Инструмент определения поверхности атаки «Natch»

Краткое руководство пользователя

1. Описание основных функциональных возможностей инструмента «Natch»

Поверхность атаки (программного обеспечения): совокупность интерфейсов и реализующих их модулей ПО, посредством непосредственного или косвенного использования которых, могут реализовываться угрозы безопасному функционированию ПО.

Инструмент «Natch» предназначен для определения поверхности атаки программного обеспечения (ПО) средствами динамического анализа помеченных данных. В отличие от других программных инструментов «Natch» применяет не двоичные, а целочисленные пометки, что позволяет качественно улучшить результаты анализа. Участки кода, идентифицированные как поверхность атаки, средствами интроспекции соотносятся с программными модулями и функциями. Собранная информация о ПО позволяет оптимизировать меры по разработке безопасного программного обеспечения, реализуемые согласно требованиям ГОСТ Р 56939-2016 и иных национальных стандартов в области разработки безопасного ПО.

Предполагается использование инструмента компаниями, занимающимися как разработкой безопасного ПО, так и сертификацией ПО в системе сертификации ФСТЭК России. Инструмент «Natch» определяет поверхности атаки в автоматическом режиме, что дает возможность интегрировать его с системами CI/CD, а также минимизирует человеческий фактор в процессе анализа.

Основные функциональные возможности инструмента «Natch».

1. Получение поверхности атаки, представленной набором процессов, модулей и функций, которые обрабатывали помеченные данные по время выполнения тестового сценария.
2. Получение подробной трассы обращений к помеченным данным для более детального анализа.
3. Получение трассы вызовов функций с диапазонами адресов памяти, на которых они выполняют запись и чтение помеченных данных.
4. Построение графа, иллюстрирующего распространение помеченных данных от некоторых источников через процессы и модули по всей системе.
5. Пометка файлов.
6. Логирование сетевых пакетов

7. Пометка сетевых пакетов на заданных шагах сценария.
8. Сбор покрытия бинарного кода.

Минимальные системные требования: Для функционирования инструмента требуется персональный компьютер с архитектурой x86-64, не менее 8 ГБ оперативной памяти, и не менее 4 ГБ свободного места на диске. Поддерживаются следующие 64-разрядные дистрибутивы операционной системы Linux: Ubuntu 18 и Ubuntu 20.

Инструмент разрабатывается на языках программирования: C, C++, Python на базе эмулятора с открытым исходным кодом Qemu (распространяется под свободной лицензией GPL версии 2) и представляет собой инфраструктуру для инструментирования кода и поддержки плагинов в самом эмуляторе, а так же набор плагинов для него.

2. Лицензирование и установка

По вопросам определения стоимости, приобретения и использования, обращайтесь по адресу natch@ispras.ru.

2.1 Системные требования

Минимальные системные требования: Для функционирования инструмента требуется персональный компьютер с архитектурой x86-64, не менее 8 ГБ оперативной памяти, и не менее 4 ГБ свободного места на диске. Поддерживаются следующие 64-разрядные дистрибутивы операционной системы Linux: Ubuntu 18 и Ubuntu 20.

2.2 Установка

Сборка эмулятора Qemu поставляется вместе с дистрибутивом «Natch», предварительная установка и настройка не требуются.

Для установки инструмента в ОС Ubuntu 20 необходимо в терминале с помощью команды cd перейти в каталог, в котором расположен дистрибутив «Natch» и запустить скрипт install.sh. Необходимые файлы будут распакованы, для дальнейшей работы необходимо перейти в каталог natch/Natch/bin и запустить скрипт natch_run.sh.

3. Тестовый сценарий работы с инструментом «Natch»

Для проверки работоспособности и начального ознакомления с инструментом динамического анализа «Natch» разработан тестовый сценарий работы. Для выполнения данного сценария потребуется:

- рабочая станция под управлением ОС Ubuntu 20;
- дистрибутив «Natch»;
- подготовленный разработчиком тестовый набор, включающий в себя образ системы на базе Debian10 с пересобранной разработчиком утилитой wget2, а так же набор файлов с символами и map-файлами для утилиты wget2 и конфигурационный файл для их подключения.

3.1 Первоначальная настройка

Дистрибутив состоит из двух zip архивов — с исполняемыми файлами и с библиотеками. Эти архивы необходимо распаковать, после чего в папке с исполняемыми файлами перейти в каталог bin и попробовать запустить qemu-system-x86_64. Если программа не запускается, то эмулятор следует запускать так: LD_LIBRARY_PATH=<путь к каталогу libs> ./qemu-system-x86_64.

Тестовый набор удобно расположить в директории bin, однако его можно распаковать в любое место.

В поставку инструмента входит скрипт *natch_run.sh* (так же расположен в директории bin), предназначенный для генерации командных строк для запуска *«Natch»*. Запущенный скрипт задаст пользователю ряд вопросов, для тестового сценария достаточно будет ответить только на первый - указать путь к тестовому образу, остальные же опции можно оставить по умолчанию, нажимая клавишу Enter. В результате сформируются три скрипта: *run.sh*, *run_record.sh* и *run_replay.sh* для запуска *«Natch»* в режиме обычного выполнения, записи и воспроизведения работы соответственно.

Помимо командных строк скрипт сформирует заготовку конфигурационного файла «Natch». Если было выбрано имя конфигурационного файла по умолчанию, то следует открыть в текстовом редакторе файл natch_config.cfg и внести в него ряд изменений. В заготовке перечислены и закомментированы все возможные секции и опции, доступные в инструменте. Для тестового сценария необходимо полностью раскомментировать секции Ports, Taint, Modules и Tasks. В секции Modules следует указать правильный путь к конфигурационному файлу wget_api.cfg из тестового набора. Конфигурационный файл из секции Tasks будет создан автоматически при первом запуске инструмента.

Для первого запуска следует использовать скрипт run.sh. Во время работы эмулятора будет произведено сканирование системы и получение необходимых для работы «Natch» данных. На этом этапе следует просто запустить скрипт и подождать пока эмулятор отработает и закроется. Как только это произошло, инструмент готов к использованию.

3.2 Проведение эксперимента и оценка результата

Тестовый сценарий необходимо выполнять с использованием технологии детерминированного воспроизведения. Для этого генерируются скрипты run_record.sh и run_replay.sh.

Алгоритм выполнения тестового сценария следующий:

1. Запустить скрипт run_reconrd.sh, что инициирует запись журнала.
2. Дождаться загрузки операционной системы, логин-пароль nat/123. Затем следует перейти в режим супер пользователя командой su, пароль 123. Интересующая нас утилита wget2 находится по адресу /home/nat/wget2/build/bin. Можно перейти в этот каталог с помощью команды cd, либо запустить, использовав путь. Пример запуска: ./wget2 www.google.com. Непосредственно перед запуском утилиты можно перейти в консоль эмулятора и сохранить его состояние командой savevm <name>. Сохраненное состояние позволит воспроизвести работу эмулятора с этого момента и не придется ждать загрузку системы. После сохранения состояния возвращаемся в образ и выполняем wget2. Как только он работал, эмулятор можно закрывать.
3. Если во время записи было сохранено состояние, следует откорректировать скрипт run_replay.sh, а именно в строчке -icount shift=1,rr=replay,rrfile=record.bin,rrsnapshot=init заменить имя init опции rrsnapshot на то, которое было задано в команде savevm. После чего можно запустить скрипт run_replay.sh. Если состояние не сохранялось, скрипт запускается без изменений. Во время работы эмулятора ничего делать не нужно, необходимо дождаться пока он отработает и закроется.
4. В качестве результата будут сформированы два файла: surface_modules.txt и surface_functions.txt, которые можно найти в каталоге с исполняемым файлом инструмента. В первом будут представлены обнаруженные модули, во втором функции с разбивкой по модулям.

В результате в полученном файле surface_modules.txt должны присутствовать такие модули как wget2, libwget.so.1, libz.so.1, libthread.so.0, libc.so.6, libpsl.so.5.

В файле surface_functions.txt будет много неименованных функций по причине того, что не все модули были предподготовлены, но в таких модулях как wget2 и libwget.so.1 должны отображаться именованные функции.

4. Основы работы с инструментом «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:

1. Запуск Qemu с плагином *natch*.
2. Ожидание загрузки до интересующего момента.
3. Включение анализа помеченных данных (если он не был включен автоматически через конфигурационный файл).
4. Выполнение тестового сценария.
5. Запрос результата либо завершение работы эмулятора.

В разделе 2 описаны конфигурационные файлы *«Natch»*, в разделе 3 способы запуска инструмента, раздел 4 содержит информацию о функциональных возможностях инструмента, раздел 5 представляет собой справочник по командам монитора Qemu, доступных при работе с *«Natch»*.

4.1 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».

4.1.1 Основной конфигурационный файл

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

```
# Natch settings

[Ports]
in=22;80;3500;5432
out=22;80;3500;5432

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

# section for loading map files
[Modules]
config=conf_api.cfg

# section for loading task_struct offsets
[Tasks]
config=conf_task.cfg

# section for loading custom syscall config
[Syscalls]
config=custom_x86_64.cfg

# section for network log. only for replay
[NetLog]
log=netpackets.log
tlog=tnetpackets.log

# section for specify replay steps to perform net tainting
[Icount]
steps=4256525458;6874511412
```

Секция Ports

- Поля *in* и *out*: через точку с запятой могут быть перечислены входные и выходные порты входящего трафика. TCP-пакеты, соответствующие этим портам, будут помечаться автоматически.

Секция Taint

- Поле *threshold*: пороговое значение для отслеживания помеченных данных, задается десятеричным числом в диапазоне от 0 до 255.
- 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.

Секция Modules

- Поле *config*: указывается имя конфигурационного файла для распознавания модулей.

Секция Tasks

- Поле *config*: указывается имя конфигурационного файла для распознавания процессов.

Секция Syscalls

- Поле *config*: указывается имя конфигурационного файла для перехвата системных вызовов.

Следующие секции актуальны только для режима воспроизведения работы виртуальной машины.

Секция NetLog

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

Секция Icount

- Поле *steps*: через точку с запятой могут быть перечислены шаги записанного сценария, на которых (и только на них) будет осуществляться пометка данных, если выполняются другие условия установки пометки.

Из вышеописанных секций обязательной является только *Tasks*, остальные при ненадобности могут быть закомментированы или опущены.

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

4.1.2 API

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 двух случаях: - Для правильного определения имен бинарных файлов и их экспортируемых функций - Для подгрузки map-файла с именами функций

При отсутствии map-файла экспортируемые символы все равно будут прочитаны из бинарного файла. Пример конфигурационного файла:

```
# conf_api.cfg

[Image1]
path=vmlinux
map=System.map
textstart=0xffffffff81000000

[Image2]
path=exe/dpkg

[Image3]
path=exe/apt-get

[Dir1]
path=Release
```

Конфигурационный файл поддерживает два типа секций: с префиксами *Image* и *Dir*.

В секции с префиксом *Image* описывается каждый отдельный бинарный файл. В этой секции могут быть объявлены три поля: *path*, *map* и *textstart*. 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 (здесь и далее использование коммерческой версии инструмента не является обязательным).

Секция с префиксом *Dir* позволяет указать путь к каталогу, из которого будут загружены все найденные исполняемые файлы.

Вместе с *map* может быть задано поле *textstart*, которое нужно, если адреса в символьном файле абсолютные. 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.

Пример, когда *textstart* необходимо указывать:

map-файл:

```
ffffffffa0000bed t cleanup_module
ffffffffa00008c2 t logring_syslog_write_raw
ffffffffa0000a4b t init_module
ffffffffa000098d 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     0
```

```
[ 1] .text             PROGBITS         0000000000000000  00000040
     0000000000000c8c  0000000000000000  AX       0     0     4
```

Здесь адреса из map-файла невозможно автоматически сопоставить с исполняемым файлом, 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:

```
[11] .init             PROGBITS         000000000000c000  0000c000
    0000000000000017  0000000000000000  AX       0     0     4
```

Здесь используются относительные адреса, которые автоматически будут привязаны к исполняемому файлу.

4.1.3 Tasks

Конфигурационный файл этой секции генерируется автоматически. От пользователя требуется только указать имя в поле *config*. Если такой файл уже был сгенерирован, то произойдет штатная загрузка *«Natch»*, в противном случае запустится автоматическая настройка инструмента, в результате которой будет сформирован конфигурационный файл. Он содержит в себе смещения полей структуры task_struct из ядра Linux, необходимых для работы инструмента.

4.1.4 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.

4.2 Запуск «Natch»

4.2.1 Использование скрипта для генерации командных строк запуска

В поставку инструмента входит скрипт *natch_run.sh*, который нужен для генерации командных строк для запуска *«Natch»*. В процессе выполнения скрипта пользователь должен уточнять параметры запуска инструмента. На выходе сформируются три

скрипта: *run.sh*, *run_record.sh* и *run_replay.sh* для запуска *«Natch»* в режиме обычного выполнения, записи и воспроизведения работы соответственно.

Инструмент *«Natch»* агрегирован в плагине *natch*, который подключается к Qemu. Прежде чем перейти к описанию скрипта, рассмотрим опции плагина *natch*.

Опции плагина *natch*:

- *config*

  Задает файл конфигурации инструмента (1.1).

- *task_graph*

  Если указана эта опция, при завершении работы эмулятора будет создан граф задач и потоков помеченных данных.

- *module_graph*

  Если указана эта опция, при завершении работы эмулятора будет создан граф модулей и потоков помеченных данных.

- *net_log*

  Актуально для режима записи работы виртуальной машины. Если указана эта опция, происходит логирование сетевых пакетов.

Пример командной строки, отвечающей за подключение плагина с опциями *config* и *task_graph*:

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

Опции могут быть подключены в любых сочетаниях.

Список вопросов, которые будут заданы при выполнении скрипта:

- Название образа. Необходимо указать полный путь до используемого образа системы.
- Ядро. При необходимости можно указать полный путь до ядра (опция Qemu - *kernel*). Если ядро ОС загружается из образа системы, следует просто нажать *Enter*.
- Количество оперативной памяти. Необходимо указать объем оперативной памяти, выделяемый виртуальной машине. Указывается только число в гигабайтах или мегабайтах (например, 6 или 512). По умолчанию значение будет установлено в 4 гигабайта.
- Включение *«Natch».* Если вы хотите включить в командную строку инструмент *«Natch»*, можно нажать *Enter* или ввести *y*, если не нужно - *n*. Следующие пункты актуальны в случае, если был выбран вариант с включением *«Natch»*.
- Имя конфигурационного файла *«Natch».* В этом поле можно указать желаемое имя конфигурационного файла или пропустить вопрос, нажав *Enter*. В этом

случае имя конфигурационного файла будет задано по умолчанию, а именно *natch_config.cfg*.

- Опция *task_graph.* Если вы хотите включить эту опцию, можно нажать *Enter* или ввести *y*, если нет, то *n*.
- Опция *module_graph.* Если вы хотите включить эту опцию, можно нажать *Enter* или ввести *y*, если нет, то *n*.
- Опция *net_log.* Опция работает только в режиме записи. Если вы хотите включить эту опцию, можно нажать *Enter* или ввести *y*, если нет, то *n*.

После этих вопросов скрипт попробует обратиться к утилите *qemu-img*, входящей в поставку эмулятора Qemu, чтобы создать оверлей для образа, необходимый для хранения состояний системы. Если все пройдет хорошо, то скрипт предложит при необходимости добавить к командной строке дополнительные опции эмулятора, если необходимости нет, пункт следует пропустить, нажав *Enter*. Если оверлей создать не удалось, выполнение скрипта будет прервано.

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 в виде комментариев описаны все возможные секции и поля с примерами заполнения. Этот файл необходимо отредактировать - раскомментирвать нужные секции и внести актуальные значения параметров и пути к файлам.

4.2.2 Подготовка запуска «Natch» вручную

Скрипт *natch_run.sh* призван упростить пользователю работу с инструментом, однако можно формировать командные строки для запуска самостоятельно. Пример конфигурационного файла *«Natch»* представлен в пункте 4.1.1 (Основной конфигурационный файл).

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.ini
```

Если конфигурационный файл называется *natch_config.cfg*, плагин *natch* можно запускать без параметра *config*.

4.2.3 Первый запуск «Natch»

Для работы инструмента необходимы дополнительные данные, которые относятся к конкретному запускаемому образу виртуальной машины. Поэтому при первом

использовании образа гостевой операционной системы необходимо произвести настроечный запуск. Для пользователя это выльется в небольшое время ожидания и необходимость перезапустить инструмент по окончании настройки.

Для настройки нужно запустить скрипт *run.sh* и дождаться завершения работы эмулятора. Либо использовать командную строку из пункта 2.2.

В результате этого запуска будет получен конфигурационный файл (1.3), требуемый для работы одного из плагинов, входящих в состав *«Natch»*. Имя этого файла указывается в конфигурационном файле *«Natch»* в секции *Tasks*. Если такого файла не существует, он будет создан.

4.2.4 Штатный запуск «Natch»

После настроечного запуска можно использовать инструмент. Для этого так же можно воспользоваться скриптом *run.sh* для обычной работы эмулятора, либо записать намеченный сценарий работы виртуальной машины с помощью скрипта *run_record.sh* и воспроизводить его в дальнейшем с помощью скрипта *run_replay.sh*.

Важно! Если один и тот же конфигурационный файл будет использован для разных образов гостевых ОС, не стоит забывать обновлять имя файла конфигурации секции *Tasks*. Настройка запустится автоматически только в том случае, если файла нет, а при его наличии будет работать с возможно неподходящими для исследуемого образа смещениями. Совет: используйте отдельные файлы конфигурации для каждого используемого образа.

4.2.5 Запуск «Natch» с использованием детерминированного воспроизведения

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

При использовании скрипта для генерации командных строк, для записи и воспроизведения строки будут получены автоматически: *run_record.sh* для записи и *run_replay.sh* для воспроизведения.

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

Ниже приведен пример строк запуска эмулятора в режимах записи и воспроизведения, соответствующих генерируемым скриптам запуска.

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 \
```

4.3 Функции «Natch»

4.3.1 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* и *<filename>_functions*. 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 wget
    Module 0x7ffc731fe000
    Module /lib/x86_64-linux-gnu/libssl.so.1.1 0x7f5e1cdb8000
    Module /lib/x86_64-linux-gnu/libcrypto.so.1.1 0x7f5e1cae2000
```

```
    Module 0x55ba5ecbb000
    Module /lib/x86_64-linux-gnu/libc.so.6 0x7f5e1c8bf000
    Module /lib/x86_64-linux-gnu/libpthread.so.0 0x7f5e1c89c000
    Module 0x0
```

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 wget
    Module 0x7ffc731fe000
            Function __vdso_clock_gettime 0x7ffc731fea20 9
    Module /lib/x86_64-linux-gnu/libssl.so.1.1 0x7f5e1cdb8000
            Function 0x7f5e1ce13570 85
            Function 0x7f5e1cde4480 23
            Function SSL_CTX_set_record_padding_callback_arg
0x7f5e1cdf30c0 43
            Function 0x7f5e1ce09f00 95
            Function 0x7f5e1ce20cc0 984
            Function 0x7f5e1cde8500 4
            Function 0x7f5e1ce10630 15
```

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

4.3.2 Подробная трасса помеченных данных

Для более детального анализа может потребоваться больше информации, которую можно получить с помощью команды монитора taint_log_enable. Команда инициирует запись лога, в котором на каждое обращение к помеченной памяти формируется расширенный набор данных.

Фрагмент лога для одного обращения:

```
Process name: wget cr3: 0x13a13e000
Tainted access at ffffffffb82243c5
Access address 0xffff8bddf903ecc0 size 8 taint 0xffffffffffffff00
Call stack:
            0: ffffffffb82243c5 in func ffffffffb8a001b8
            1: ffffffffb8a001ac in func ffffffffb8a001ac
            2: ffffffffb8a001a0 in func ffffffffb8a00170
```

```
    3: ffffffffb892f8aa in func ffffffffb8a001b8
    4: ffffffffb892fe73 in func ffffffffb892fe50
    5: ffffffffb82038c7 in func ffffffffb8203870
    6: ffffffffb8203cc0 in func ffffffffb8203c70
    7: ffffffffb8a00a2f
```

Представленная трасса не пишется по умолчанию, поскольку файл получается очень ощутимого размера, при необходимости пользователь включает его сам.

4.3.3 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 команду монитора taint_params_log [filename]. Параметр *filename* является необязательными. По умолчанию результат будет записан в файл *taint_parameters.log*.

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: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: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
```

4.3.4 Получение графов взаимодействий процессов и модулей

На данный момент *«Natch»* позволяет получить укрупненную схему распространения данных от некоторых источников между процессами или модулями. Системные объекты с одинаковыми именами на данной схеме сливаются в один узел. Потоки данных между узлами распознаются по следующему принципу. Если один объект (модуль или процесс) записал помеченные данные, а другой их прочитал, то между ними появляется связь. При построении графа *«Natch»* расходует объем памяти вдвое больший, чем выделенный гостевой системе.

Для использования этой возможности при запуске инструмента необходимо включить опцию *task_graph* (для процессов) или *module_graph* (для модулей). Можно включать обе опции одновременно для получения сразу двух графов, но это будет требовать соответственно больший объем памяти. Для получения результата можно использовать команду монитора natch_get_task_graph [filename] (natch_get_module_graph [filename]) или завершить работу эмулятора. Результат будет записан в файл, указанный пользователем, или в файл *task_graph.txt* (*module_graph.txt*), если опция не указывалась.

Каждая строка выходного файла описывает ориентированное ребро между двумя узлами графа. Узлам соответствуют имена процессов или модулей, а также источники помеченных данных, записанные в квадратных скобках с указанием сетевого порта или имени файла. В конце каждой строки файла есть число: вес ребра графа. Для случаев, когда исходным узлом является источник пометки, вес ребра описывает количество помеченных данных. Вес ребер между процессами (модулями) представляет собой суммарное количество байт помеченных данных, прочитанных узлом назначения с тех мест, куда эти данные были записаны исходным узлом. Вес ребра при этом может быть больше реального объема переданных помеченных данных. Например, процесс A записал 10 байт помеченных данных, процесс B прочитал эти данные 5 раз, в итоге вес ребра от A к B равен 50.

Фрагмент выходного файла:

```
[file:/sbin/init] -> cp 1489208
[port_out:80] -> systemd 714
[port_out:80] -> wget 192
[port_out:80] -> systemd 128
systemd -> wget 208
```

4.3.5 Пометка файлов

До этого рассматривалась пометка только сетевого трафика, но аналогично можно помечать и отдельные файлы в системе и отслеживать модули и функции, которые

были затронуты в результате работы с ними. При этом берутся в расчет только операции чтения. Исполняемые файлы помечаться таким способом не будут.

Для этого необходимо подключить плагин *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.

Плагин подключается аналогично плагину *natch*, а именно: *-plugin taint_file*. Кроме этого, плагин можно включить во время работы эмулятора командой монитора load_plugin taint_file.

После этого необходимо повзаимодействовать с помеченным файлом и оценить результат с помощью команды natch_get_attack_surface <filename>.

4.3.6 Возможности в режиме воспроизведения работы виртуальной машины

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 1.1).

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.

В таком формате можно получить как все сетевые пакеты, возникающие во время сессии (поле *log* конфигурационного файла), так и только помеченные (поле *tlog*).

Пометка сетевых пакетов на заданных шагах сценария

В случае если пользователю необходимо помечать данные в конкретный момент, то можно привязать пометку к шагам сценария выполнения. Для этого в основном файле конфигурации нужно завести секцию *Icount* и в поле *steps* через точку с

запятой перечислить шаги, на которых будет происходить пометка пакетов. Все пакеты, поступающие на других шагах, будут проигнорированы.

4.3.7 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 \
```

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.

Анализ выходного файла

Для удобного анализа выходного файла можно воспользоваться скриптом *coverage.py*, который раскрашивает выполненный код в IDA Pro (Version 7.2) и выводит таблицу, с покрытием функций.

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.

4.3.8 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.
- **load_plugin <plugin_name>** - подключить плагин.
- **natch_get_attack_surface <filename>[named]** - получить поверхность атаки, параметр *filename* обязательный, параметр *named* опциональный, логического типа, при включении отображает только те сущности, которые имеют имена.
- **natch_get_task_graph [filename]** - получить схему взаимодействия процессов, параметр опциональный (файл по умолчанию *task_graph.txt*).

- **show_tasks <filename>** - посмотреть полное дерево задач Linux. Дерево будет отображено на экране и продублировано в указанный файл.
- **show_modules_list <filename>** - посмотреть полный список загруженных модулей. Список будет отображен на экране и продублирован в указанный файл.
- **taint_file <name>** - пометить файл.
- **taint_log_enable [filename]** - включить подробный лог, параметр не обязательный (файл по умолчанию *taint_log.log*).
- **taint_params_log [filename]** - лог вызовов функций с диапазонами адресов прочитанных и записанных помеченных данных, параметр опциональный (файл по умолчанию taint_parameters.txt).

5. Описание процессов, обеспечивающих поддержание жизненного цикла ПО

5.1 Процессы разработки и совершенствования ПО

Разработка инструмента определения поверхности атаки «Natch» ведется по методологии Agile с привлечением современных средств повышения качества кода.

1. Для хранения кода используется система контроля версий git, и изменения в основной ветке проходят инспекцию кода (code review) другими разработчиками.
2. Для проверки работоспособности системы созданы тесты, причем используются как модульные тесты, проверяющие функционал отдельных компонентов, так и интеграционное тестирование, при котором автоматические сценарии проверяют корректность работы системы при типичных вариантах использования.
3. При разработке используется практика непрерывной интеграции (continuous integration): при помощи сервера тестирования Jenkins происходят регулярные автоматические сборки с последующим запуском тестов, упомянутых в предыдущем пункте.

При написании кода разработчики должны придерживаться строгих правил оформления кода; нарушение этих правил не позволит пройти этап инспекции кода.

5.2 Поддержка пользователей ПО

Установка на рабочей станции дистрибутива инструмента определения поверхности атаки «Natch» не требует настройки ОС, установки внешнего ПО или каких-либо других дополнительных действий.

Пользователи могут приступать к работе с инструментом после краткого обучения, проводимого на стороне разработчика (ИСП РАН).

Информация о сбоях в работе инструмента «Natch», проблемах производительности, ошибках целевого функционала передаются пользователями непосредственно ответственным сотрудникам ИСП РАН, без использования публичных Интернет-ресурсов управления ошибками. Это обеспечивает должный уровень конфиденциальности для контрольных примеров (фрагменты исследуемых динамических профилей, снимки памяти), передаваемых пользователем инструмента в ИСП РАН для оценки и исправления программного дефекта.

Со своей стороны ИСП РАН постоянно совершенствует инструмент «Natch», применяя в жизненном цикле разработки передовые методики. Добавление новых и улучшение существующих алгоритмов ведется в инициативном порядке. Обновления инструмента «Natch» передаются пользователям через согласованные с ними каналы распространения обновлений.

5.3 Необходимый персонал для разработки и поддержки

Для разработки и поддержки программного продукта необходима соответствующая квалификация разработчиков. Это вызвано следующими причинами.

1. Специфическая предметная область, требующая глубоких знаний одновременно в нескольких областях: устройство современной аппаратуры и операционных систем, компиляторные технологии, компьютерная безопасность, технологии разработки ПО.
2. Наличие среди алгоритмов, реализованных в среде анализа, алгоритмов, впервые разработанных сотрудниками ИСП РАН.
3. Требования к производительности системы, из-за чего необходимо применять эффективные алгоритмы, в т.ч. хорошо масштабируемые по нескольким вычислительным ядрам.

В силу приведенных причин коллектив разработчиков «Natch» формируется из специалистов, получивших высшее профильное образование.

Для гарантийного обслуживания задействовано 7 научных сотрудников, для Технической поддержки задействованы 2 научных сотрудника, для модернизации программного обеспечения задействованы 5 научных сотрудников.

Адрес электронной почты, по которому можно обратиться по вопросам, связанным с инструментом определения поверхности атаки «Natch» — natch@ispras.ru.