + (in=-1 или out=-1)`
- Помечать все HTTP-пакеты от внешнего веб-сервера. `ip_protocol=6 + out=80`
- Помечать все ICMP-пакеты. `ip_protocol=1`

## 5.6. Возможности в режиме воспроизведения работы виртуальной машины

### 5.6.1. Логирование сетевых пакетов

В режиме воспроизведения существует возможность логирования обработанных сетевых пакетов. Для этого в основной конфигурационный файл *Natch* необходимо добавить секцию *NetLog* (пример приведен в пункте [Дополнительная ручная настройка](#)).

Информация о пакетах вносится в лог в Json формате и имеет следующую структуру:

```
[
  {
    "type" : "net"
    "icount" : uint64
    "addr" : String
    "packet" : String
  },
  ...
]
```

Поля структуры:

- *type* - тип логированного пакета, в данный момент поддерживается только net
- *icount* - шаг воспроизведения
- *addr* - шестнадцатеричный адрес пакета в физической памяти, записанный в формате строки
- *packet* - побайтовое содержимое пакета в шестнадцатеричном формате в виде строки.

## 5.7. Анализ покрытия бинарного кода

Плагин *coverage* используется для сбора покрытия исполняемого кода.

Опции плагина:

- *file* Задаёт название файла, куда будут записываться данные о покрытии кода (по умолчанию *coverage.cov64*)
- *taint* Переключает режимы сбора покрытия. При установке в true, сбор ведётся только для помеченных данных. Иначе, для всех выполненных базовых блоков (ББ), которые относятся к какому-либо модулю (данный плагин не собирает покрытия для ББ, которые не относятся ни к одному модулю).

Пример командной строки для сбора покрытия всех выполненных ББ в режиме воспроизведения выглядит следующим образом:

```
./qemu-system-x86_64 -m 4G \  
-monitor stdio \  
-os-version Linux \  
-drive file=debian10_w_gui.diff,if=none,id=img-direct \  
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay \  
-device ide-hd,drive=img-blkreplay \  
-netdev user,id=net0 \  
-device e1000,netdev=net0 \  
-object filter-replay,id=replay,netdev=net0 \  
-icount shift=5,rr=replay,rrfile=replay.bin,rrsnapshot=snap \  
-plugin natch,config=natch_config.cfg \  
-plugin coverage,taint=false \  

```

### 5.7.1. Формат выходного файла

Выходной файл начинается со списка загруженных модулей, из которого собирается информация о покрытии. Каждый элемент этого списка содержит следующую информацию о модуле:

- Идентификатор
- Адрес его начала и конца
- Адрес начала секции *.text* (если данная информация была указана в конфигурационном файле)
- Идентификатор и имя процесса, в котором модуль был выполнен
- Путь, по которому модуль расположен на диске

После списка загруженных модулей, находится таблица, содержащая ББ, которые были выполнены при сборе информации о покрытии.

Каждый ББ представляет собой двоичную структуру, размером 8 байт, со следующими полями:

- 4 байта : Смещение от начала модуля, к которому принадлежит ББ
- 2 байта : Размер ББ
- 2 байта : Идентификатор модуля, в котором находится ББ

Каждый ББ встречается в логе ровно один раз, независимо от того, сколько раз он был выполнен.

## 5.7.2. Анализ выходного файла

Для удобного анализа выходного файла можно воспользоваться скриптом *coverage.py*, который раскрашивает выполненный код в IDA Pro и выводит таблицу с покрытием функций. На данный момент скрипт протестирован и гарантированно работает на версиях 7.2 и 7.6, теоретически и на тех, что между ними.

Чтобы использовать скрипт *coverage.py*, необходимо:

1. Открыть интересующий двоичный файл в IDA Pro.
2. Для совпадения адресов в Qemu и IDA Pro необходимо выполнить Rebase (Edit -> Segments -> Rebase program). Рекомендуется использовать адрес начала модуля.
3. Импортировать скрипт в окно File -> Script Command.
4. Убедиться, что выбран язык сценариев Python.
5. Нажать "Выполнить" и выбрать файл с покрытием в формате *.cov64*.
6. В появившемся окне выбрать интересующие процессы.

## 6. Графический интерфейс для анализа SMatch

*SMatch* – это инструмент с графическим интерфейсом, предназначенный для постобработки и отображения данных, полученных от инструмента *Natch*. Работать с этим интерфейсом можно через браузер.

### 6.1. Системные требования

Тестирование инструмента проводилось на ОС Ubuntu 20.04 и Ubuntu 22.04. В качестве браузеров использовались Google Chrome версии 105.0.5148.2 и выше и Mozilla Firefox версии 101.0 и выше.

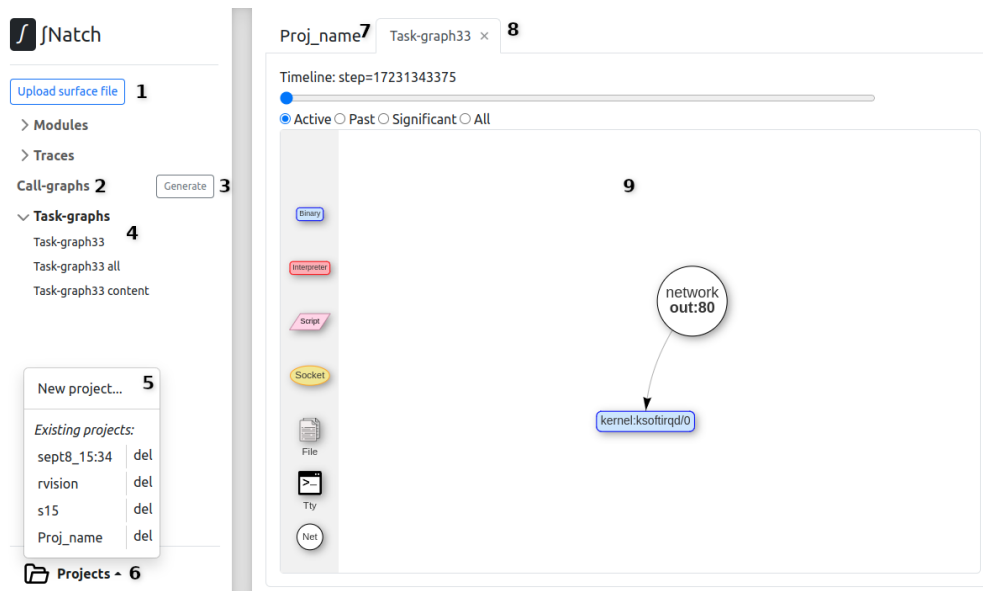
Необходимые для функционирования *SMatch* пакеты будут автоматически установлены в ходе первоначальной настройки.

### 6.2. Настройка и запуск

Для работы графического интерфейса *SMatch* необходимо подготовить окружение, а именно установить ряд пакетов для Ubuntu и Python, а также создать базу данных. Все это делается автоматически с помощью скрипта *snatch\_setup.sh*. Для установки потребуется пароль для *sudo*. Пакеты для Python устанавливаются в виртуальную среду, таким образом пакеты на вашей машине не будут переустанавливаться и конфликтовать. Процесс займет некоторое время, запускать скрипт нужно только при первом использовании.

Помимо скрипта для настройки окружения в пакет поставки входят скрипты *snatch\_run.sh* и *snatch\_stop.sh* для запуска и остановки *SMatch* соответственно. Скрипт *snatch\_run.sh* запускает необходимые для работы службы, а также открывает браузер с интерфейсом. В терминал, из которого был запущен скрипт, будут приходить сообщения от сервера, однако, он свободен для использования, поэтому по окончании работы из него же можно запустить скрипт *snatch\_stop.sh* для остановки служб. Запускать *snatch\_stop.sh* следует всегда, в противном случае процессы останутся висеть в памяти вашего компьютера до перезагрузки.

Все скрипты для *SMatch* можно запускать из любого расположения. Если по какой-то причине при запуске *snatch\_run.sh* страница в браузере не загрузилась, но при этом службы запустились, следует обновить страницу.



Окно SMatch

На данном рисунке показано окно *SMatch*, где цифрами обозначены основные элементы интерфейса:

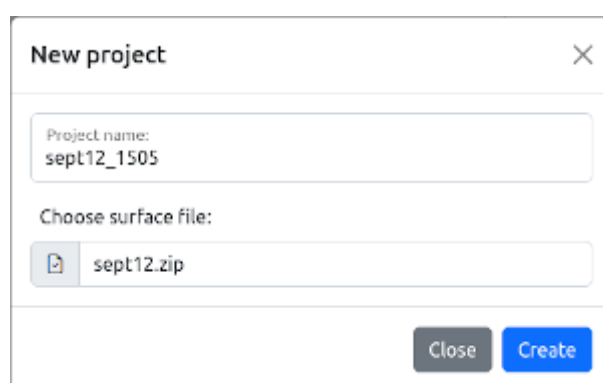
1. Кнопка для повторной загрузки архив с файлами поверхности атаки (далее просто: "архив")
2. Пункт меню, открывающий граф вызовов

3. Кнопка генерации графа вызовов
4. Список графов процессов
5. Всплывающее меню для взаимодействия с проектами (создание нового, открытие существующих и удаление существующих)
6. Кнопка открытия меню взаимодействия с проектами
7. Название открытого проекта
8. Открытые вкладки с содержимым (таким как: графы вызовов и графы процессов)
9. Окно отображения содержимого

Для создания нового проекта необходимо нажатием на (6) открыть меню проектов и выбрать *New project...*. Для переключения между существующими проектами необходимо из меню проектов в списке *Existing projects*: выбрать название интересующего проекта. Для удаления проекта достаточно в списке проектов нажать на кнопку *del* рядом с именем удаляемого проекта.

При нажатии на (1) в уже открытом проекте пользователю будет предложено выбрать новый архив (данный архив получается в результате работы *Natch*), после чего текущие данные в проекте будут стёрты и заменены полученными в результате обработки предоставленного файла.

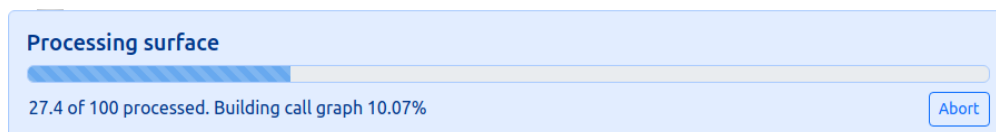
При создании нового проекта происходит открытие модального окна:



*Создание проекта*

В данном окне пользователь может ввести имя нового проекта в соответствующее поле *Project name*, а также выбрать архив поверхности атаки в *Choose surface file* (данный архив получается в результате работы *Natch*). Если оба этих поля заполнены, то становится активна кнопка *Create*, по нажатию на которую произойдет закрытие данного модального окна, создание нового проекта, автоматическое переключение на него и запуск процесса обработки архива. При нажатии на кнопку *Close* происходит закрытие модального окна.

Во время обработки архива поверхности атаки в верхней части экрана появляется шкала прогресса следующего вида:



*Шкала прогресса*

В процессе обработки текст, сопровождающий шкалу прогресса, будет изменяться таким образом, чтобы оповещать об актуальном статусе выполняемой задачи. Нажатие на кнопку *Abort* аварийно прерывает процесс обработки, и в данном случае корректное функционирование полученных данных не может быть гарантировано.

Основными интерактивными элементами, получаемыми в результате обработки архива поверхности атаки и доступными в интерфейсе *SNatch*, являются: граф вызовов и граф процессов.

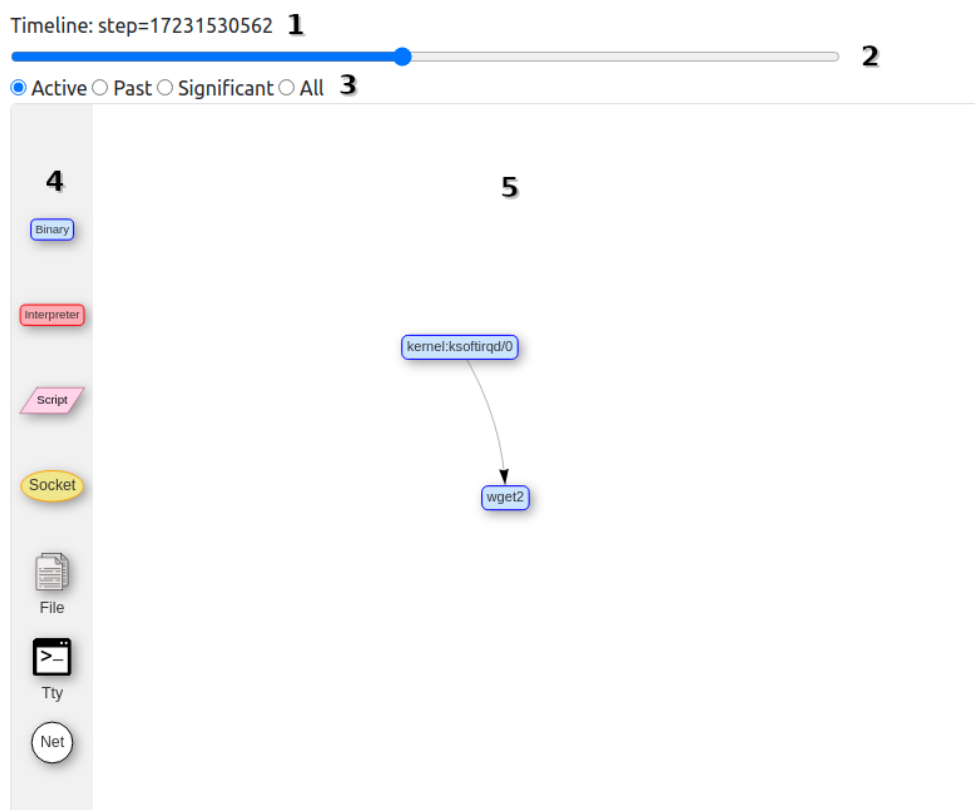
В процессе первоначальной обработки архива поверхности атаки граф вызовов не создаётся. Чтобы граф вызовов появился в проекте, необходимо нажать на кнопку *Generate* (3). По завершении процесса генерации в боковом меню проекта под пунктом *Call-graphs* появится строка *call-graph* с цифровым идентификатором. Для открытия графа вызовов достаточно нажатие на него в меню. После открытия появится новая вкладка с соответствующим названием, а в окне отображения содержимого откроется граф, представленный в виде древовидной структуры:



Граф вызовов

В корне отображаемых деревьев располагается идентификатор контекста, далее же идут адреса вызываемых функций. Интерактивное взаимодействие с графом достигается с помощью нажатия на кнопки "+"/"-", которые раскрывают/сворачивают данную ветку. Также по нажатию на символ из трех горизонтальных отрезков рядом с корнем дерева происходит полное раскрытие или же сворачивание выбранного дерева.

В меню графа процессов есть три пункта: *task graph* (процессы с идентичными именами сливаются в один примитив в пределах одного контейнера), *task graph all* (процессы не сливаются), *task graph content* (json представление графа). После выбора одного из этих пунктов открывается новая вкладка с соответствующим названием, а в окне отображения содержимого откроется граф, представленный следующим образом:



Граф процессов

В данном окне присутствуют следующие элементы:

1. Номер активного шага выполнения
2. Шкала выбора активного шага выполнения
3. Выбор режима отображения графа
4. Легенда графа

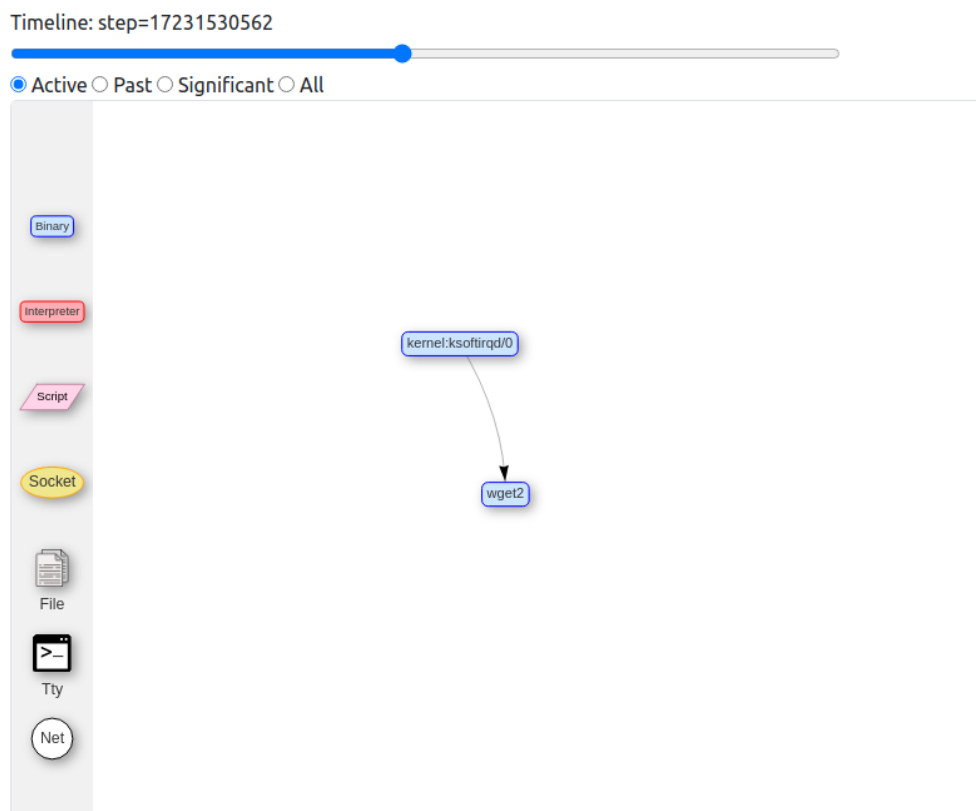


## 5. Граф процессов

Значение в (1) отображается в соответствии с выбранным на (2) шагом. Перемещение по (2) можно осуществлять как перетаскиванием мышью, так и нажатием стрелок на клавиатуре при активной шкале. Актуальные значения в легенде (4) следующие: бинарный файл, интерпретатор, файл скрипта (связанный с интерпретатором), сокет, текстовый файл, терминал, узел сети. В (5) отображаются цветные (и серые в некоторых режимах) взаимодействующие узлы, взаимодействие отображается стрелками. Толщина стрелки соответствует относительному объему данных при взаимодействии. Все узлы в (5) интерактивны, их расположение можно изменять удобным для пользователя образом.

При переключении режимов (3) отображаемые на (5) элементы будут изменяться соответствующим образом:

*Active* – отображаются элементы (узлы и стрелки), которые задействованы на текущем шаге.

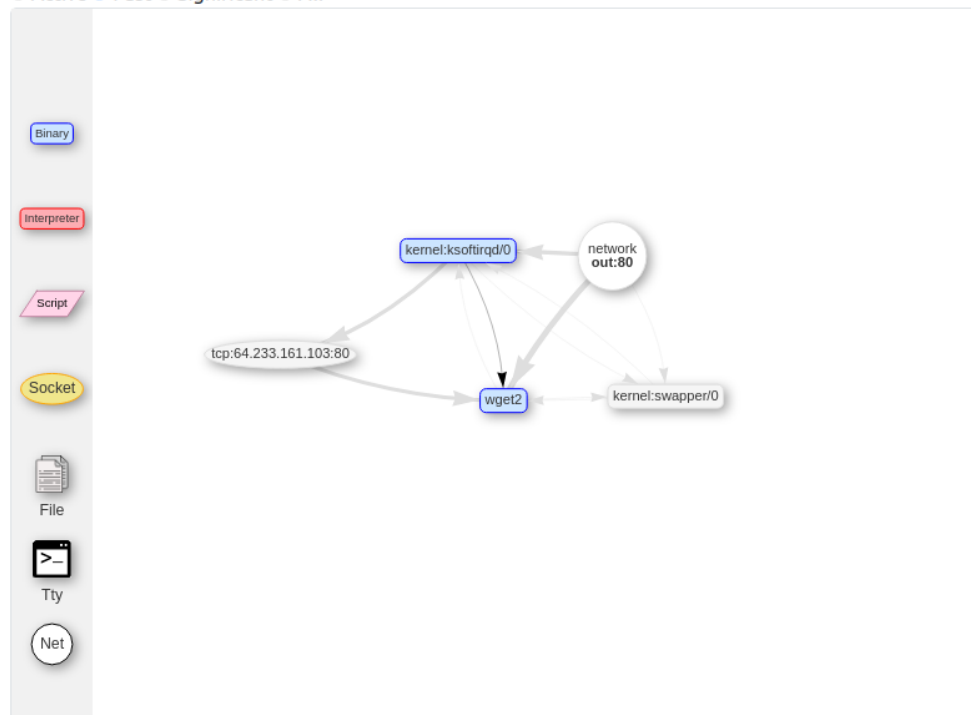


*Режим Active*

*Past* – отображаются элементы, которые задействованы на текущем шаге, а также те, которые произошли в прошлом (узлы и стрелки серого цвета).

Timeline: step=17231530562

☐ Active ☒ Past ☐ Significant ☐ All

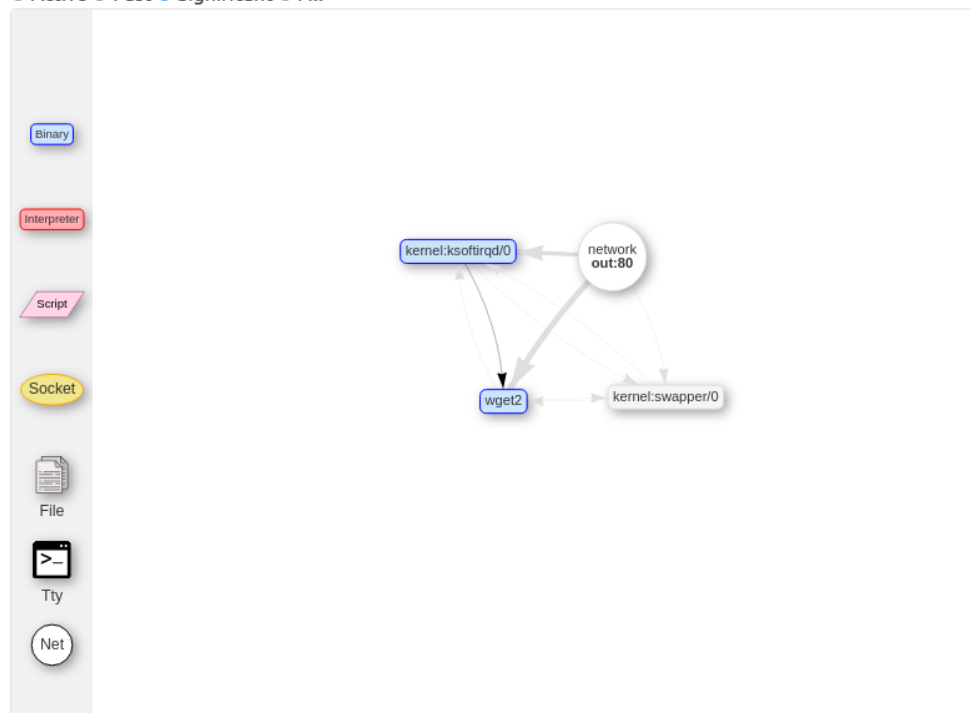


*Режим Past*

*Significant* – отображаются элементы, которые задействованы на текущем шаге, а также те узлы, которые были задействованы в прошлом и при этом еще будут задействованы в будущем, и прошедшие стрелки между ними (узлы и стрелки серого цвета).

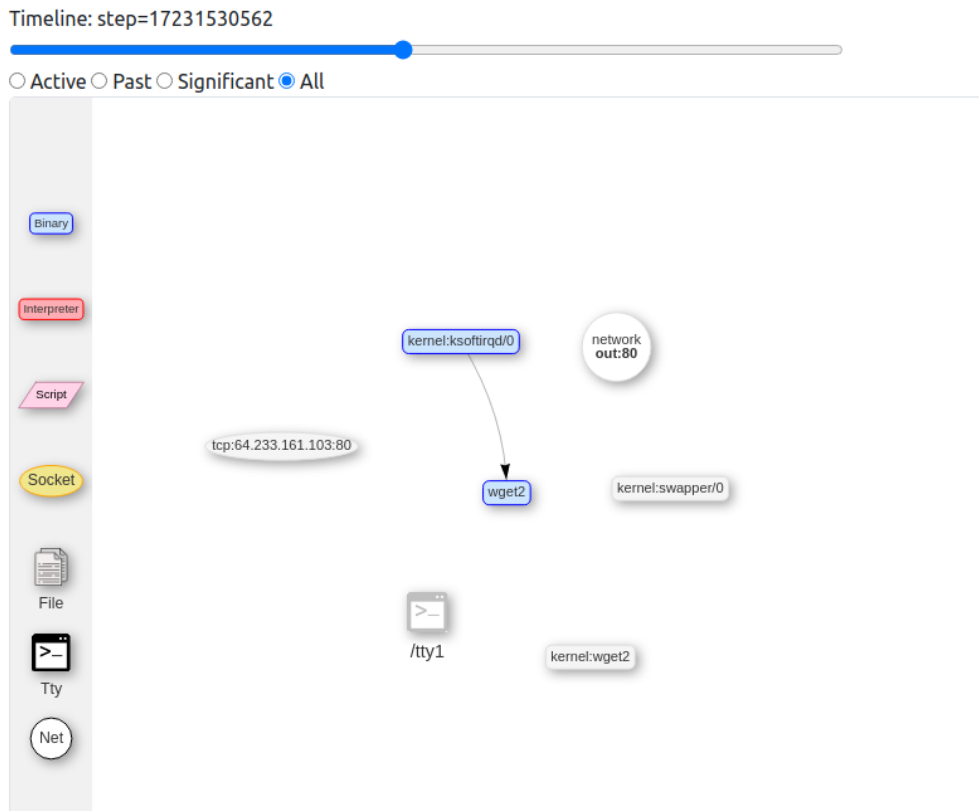
Timeline: step=17231530562

☐ Active ☐ Past ☒ Significant ☐ All



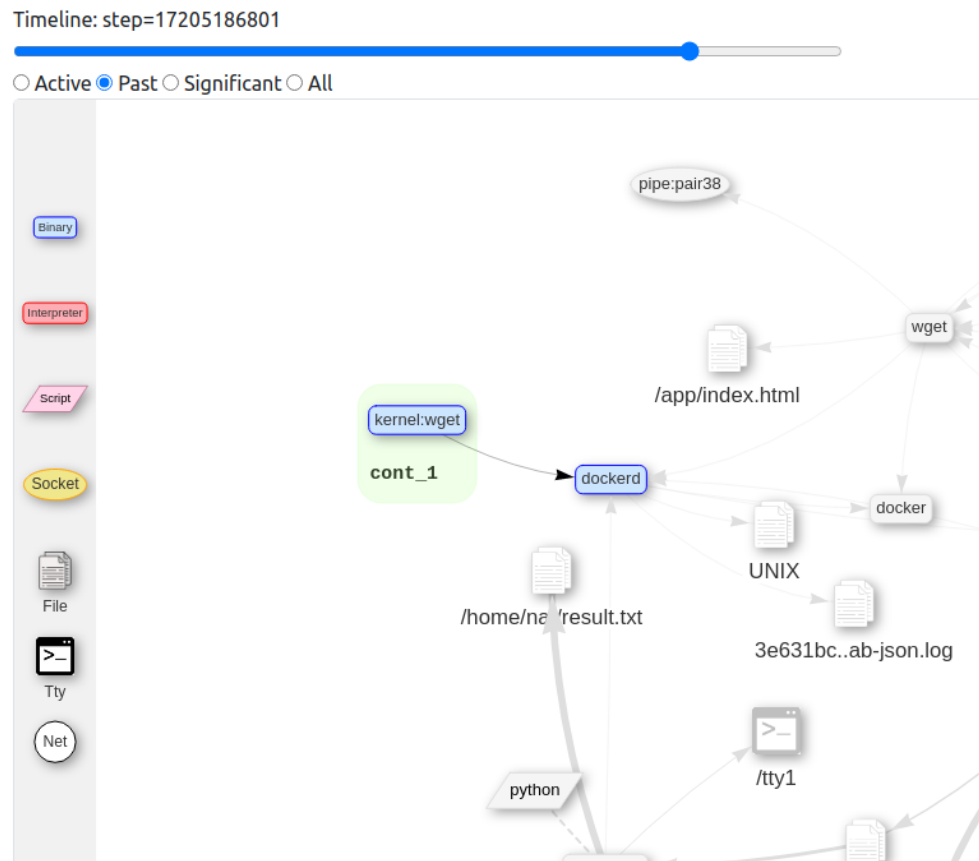
*Режим Significant*

All – отображаются элементы, которые задействованы на текущем шаге, а также те узлы (но не стрелки), которые вообще существуют на схеме (узлы серого цвета).



### Режим All

В случае наличия информации о контейнерах, таковые будут отображены на графе в виде зеленых блоков, содержащих соответствующие им узлы (`cont_1` на рисунке):



## Отображение контейнеров

Также по двойному нажатию на стрелку в графе процессов в новой вкладке открывается соответствующий этому взаимодействию стек вызовов:

```
• Load:
Process name: context: 0x1364ee000
Tainted access at 0x7fba626f23f9
Access address 0x7fba5c036f13 size 1 taint 0xff
Icount: 17232552742
Module base: 0x7fba626f0000
Call stack:
  ◦ 0x7fba626f23f9 in func 0x7fba626f2950
  ◦ 0x7fba6295f551 /home/nat/wget2/libwget/cookie.c:133 in func 0x7fba6295f540 libwget.so.1.0.0::wget_cookie_check_psl
    /home/nat/wget2/libwget/cookie.c:126
  ◦ 0x7fba6295f5bb /home/nat/wget2/libwget/cookie.c:167 in func 0x7fba6295f5a9 libwget.so.1.0.0::wget_debug_printf
    /home/nat/wget2/libwget/cookie.c:161
  ◦ 0x7fba6295fb7b /home/nat/wget2/libwget/cookie.c:200 in func 0x7fba6295fb75 libwget.so.1.0.0::wget_vector_get
    /home/nat/wget2/libwget/cookie.c:200
  ◦ 0x5628c1d90694 /home/nat/wget2/src/wget.c:1724 in func 0x5628c1d90689 wget2::wget_cookie_normalize_cookies
    /home/nat/wget2/src/wget.c:1724
  ◦ 0x5628c1d93671 /home/nat/wget2/src/wget.c:2358 in func 0x5628c1d93659 wget2::http_receive_response /home/nat/wget2/src/wget.c:2346
  ◦ 0x7fba62527fa1 in func 0x7fba62527eb0
  ◦ 0x7fba624584cd in func 0x7fa224ebc9a0
```

## Стек вызовов

# Приложение 1. Настройка окружения для использования лицензированного Natch

Для использования лицензированной версии инструмента *Natch* необходимо иметь HASP ключ или сетевую лицензию. При наличии ключа следует только выполнить пункт 1 из этого приложения. В случае с сетевой лицензией необходимо выполнить все нижеописанные действия.

В качестве хостовой системы предлагается использовать Ubuntu 20.04.

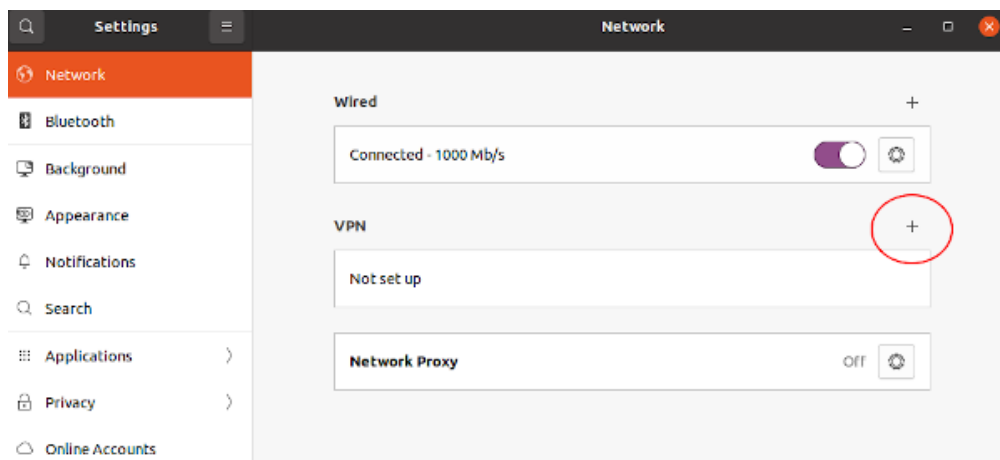
Все необходимые файлы для установки и настройки будут предоставлены пользователю, а именно пакет `aksusbd_8.31-1_amd64.deb`, `sailor-license.ovpn` и логин/пароль для подключения.

1. Установить deb пакет `aksusbd_8.31-1_amd64.deb` с помощью команды:

```
sudo dpkg -i aksusbd_8.31-1_amd64.deb
```

1. Открыть браузер и ввести строку `localhost:1947`
2. Откроется главная страница Sentinel Admin Control Center, на которой нужно перейти в раздел Configuration, в нем найти раздел Access to Remote License Managers.
3. Убедиться, что в полях Allow Access to Remote Licenses и Broadcast Search to Remote Licenses стоят галочки.
4. В поле Remote License Search Parameters ввести `license.intra.ispras.ru` и нажать кнопку Submit.
5. Создать VPN соединение.

Для этого создания VPN соединения необходимо открыть настройки операционной системы и перейти в раздел *Network*. В секции VPN нажать на плюсик.



*Настройки сети*

Появится окно для добавления новой конфигурации VPN. Нужный вариант *Import from file..*, куда следует передать входящий в поставку файл `sailor-license.ovpn`. В предлагаемой ОС OpenVPN установлен по умолчанию.

The screenshot shows a dark-themed dialog box titled 'Add VPN'. At the top, there are two buttons: 'Cancel' and 'Add VPN'. Below the title bar, there are three options listed with their descriptions:

- OpenVPN**  
Compatible with the OpenVPN server.
- Point-to-Point Tunneling Protocol (PPTP)**  
Compatible with Microsoft and other PPTP VPN servers.
- Import from file...** (This option is circled in red)

### *Импорт настроек VPN*

После импорта файла появится окно настройки соединения, в которое необходимо вписать логин и пароль, так же входящие в поставку. Далее осталось нажать кнопку *Add* в верхнем правом углу и конфигурация будет создана.

The screenshot shows the 'Add VPN' dialog box with the 'Import from file...' option selected. The dialog is now open to the configuration screen. At the top, there are three tabs: 'Identity', 'IPv4', and 'IPv6'. The 'Identity' tab is selected. Below the tabs, there are several fields and sections:

- Name:** A text field containing 'sailor-license'.
- General:** A section containing a 'Gateway' field with the value 'sailor.ispras.ru:1200'.
- Authentication:** A section containing:
  - Type:** A dropdown menu set to 'Password'.
  - User name:** A text field.
  - Password:** A text field (this field is circled in red).
  - CA certificate:** A field showing 'sailor-license-ca.pem' with a file icon to its right.

At the bottom right of the dialog, there is a button labeled 'Advanced...' with a gear icon.

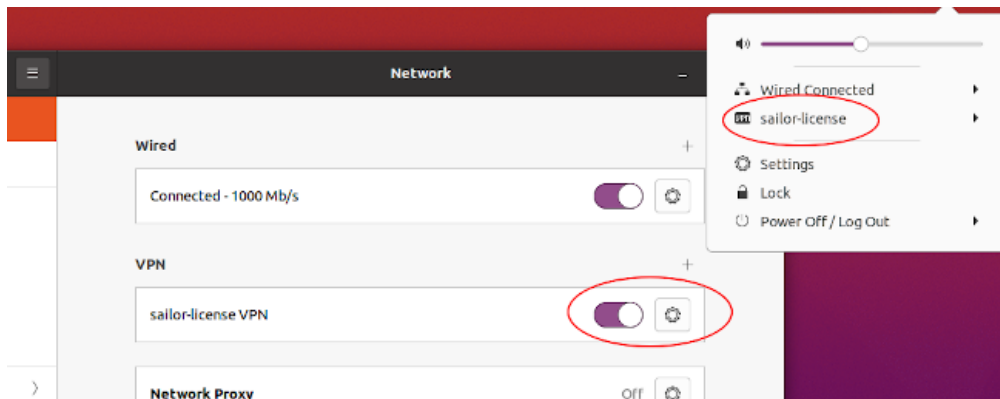
### *Добавление конфигурации VPN*

Осталось только включить переключатель напротив VPN и соединение будет установлено.



*VPN готов*

Кроме того, управлять подключением можно в трее, как показано на рисунке ниже.



*Подключение VPN*

После всех проделанных действий инструмент готов к использованию на вашем компьютере.

## Приложение 2. Командная строка эмулятора Qemu

Пример командной строки для запуска Qemu выглядит следующим образом:

```
./qemu-system-x86_64 -hda debian.qcow2 -m 6G -monitor stdio -netdev user,id=net0 -device e1000,netdev=net0
```

- `qemu-system-x86_64`: исполняемый файл эмулятора
- `-hda debian.qcow2`: подключение образа гостевой операционной системы
- `-m 6G`: выделение оперативной памяти гостевой системе
- `-monitor stdio`: подключение управляющей консоли эмулятора к терминалу
- `-netdev user,id=net0 -device e1000,netdev=net0`: настройка сети и подключение сетевой карты модели e1000

Командная строка выше просто запускает эмулятор с заданным образом диска. Для работы Natch потребуются дополнительные опции командной строки, а именно:

```
-os-version Linux
-plugin <plugin_name>
```

Опция `os-version` настраивает *Natch* для работы с операционной системой Linux, а `plugin` непосредственно загружает плагин.

## Приложение 3. Формат списка исполняемых модулей (`module_config.cfg`)

Пример конфигурационного файла:

```
# module_config.cfg
```

[Image1]

```
path=vmlinux
map=System.map
textstart=0xffffffff81000000
```

#### [Image2]

```
path=exe/dpkg
debuginfo=exe/dpkg.dbg
```

#### [Image3]

```
path=exe/apt-get
```

Конфигурационный файл содержит набор секций с префиксом *Image*.

В каждой такой секции описывается отдельный бинарный файл. В этой секции могут быть объявлены четыре поля: *path*, *map*, *textstart* и *debuginfo*. Обязательным является только *path* (путь к бинарному файлу в хостовой системе). В поле *map* можно указать путь к файлу с символьной информацией, сгенерированной компилятором или дизассемблером IDA. А в *debuginfo* - путь к ELF-файлу с отладочными символами.

При использовании IDA для генерации *map* файлов нужно выставить галочку *Segmentation information*.

### Поле *textstart*

В разделах типа *Image* вместе с *map* может быть задано поле *textstart*. Как правило, нет необходимости определять *textstart* вручную, потому что это может делать скрипт *module\_config.py*. Если же утилита вывела сообщение об ошибке, необходимо проанализировать бинарный файл самостоятельно.

*textstart* используется, если адреса в символьном файле абсолютные, а в исполняемом файле нет. Так как модуль может загружаться в разные места памяти, необходимо вычислить смещение каждой функции от начала секции *.text*. В поле *textstart* как раз и указывается адрес начала секции *.text*. Это поле нужно в редких случаях, например, для ядерных модулей (см. ниже). В остальных случаях можно ничего не указывать.

Чтобы получить информацию о секциях в исполняемом файле, можно использовать утилиту *readelf*:

```
readelf -S <config_name>
```

Пример, когда *textstart* необходимо указывать:

map-файл:

```
fffffffa0000bed t cleanup_module
fffffffa00008c2 t logring_syslog_write_raw
fffffffa0000a4b t init_module
fffffffa000098d t logring_syslog_write
```

вывод утилиты *readelf*:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[ 0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0	0
[ 1]	.text	PROGBITS	0000000000000000	00000040
	00000000000000c8c	0000000000000000	AX	0



Нас интересуют секции типа *PROGBITS*. Если у них указан нулевой адрес, то их не получится сопоставить с *map*-файлом при его загрузке в *Natch*. Поэтому необходимо вручную определить адрес секции *.text*.

Пример, когда *textstart* не нужен:

*map*-файл:

```
00000006:000000000000C000      .init_proc
00000007:000000000000C030      .wget_netrc_db_free
00000007:000000000000C040      .wget_bar_update
00000007:000000000000C050      .seteuid
00000007:000000000000C060      .chdir
00000007:000000000000C070      .fileno
00000007:000000000000C080      .wget_list_free
00000007:000000000000C090      .dup2
00000007:000000000000C0A0      .printf
```

вывод утилиты *readelf*:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[11]	.init	PROGBITS	000000000000C000	0000C000
	0000000000000017	0000000000000000	AX 0 0	4

Адрес секции здесь указан и он соответствует адресам, записанным в *map*-файле, поэтому параметр *textstart* можно не указывать.

## Приложение 4. Формат конфигурационного файла для секции *Tasks (task\_struct\_offsets.ini)*

Конфигурационный файл содержит в себе смещения полей структур ядра Linux, необходимых для работы инструмента.

Смещения полей в структурах данных ядра определяются в ходе отдельного настроечного запуска *Natch*. В первую очередь ищется смещение переменной *current\_task* в сегменте *GS*. Поиск опирается на эвристики, связанные с перехватом системного вызова *getpid*. Затем определяются смещения полей *pid*, *name*, *parent* и всех переменных из раздела *files struct offsets*. Для их вычисления обрабатываются системные вызовы *getpid* и *open*. Для поиска смещения поля *state* перехватывается системный вызов *exit*. Смещения переменных из раздела *memory mapping struct offsets* вычисляются на основе системного вызова *mmap*.

Пример конфигурационного файла:

[Version]

Version=3

[Task struct offsets]

pid=1224

name=1648

parent=1240

state=16

task\_struct=89152

[Files struct offsets]

```
ts_files=1720
fs_file=48
fs_fdt=32
fdt_file=8
f_dentry=24
d_parent=24
d_name=40
d_iname=56
```

#### [Memory mapping struct offsets]

```
ts_mm=1048
ts_mm_active=1056
mm_mmap=0
mm_map_count=104
mm_exe_file=928
vm_start=0
vm_end=8
vm_next=16
vm_prev=24
vm_mm=64
vm_file=160
vma_struct_size=208
```

### Секция Version

- *Version*. Версия формата конфигурационного файла.

### Секция Task struct offsets

- *task\_struct*. Смещение переменной *current\_task* в сегменте *GS*. Для старых ядер должно быть 0, *current\_task* извлекается из стека.
- *pid*. Смещение поля *pid* внутри *task\_struct*.
- *name*. Смещение поля *comm* внутри *task\_struct*.
- *parent*. Смещение поля *real\_parent* внутри *task\_struct*.
- *state*. Смещение поля *\*\_\_state\** внутри *task\_struct*.

### Секция Files struct offsets

- *ts\_files*. Смещение поля *files* внутри *task\_struct*.
- *fs\_file*. Смещение поля *fd* или *fdtab.fd* внутри *files\_struct*.
- *fs\_fdt*. Смещение поля *fdt* внутри *files\_struct* или 0 для старых ядер.
- *fdt\_file*. Смещение поля *fd* внутри *fdtable* или 0 для старых ядер.
- *f\_dentry*. Смещение поля *f\_path.dentry* внутри *file*.
- *d\_parent*. Смещение *d\_parent* внутри *dentry*.
- *d\_name*. Смещение поля *name* или *d\_name.name* внутри *dentry*.
- *d\_iname*. Смещение *d\_iname* внутри *dentry*.

### Секция Memory mapping struct offsets

- *ts\_mm*. Смещение поля *mm* внутри *task\_struct*.
- *ts\_mm\_active*. Смещение поля *active\_mm* внутри *task\_struct*.
- *mm\_mmap*. Смещение поля *mmap* внутри *mm\_struct*.
- *mm\_map\_count*. Смещение поля *map\_count* внутри *mm\_struct*.
- *mm\_exe\_file*. Смещение поля *exe\_file* внутри *mm\_struct*.
- *vm\_start*. Смещение поля *vm\_start* внутри *vm\_area\_struct*.
- *vm\_end*. Смещение поля *vm\_end* внутри *vm\_area\_struct*.
- *vm\_next*. Смещение поля *vm\_next* внутри *vm\_area\_struct*.
- *vm\_prev*. Смещение поля *vm\_prev* внутри *vm\_area\_struct*.
- *vm\_mm*. Смещение поля *vm\_mm* внутри *vm\_area\_struct*.

- *vm\_flags*. Смещение поля *vm\_flags* внутри *vm\_area\_struct*.
- *vm\_file*. Смещение поля *vm\_file* внутри *vm\_area\_struct*.
- *vma\_struct\_size*. Размер структуры *vm\_area\_struct*.

## Приложение 5. Команды монитора Qemu для работы с Natch

Некоторыми плагинами, входящими в состав *Natch*, можно управлять с помощью дополнительных команд. В этом разделе перечислены доступные команды.

- **enable\_tainting** - включить анализ помеченных данных.
- **info replay** - посмотреть текущий шаг записи или воспроизведения. Шаги соответствуют числу выполненных процессорных команд.
- **natch\_get\_attack\_surface [named]** - получить поверхность атаки, параметр *filename* обязательный, параметр *named* опциональный, логического типа, при включении отображает только те сущности, которые имеют имена.
- **show\_tasks** - посмотреть полное дерево задач Linux. Дерево будет отображено на экране и продублировано в указанный файл.
- **show\_modules\_list** - посмотреть полный список загруженных модулей. Список будет отображен на экране и продублирован в указанный файл.
- **taint\_file** - пометить файл (если секция *TaintFile* в конфигурации не активна, необходимо загрузить плагин *taint\_file* командой `load_plugin taint_file`).

## Приложение 6. История релизов Natch

### Natch v.2.0

- Представлен графический интерфейс SNatch v.1.0. Основные возможности:
  - построение графа взаимодействия процессов
    - интерактивный просмотр с помощью привязки к timeline
    - доступно четыре режима отображения сущностей
  - построение стека вызовов
- Улучшено распознавание модулей
- Добавлена поддержка сжатых секций с отладочной информацией
- Добавлена возможность фильтрации сетевых пакетов по протоколу
- Доработан скрипт для конфигурирования Natch
  - рабочая директория для проектов
  - добавлена возможность проброса портов в гостевую систему
- Запущен внешний баг-трекер

### Natch v.1.3.2

- Улучшение и рефакторинг механизма распознавания модулей
- Поддержка набора инструкций SSE4.2 при отслеживании помеченных данных
- Настройка Natch теперь осуществляется с помощью одного скрипта
- Выходные файлы инструмента собираются в одну директорию
- Добавлен журнал событий Natch
- Небольшие изменения в настройке и конфигурационном файле Natch

### Natch v.1.3.1

- Исправлена ошибка сбора покрытия для Ida 7.0
- Исправлена ошибка сохранения лога для помеченных параметров функций
- Исправлена опечатка в генерируемом конфигурационном файле
- Название снапшота вынесено в начало скрипта запуска Natch в режиме воспроизведения

### Natch v.1.3

- Добавлена поддержка отладочной информации

- Добавлена поддержка tar файлов, сгенерированных компилятором gcc
- Расширен набор опций конфигурационного файла Natch:
  - добавлена возможность указывать список файлов для пометки
  - добавлена возможность загрузки дополнительных плагинов
- Добавлен скрипт для генерации конфигурационного файла для модулей
- Обновлен скрипт для генерации командных строк запуска Natch
- Обновлено ядро эмулятора Qemu до версии 6.2

#### **Natch v.1.2.1**

- Исправлена ошибка работы утилиты qemu-img в VirtualBox под Windows 10
- Исправлена ошибка с генерацией имени оверлея в скрипте для генерации командных строк
- Добавлена возможность задавать поля скрипта перед его запуском

#### **Natch v.1.2**

- Скрипт для генерации командных строк
- Выгрузка данных о покрытии кода в IDA Pro
- Построение графа модулей, передающих друг другу помеченные данные
- Ранжирование функций поверхности атаки по числу обращений к помеченным данным
- Исправлены дефекты в механизме распространения пометок
- Мелкие изменения в конфигурационном файле инструмента

#### **Natch v.1.1**

- Возможность настраивать Natch с помощью конфигурационного файла
- Логирование входящих сетевых пакетов
- Отображение операций записи помеченных данных
- Построение графа, описывающего взаимодействие процессов
- Поддержка ELF32
- Исправление списка процессов: уничтожение завершившихся

#### **Natch v.1.0**

- Пометка сетевого трафика (сетевая карта e1000)
- Пометка файлов
- Возможность задания порогового значения для пометок
- Определение модулей с исполняемым кодом в памяти виртуальной машины
- Возможность подгружать tar файлы из IDA
- Получение списка процессов, модулей и функций, участвующих в обработке помеченных данных
- Получение подробной трассы по каждому обращению к помеченным данным, включающей стек вызовов функций, адрес обращения к помеченным данным и количество помеченных байтов