DOI: 10.15514/ISPRAS-2022-34(5)-6

Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин

[1,2] *П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *М.А. Климушенкова, ORCID: 0000-0001-6737-9092 <maria.klimushenkova@ispras.ru>*
[1] *Н.И. Фурсова, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*
[1] *В.М. Степанов, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>*
[1] *И.А. Васильев, ORCID: 0000-0003-3824-2753 <ivan.vasiliev@ispras.ru>*
[1] *А.А. Иванов, ORCID: 0000-0003-2697-1449 <arkadiy.ivanov@ispras.ru>*
[1] *А.В. Иванов, ORCID: 0000-0001-9286-4719 <alexey.ivanov@ispras.ru>*
[1] *М.Г. Бакулин, ORCID: 0000-0002-8569-7382 <bakulinm@ispras.ru>*
[1] *Д.И. Егоров, ORCID: 0000-0001-8037-072X <egorov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Новгородский государственный университет им. Ярослава Мудрого, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41*

Аннотация. Natch – это инструмент для получения поверхности атаки, то есть поиска исполняемых файлов, динамических библиотек, функций, отвечающих за обработку входных данных (файлов, сетевых пакетов) во время выполнения задачи. Функции из поверхности атаки могут быть причиной уязвимостей, поэтому им следует уделять повышенное внимание. В основе инструмента Natch лежат доработанные методы отслеживания помеченных данных и интроспекции виртуальных машин. Natch построен на базе полносистемного эмулятора QEMU, поэтому позволяет анализировать все компоненты системы, включая ядро ОС и драйверы. Собранные данные визуализируются в графическом интерфейсе SNatch, входящем в поставку инструмента. Построение поверхности атаки может быть встроено в CI/CD для интеграционного и системного тестирования. Уточненная поверхность атаки позволит поднять эффективность технологий функционального тестирования и фаззинга в жизненном цикле безопасного ПО.

Ключевые слова: динамический анализ; интроспекция; анализ помеченных данных; qemu; инструментирование; natch

Для цитирования: Довгалюк П.М., Климушенкова М.А., Фурсова Н.И., Степанов В.М., Васильев И.А., Иванов А.А., Иванов А.В., Бакулин М.Г., Егоров Д.И. Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин. Труды ИСП РАН, том 34, вып. 5, 2022 г., стр. 89-110. DOI: 10.15514/ISPRAS-2022-34(5)-6

Natch: using virtual machine introspection and taint analysis for detection attack surface of the software

[1,2] *P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>*
[1] *M.A. Klimushenkova, ORCID: 0000-0001-6737-9092 <maria.klimushenkova@ispras.ru>*
[1] *N.I. Fursova, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>*
[1] *V.M. Stepanov, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>*
[1] *I.A. Vasiliev, ORCID: 0000-0003-3824-2753 <ivan.vasiliev@ispras.ru>*
[1] *A.A. Ivanov, ORCID: 0000-0003-2697-1449 <arkadiy.ivanov@ispras.ru>*
[1] *A.V. Ivanov, ORCID: 0000-0001-9286-4719 <alexey.ivanov@ispras.ru>*
[1] *M.G. Bakulin, ORCID: 0000-0002-8569-7382 <bakulinm@ispras.ru>*
[1] *D.I. Egorov, ORCID: 0000-0001-8037-072X <egorov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Yaroslav-the-Wise Novgorod State University, 41, B. Sankt-Peterburgskaya st., Novgorod, 173003, Russia*

Abstract. Natch is a tool that provides a convenient way of obtaining an attack surface. By attack surface we mean a list of executable files, dynamic libraries and functions that are responsible for input data processing (such as: files, network packets) during task execution. Functions that end up in the attack surface are possible sources of software vulnerabilities, so they should be given an increased attention during an analysis. At the heart of the Natch tool lay improved methods of tainted data tracking and virtual machines introspection. Natch is built on the basis of the full-system QEMU emulator, so it allows you to analyze any system components, including even the OS kernel and system drivers. The collected attack surface data is visualized by SNatch, which is tool for data post-processing and GUI implementation. SNatch comes with Natch tool by default. Attack surface obtaining can be built into CI/CD for integrational and system testing. A refined attack surface will increase the effectiveness of functional testing and fuzzing in the life cycle of secure software.

Keywords: dynamic analysis; introspection; taint analysis; qemu; instrumentation; natch

For citation: Dovgalyuk P.M., Klimushenkova M.A., Fursova N.I., Stepanov V.M., Vasiliev I.A., Ivanov A.A., Ivanov A.V., Bakulin M.G., Egorov D.I Natch: using virtual machine introspection and taint analysis for detection attack surface of the software. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 5, 2022. pp. 89-110 (in Russian). DOI: 10.15514/ISPRAS-2022-34(5)-6

1. Введение

Один из видов поверхности атаки программного обеспечения (ПО) – это те функции, которые получают данные из недоверенных источников. В такие функции злоумышленник может подать некорректные данные и тем самым нарушить работу программы.

Разработчики определяют поверхность атаки своего ПО для тщательного тестирования (в том числе с помощью фаззинга) этих функций. Также поиск поверхности атаки необходим для того, чтобы найти зависимости от лишних или устаревших библиотечных функций. Кроме того, такой анализ может использоваться при сертификации программного обеспечения, чтобы убедиться, что оно достаточно хорошо протестировано.

При сертификации аналитики для нахождения поверхности атаки используют статический анализ (построение зависимостей функций, просмотр кода через IDE) или динамический анализ (поиск покрытия кода набором тестов). Статический анализ не способен в принципе находить некоторые типы зависимостей, к примеру, из-за динамического связывания.

Динамический анализ (при достаточно хорошем наборе тестов) находит именно тот код, который выполнялся. Но сведения о выполненных функциях (и строках кода) недостаточно точно отражают характер этих функций. Зависят ли они от входных данных? Например, код

инициализации выполняется всегда, но злоумышленник редко может на него повлиять. Поэтому этот код не должен быть включён в поверхность атаки.

Для определения более точной поверхности атаки необходимо отследить те программные модули, которые взаимодействовали со входными данными из недоверенных источников.

Мы предлагаем использовать для этого отслеживание помеченных данных в полносистемном эмуляторе [1]. Это позволяет проследить путь данных по всем исполняемым файлам и между процессами. Однако при использовании полносистемного эмулятора для анализа системы нет простого способа определить, что именно там выполняется. Это называется семантическим разрывом - код выполняется, но его высокоуровневая логика неизвестна. Для выделения из выполняющегося кода процессов, модулей, функций и других абстракций существуют методы интроспекции виртуальных машин [2].

В открытом доступе есть два инструмента для полносистемного динамического анализа помеченных данных для архитектуры x86-64: DECAF [3] и Panda [4]. Оба инструмента основаны на эмуляторе QEMU и используют динамическую бинарную трансляцию для отслеживания пометок. Однако ни в одном из этих инструментов нет достаточно развитых средств интроспекции для создания отчётов об искомой поверхности атаки.

В этой статье представлено развитие методов анализа помеченных данных и интроспекции, а также инструмент Natch, который использует их для нахождения поверхности атаки как отдельных приложений, так и целых систем из приложений, контейнеров и динамических библиотек.

2. Общая схема работы Natch

Natch предназначен для извлечения из хода работы системы подробностей о выполняемых приложениях: какие программные модули загружались, какие процессы были запущены, порядок вызова функций и т.п. Чтобы выделить из этого множества только те сущности, которые относятся к поверхности атаки (то есть обрабатывающие небезопасные данные), Natch помечает входные данные из недоверенных источников, а затем средствами эмулятора отслеживает их распространение по памяти виртуальной машины. Так можно узнать, где обрабатывались эти данные и получить поверхность атаки, интересующую аналитика.

Часть Natch, отвечающая за нахождение поверхности атаки, построена на основе эмулятора QEMU [5]. Эмулятор поддерживает множество аппаратных архитектур (в частности, x86, ARM, MIPS, RISC-V) и платформ на их основе.

Из возможностей по анализу системы в QEMU есть только эмулятор gdb-сервера для подключения отладчика и логирование выполняющихся инструкций. Для отслеживания потоков данных и определения того, какие именно приложения выполняются в гостевой системе, этого недостаточно.

Natch представляет собой набор плагинов [6] для Qemu, которые инструментируют выполняемый код и этим замедляют работу эмулятора. В связи с этим, анализ предлагается проводить с использованием детерминированного воспроизведения [7]. Это минимизирует воздействие средств анализа на изучаемую программу. При подключении инструментов анализа к виртуальной машине, воспроизводящей сценарий работы, поведение гостевой системы никак не изменится благодаря тому, что оно уже записано заранее. 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.

Эмулятор QEMU поддерживает детерминированное воспроизведение для всех гостевых платформ, что позволяет использовать его как основу при создании средств полносистемного анализа и отладки.

Примерный алгоритм получения поверхности атаки может выглядеть следующим образом.

1) Запись тестового сценария средствами QEMU.

2) На этапе воспроизведения готового сценария выполняются следующие шаги:
   a) загрузка плагина natch;
   b) ожидание загрузки гостевой системы до интересующего аналитика момента работы системы;
   c) включение анализа помеченных данных (если он не был включен автоматически через конфигурационный файл);
   d) автоматический сбор информации о потоках данных в тестовом сценарии;
   e) сохранение собранных данных и завершение работы эмулятора.

3. Поиск поверхности атаки

При анализе кода, выполняющегося в виртуальной машине, возникает семантический разрыв между представлением программы и кода ОС как последовательности выполняющихся инструкций и тем представлением, которое необходимо для анализа (вызываемые функции, выполняющиеся процессы и т.п.). Для сокращения этого разрыва применяется интроспекция виртуальных машин (virtual machine introspection, VMI) [8, 9].

Современные подходы к интроспекции сопоставляют исходный код ядра ОС с выполняемыми инструкциями, чтобы получить сведения о расположении структур данных в памяти и их содержимом [10, 11] или встраивают свои модули в гостевую систему [4].

Первый подход привязывает алгоритмы анализа к конкретным операционным системам и даже их недокументированным внутренним структурам, если исходные коды ОС недоступны. Внедрение модулей анализа вторым подходом может менять работу системы, требует наличия SDK для сборки гостевого агента, а также не дает возможности использовать детерминированное воспроизведение работы системы.

Наш подход к интроспекции основан на перехвате редко меняющихся событий (системные вызовы) и анализе ядерных структур данных в памяти виртуальной машины.

Интерфейс системных вызовов меняется в рамках существующей системы очень редко [12]. Обычно только добавляются новые варианты функций или старые перестают использоваться.

Но данных, которые системные вызовы получают в виде параметров или возвращают как результат, недостаточно для восстановления подробной картины о работающих процессах и загруженных модулях. К примеру, невозможно связать системный вызов execve (куда передаётся имя программы) и порождённый им процесс.

Более подробные данные о системе извлекаются из ядерных структур данных. Структуры и адреса их расположения в памяти могут различаться в зависимости от версии и сборки ядра. В связи с этим, для применения такого подхода к каждой ОС необходим профиль интроспекции, который содержит смещения и адреса нужных структур и глобальных переменных. Построение такого профиля происходит во время отдельного запуска эмулятора с анализируемой системой.

3. Настройка интроспекции

Для построения профиля интроспекции применяются методы восстановления структур данных ядра. Такие методы можно разбить на 5 категорий: (1) с помощью компилятора [13, 14], (2) с помощью отладчика [15, 16], (3) с помощью гостевой программы [3, 4, 17, 18, 19], (4) с помощью бинарного анализа [20, 21, 22], (5) ручные подходы [23, 24]. Первые две категории предполагают наличие у пользователя исходного кода ядра или отладочных символов в сборке, что может быть доступно не для всех систем. Методы третьей категории требуют внесения изменений в гостевую ОС, что не всегда возможно в рамках выполняемой задачи анализа. Основанные на бинарном анализе методы сопоставляют бинарный код ядра с представлением системы, построенном на базе исходного кода или бинарного кода другой

версии ядра. Ручные подходы опираются на существенные усилия, которые разработчик прикладывает к разработке специальных алгоритмов восстановления для каждой структуры и поля.

Метод генерации профиля интроспекции, применяемый в нашей работе, следует отнести к ручным подходам. Мы применяем эвристические алгоритмы, которые сопоставляют параметры и возвращаемые значения системных вызовов с нужными для интроспекции полями структур данных ядра. Так как наборы системных вызовов в разных ОС похожи, то основные идеи эвристик, разработанных для одной системы, можно повторно использовать для восстановления структур данных ядра других ОС.

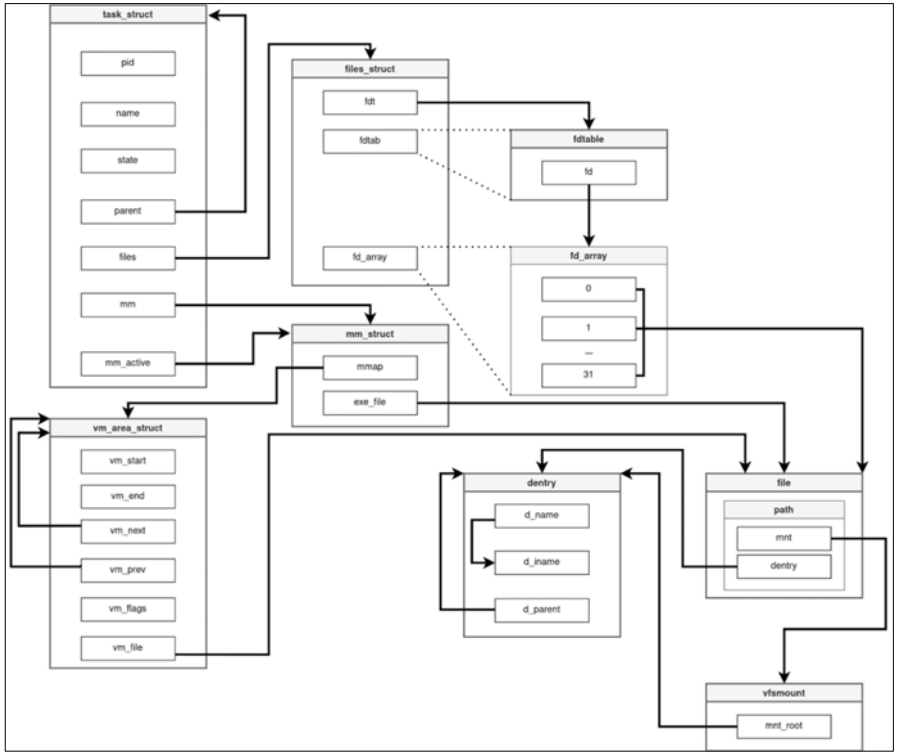

Рис. 1. Ключевые структуры данных ядра Linux
Fig. 1. Key Data Structures of the Linux Kernel

На рис. 1 представлены основные структуры и поля ОС Linux, которые восстанавливаются нашим подходом. Основная структура данных на этой схеме – это task_struct. Она описывает процесс.

На старых версиях Linux адрес task_struct текущего процесса можно извлечь в любой момент из стека ядра. Для платформы x86 адрес лежит на верхушке этого стека. Новые ядра Linux могут не хранить этот адрес в стеке. Вместо этого, адрес task_struct можно найти на определенном смещении от адреса из регистра GS. Определение этого смещения является первой задачей настройки интроспекции, так как в task_struct хранится много полезных атрибутов процесса (имя, загруженные модули, командная строка и т.п.). Чтобы найти task_struct, мы сначала перехватываем системный вызов getpid. В области 128 килобайт памяти от адреса GS происходит разыменование каждого возможного указателя. В каждой

возможной структуре task_struct ищется известное значение PID из системного вызова и строка, похожая на имя процесса (поле comm). Когда искомые значения повторно находятся на одних и тех же смещениях некоторое количество раз, смещения фиксируются в профиле интроспекции и используются в последующих фазах восстановления структур.

Поле parent в task_struct указывает на структуру, описывающую родительский процесс. Для определения смещения этого поля наше решение собирает значения адресов всех встречаемых task_struct. Затем ищет поле с указателем на другую такую структуру и проверяет, что путем разыменования этого поля можно дойти до корневого процесса, поле parent которого указывает на его же task_struct.

Поиск смещения поля state, которое описывает состояние процесса, опирается на тот факт, что в течение системного вызова exit в него записывается значение 40h или 80h (варианты константы TASK_DEAD для разных ядер). Поиск смещений файловых структур выполняется по завершению системного вызова open: нам нужны структуры dentry, поле name которых содержит имя файла, совпадающее с параметром системного вызова. Смещения указателей на пути к этим структурам перебираются в определенных диапазонах значений. На предполагаемых таблицах файловых дескрипторов используется значение файлового дескриптора из системного вызова для перехода к следующей структуре.

Смещения структуры mm_struct ищутся на момент завершения системных вызовов mmap. Поле exe_file содержит адрес структуры file и описывает исполняемый файл процесса. По ранее определенным смещениям файловых структур для каждого предполагаемого указателя на file, наше решение восстанавливает имя исполняемого файла и сопоставляет его с именем процесса. По известным смещениям также определяется адрес структуры file для файлового дескриптора из параметров системного вызова.

Поле mmap в mm_struct указывает на список структур vm_area_struct, которые описывают регионы памяти. Здесь мы применяем набор условий, которые для предполагаемого набора смещений полей должны выполняться одновременно. Поле vm_file первой структуры в списке должно указывать на ту же структуру file, что и exe_file. Поле vm_next должно указывать на следующую такую же структуру в списке. Для некоторых соседних структур в списке поле vm_start одной структуры должно совпадать с vm_end другой. Поле vm_file должно принимать некоторые специфичные значения. И последнее условие: в списке должна быть структура, описывающая новое отображение памяти с параметрами системного вызова mmap.

Также во время вызовов mmap определяются смещения полей arg_start, arg_end, env_start и env_end в структуре mm_struct, описывающих области памяти, где хранятся параметры командной строки и переменные окружения. Алгоритм следующий: выполняем обход списка структур vm_area_struct. В нем по значению поля vm_flags находим структуру, описывающую область памяти стека. Далее, находим указатели на четыре самых больших адреса в структуре mm_struct, которые принадлежат стеку. Это и будут искомые поля, потому что загрузчик в ядре Linux записывает искомые данные в самый конец стековой области памяти.

По завершению настроечного запуска, все найденные смещения структур записываются в конфигурационный файл. Этот файл затем используется для выполнения задач интроспекции.

4. Анализ помеченных данных

Динамический анализ помеченных данных [25] – метод, при котором отслеживается использование тех или иных данных в процессе выполнения программы. Выделяется три набора зависящих от задачи правил, определяющих анализ.

1) Какие данные отслеживать (когда вносить пометку); например, помечать данные из недоверенных источников, или помечать чувствительные данные (пароль)?
2) Как продвигать пометку: как операции над данными влияют на помеченность результата; например, при операциях копирования пометка также копируется, для бинарных операций результат будет помечен если был помечен хотя бы один исходный операнд?
3) Опционально: в каком случае выводить предупреждение или сообщать об ошибке? Например, если изначально помечались данные из недоверенных источников, попадание пометки в счётчик команд может свидетельствовать о попытке перехвата управления.

В открытом доступе есть два инструмента для полносистемного динамического анализа помеченных данных для архитектуры x86-64: DECAF [3] и Panda [4]. Оба инструмента основаны на эмуляторе QEMU и используют динамическую бинарную трансляцию для отслеживания пометок. Первый инструмент производит инструментирование на уровне внутреннего представления TCG, и благодаря различным оптимизациям является одним из самых быстрых известных на данный момент, однако базируется на старой версии эмулятора (1.0). Второй инструмент использует преобразование внутреннего представления TCG в LLVM-биткод, который инструментируется для продвижения пометок, благодаря чему поддерживается большое количество гостевых архитектур, но при этом значительно снижает скорость работы (более чем десятикратное замедление по сравнению с обычной эмуляцией).

В Natch используется метод «разбавленных» пометок [26], при котором чем больше преобразований происходило с помеченными данными, тем меньше «сила» полученной пометки. Это позволяет более точно описывать поверхность атаки, потому что злоумышленников в первую очередь интересуют функции, получающие слабо преобразованные данные. Если же входные данные подверглись, например, шифрованию, то сконструировать эксплоит будет слишком сложно.

Поэтому в отчётах Natch выводятся те функции, которые работали с помеченными данными с уровнем пометки не меньше заданного порога. Порог настраивается в конфигурационном файле и по умолчанию равен 250 (максимальное значение пометки равняется 255).

5. Монитор процессов

Плагин мониторинга процессов записывает историю созданий, переключений и завершений процессов. Для этого он выполняет чтение параметров task_struct из гостевой памяти. Создание нового процесса определяется по появлению нового экземпляра task_struct. Завершение процесса фиксируется в момент записи соответствующего значения в поле state этой структуры. Переключение текущего процесса проверяется, когда обновляется значение адреса каталога таблиц (регистр CR3).

Также плагин выполняет обход выполняемых процессов через поле parent, в ходе которого строит дерево процессов.

С помощью параметров mm_struct извлекается информация о регионах памяти процесса. По полю vm_flags в структуре vm_area_struct определяется принадлежность региона к разделяемой или приватной памяти. Также плагин извлекает адреса памяти со строками запуска процессов. За счет этих строк плагин определяет имена контейнеров docker.

6. Монитор файлов

Плагин мониторинга файлов отслеживает операции чтения и записи с файлами. Он может работать без чтения структур ядра. В таком случае он извлекает имена файлов из системных вызовов open. Эти имена привязываются к номерам файловых дескрипторов и процессам. Как правило, извлекаемые таким образом имена являются относительными. Чтобы получить полные имена файлов, плагин использует структуры ядра из гостевой памяти. По ним для файловых дескрипторов из системных вызовов он восстанавливает полные имена файлов.

Файлы сокетов в Linux не имеют уникальных имен. Их имена соответствуют типу сокетов, к которому они принадлежат. В нашей работе параметры сокетов, такие как порт и ip адрес, извлекаются из параметров системных вызовов bind и accept, затем сопоставляются адресам структур file.

Почему параметры сокетов сопоставляются структуре file, а не номерам файловых дескрипторов? Номера файловых дескрипторов идентифицируют файл в рамках одного процесса. Когда открытые файлы наследуются дочерним процессом после вызова fork, возникает необходимость привязывать сопоставленные параметры к номерам файловых дескрипторов нового процесса. В то же время, адрес структуры file остается одним и тем же в родительском и дочернем процессе, за счет чего применять его в качестве ключа для хранения параметров значительно проще.

7. Монитор модулей

Был разработан плагин, который определяет имена и адреса отображений в памяти для исполняемых модулей. Для этого он использует параметры системных вызовов mmap. Номер файлового дескриптора из аргументов mmap через монитор файлов преобразуется в название модуля.

Но из памяти виртуальной машины как правило невозможно прочитать секции исполняемых файлов с отладочной информацией, потому что операционной системе они не нужны. Поэтому основной набор исполняемых файлов, интересующих аналитика, загружается отдельно. Так Natch может прочитать из них всю отладочную информацию и сопоставить адреса функций с их именами.

Для обнаружения этих файлов в памяти, Natch сравнивает содержимое их кодовых секций с данными из страниц памяти в момент их отображения. Если загружаемая страница не относится ни к одному из этих файлов, плагин извлекает информацию об отображении памяти из структур ядра. Выполняется поиск структуры vm_area_struct, поля vm_start и vm_end которой соответствуют диапазону области памяти искомого модуля. Из поля vm_file данной структуры извлекается полное имя исполняемого файла.

8. Восстановление процессов и потоков

Для получения представления о поведении системы необходимо иметь данные о стеках вызовов, однако восстановление такой информации невозможно без заведомого получения данных о процессах и потоках в системе. Восстановление этих данных также должно следовать следующим принципам: работа в условиях полносистемного анализа, работа алгоритма без встраиваемых программ и программ-агентов, универсальность метода вне зависимости от анализируемой системы, возможность реализации для различных процессорных архитектур, алгоритм должен опираться на низкоуровневую информацию от системы. Таким образом полученный в результате алгоритм будет обеспечивать высокую переносимость, портируемость, простоту сопровождения и надежность результата.

Восстановление процессов является задачей тривиальной, которая решается в Natch схожим образом с существующими аналогами: для организации виртуальной памяти система выделяет для каждого процесса новое значение записи в таблице трансляции виртуальных адресов в физические. Процессор выделяет отдельный регистр для хранения этого значения (для x86 это CR3, для ARM – CP15 и т.п.). Таким образом, наблюдая за значением этого регистра, можно однозначно отличать друг от друга запущенные процессы.

Однако для восстановления информации о потоках обычного наблюдения за состоянием регистров недостаточно, требуется вводить дополнительную логику.

Системы, реализующие анализ уровня процессов, такие как Pin [27], Valgrind [28], DynamoRIO [29], как правило, представляют из себя виртуальные машины, внутри которых

запускается рассматриваемое приложение. Им свойственен прямой доступ к любой интересующей информации о рассматриваемом процессе – легкость доступа к данным и их полнота обеспечивается за счет выполнения инструментария анализа в одной операционной системе с исследуемым приложением, а значит и наличии доступа к системному API. Однако в этом случае невозможно анализировать межпроцессное взаимодействие, анализировать ядро ОС. При этом инструменты характеризуются строгой ОС зависимостью и не обеспечивают необходимой в вопросах безопасности изоляции инструмента и предмета анализа.

Среди систем, реализующих полносистемный анализ, можно выделить: Virtuoso [17], TEMU [30], DECAF [3, 31], PANDA [4], PyREBox [32], PEMU [33].

Virtuoso [17] применяет внедряемую программу-агент для обращения к интересующим данным. Трасса данного обращения анализируется и таким образом обнаруживаются адреса для получения данных о потоках из системных структур.

TEMU [30] реализует динамическое бинарное полносистемное инструментирование на базе эмулятора QEMU. Извлечение данных уровня операционной системы происходит с помощью специфического для каждой ОС семантического извлекателя. Причем для его функционирования на ОС семейства Windows используется загружаемый в систему модуль ядра

Создатели DECAF [3, 31] при разработке опирались на наработки TEMU и избавились от модификации ядра исследуемой системы, однако вся информация о процессах, потоках, модулях основывается на прямом обращении к структурам ядра исследуемой системы.

Процесс анализа в фреймворках PANDA [4] и PyREBox [32] базируется на ОС-специфичном конфигурационном файле, который в первом случае генерируется программой-агентом, а во втором заранее создается и распространяется вместе с фреймворком. PANDA также реализует и условно ОС-независимые способы восстановления данных, однако они дают результаты низкой точности: так, например, определение потоков по ASID дает ложные результаты, если одному процессу соответствует несколько потоков, а предлагаемый эвристический метод основан на определении смены потока по значительной смене указателя стека, что не всегда соответствует действительности.

Восстановление данных о потоках в PEMU [33] привязано к специфике процессора x86 и также имеет невысокую точность: поток идентифицируется по значению регистра esp с наложенной обнуляющей маской на нижние 12 бит.

Предлагаемый нами подход для восстановления данных о потоках строится на общих для различных ОС принципах, реализуется в условиях полносистемного динамического анализа, и позволяет получать результат без встраивания программ-агентов.

Суть предлагаемого метода заключается в следующем: каждому процессу соответствует один или более поток, при этом каждый поток однозначно принадлежит процессу. Идентификация процесса задача тривиальная и описана выше. Таким образом задача заключается в том, чтобы разделить в рамках процесса потоки друг от друга. Способ разделения строится на следующем наблюдении: каждый из потоков использует локальные переменные и оперирует адресами возврата, а значит должен поддерживать отдельный стек. Отсюда следует вывод, что для однозначной идентификации потока необходимо и достаточно рассматривать пару “идентификатор процесса” и “диапазон значений стека” (рисунок). Сумев выделить из общей памяти стека локальные стеки для потоков, мы сможем однозначно определить и сам поток (рис. 2).

Задача определения диапазона значений стека сводится к следующему: как понять, относится ли новое значение SP к тому же потоку, а значит свидетельствует о расширении диапазона значений стека, или сигнализирует о создании нового. Предлагаемый метод основывается на предположении, что в системе должен присутствовать определенный набор событий,

предшествующих операции создания нового стека и связанного с ним потока. Другие же события, связанные с изменением стека, служат для расширения границ диапазонов уже обнаруженных стеков.

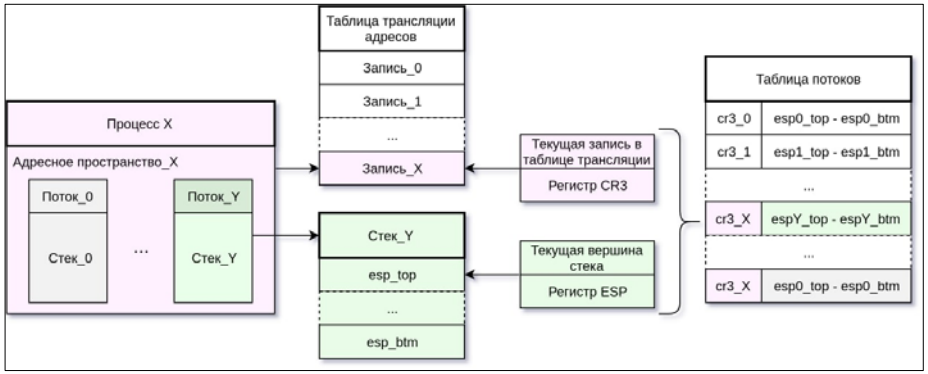

Рис. 2. Стеки потоков выполнения и адресные пространства процессов
Fig. 2. Control Flow Stacks and Process Address Spaces

В ходе разработки метода были выделены следующий события для создания новых и расширения существующих диапазонов: создание нового диапазона – возникновение нового значения SP после выполнения инструкции прямого присваивания SP нового значения, например: pop esp, mov esp; и возвращение из режима ядра в пользовательский режим. Прочие операции явно или косвенно модифицирующие стек расширяют диапазон. Данный набор событий будет отличаться для различных процессорных архитектур, однако принцип выделения подмножеств остаётся схожим.

9. Стеки вызовов

Имея данные о потоках, можно приступить к восстановлению стеков вызовов. При реализации данного алгоритма важно не опираться на соглашение о вызовах, т.к. информация об указателе фрейма не всегда описана в соглашении, а значит и восстановить стек вызова с его помощью в общем случае не удастся.

Логика работы алгоритма восстановления стека вызовов заключается в следующем: из выполняемых в гостевой машине инструкций выделяются те, что относятся к вызову функции (call), и те, что соответствуют операции возврата (ret). Для каждого из потоков сохраняется свой список адресов, по которым произошел вызов, и их расположение в стеке, а при возникновении возврата на требуемой глубине стека соответствующая запись из списка вызовов удаляется.

Однако для корректной реализации алгоритма важно было учитывать нестандартные способы системного взаимодействия со стеком, такие как: вызов процедуры с помощью безусловного или условного перехода; вызов процедуры через табличное значение; осуществление возврата через прямую модификацию стека и использование перехода; использование инструкции ret в качестве безусловного перехода; возврат через несколько записей в стеке вызовов; и ряд других. Только учитывая все особенности, нам удалось получить надежный алгоритм восстановления стеков вызовов. Данные, полученные от разработанного алгоритма, используются для последующего восстановления стека вызовов при обращении к помеченным данным, а также для построения графа вызовов, графа процессов, флейм-графа и пр., таким образом являются одной из базовых частей большой части реализуемых в Natch алгоритмов.

10. Покрытие кода

На данный момент существует множество инструментов, предназначенных для анализа покрытия. Они сильно различаются по целям и возможностям.

Компиляторы gcc и clang поддерживают сбор покрытия на уровне исходного кода. Сюда относится gcov [34] и его различные вариации. Главной проблемой такого рода инструментов является необходимость в исходном коде, который может быть недоступен. К преимуществам данного метода можно отнести поддержку большого числа архитектур и наличие инструментов для анализа результирующего покрытия в удобном формате, например, в HTML. (Сам формат gcov является достаточно неудобным для последующей обработки, о чём говорят и разработчики Lighthouse [35]). Кроме того, существуют решения, на основе данного подхода, позволяющие собирать покрытие бинарного кода для работающего ядра Linux [36]. Но и тут есть ограничения, инструменты такого рода не могут поддерживать файлы библиотеки динамической компоновки.

Другой подход к анализу покрытия – использование аппаратных механизмов трассировки. К настоящему моменту наиболее продвинутой реализацией такой технологии является Intel Processor Trace, возникшая вместе с пятым поколением процессоров от Intel. Трассировка вычислительного процесса позволяет получить полный поток исполнения некоторого приложения с минимальными издержками. Важным преимуществом технологии Intel PT является возможность отслеживать выполнение кода на максимально низком уровне – вплоть до выполнения отдельных инструкций. В отличие от сбора покрытия кода на уровне исходного кода, данная технология может работать как с исходным кодом, так и с бинарным. В качестве альтернативы инструментированию во время компиляции и случайному назначению идентификаторов базовых блоков, методы, основанные на IPT используют фактические адреса базовых блоков во время выполнения для отслеживания переходов между базовыми блоками.

Если же использовать данный метод для сбора покрытия без изменения, то возникает проблема с затратным декодированием инструкций во время анализа трассировки (более 85% процессорного времени) [37]. Так происходит из-за повторного анализа одних и тех же инструкций для одного и того же двоичного файла. Для решения данной проблемы уже существуют методы, которые с помощью кэширования инструкций увеличивают производительность вплодь до 40 раз.

Другой проблемой является неэффективное хранение данных для их последующей обработки: IPT может за несколько секунд сбрасывать гигабайты сжатых данных. Для избежания данной проблемы можно воспользоваться эвристическим обобщением данных на лету.

Наконец, главный недостаток – поддержка архитектур. Как уже говорилось выше, IPT - является особенностью процессоров Intel, начиная с пятого поколения, следовательно, сбор покрытия бинарного кода будет работать только для x86 архитектуры.

Технология динамического бинарного инструментирования (DBI) заключается во вставке в бинарный код анализирующих процедур, отвечающих за проведение необходимого анализа, модификацию и мониторинг исследуемой программы. Данные процедуры вызываются каждый раз, при достижении определенного участка кода или возникновения в программе определенного события (создание процесса, возникновения исключения и т.д.).

Данный подход не требует наличия исходного кода анализируемого приложения – работа происходит непосредственно с бинарными файлами. Число поддерживаемых архитектур в нём будет зависеть от реализованных в виртуальной машине архитектур.

Главная проблема данного подхода вытекает из-за его базирования на виртуальной машине - время работы. Помимо того, что выполнение кода в виртуальной машине сам по себе не быстрый процесс, так ещё и дополнительные временные расходы добавляются ко всей

программе, даже если пользователя интересует только отдельная ее часть, например, разделяемая библиотека.

Несмотря на недостатки, этот подход хорошо вписывается в инфраструктуру Natch. К тому же, его архитектуронезависимость позволяет в дальнейшем расширять его применение.

10.1 Сбор информации о покрытии

Рассмотрим три основных способа сбора информации о покрытии кода.

1) На основе выполненных инструкций. Данный способ позволяет инструментировать одну инструкцию за раз и является наиболее точным.
2) Сбор информации о покрытие кода на основе выполненных базовых блоков. Базовый блок (BB) – это последовательность инструкций с одним входом и одним выходом. Вместо одного анализируемого вызова для каждой инструкции, часто эффективнее вставить один анализируемый вызов для BB, тем самым сократив количество вызовов.
3) Сбор информации о покрытие кода на основе трассировки. Под трассировкой понимают последовательность базовых блоков, которая завершается безусловной инструкцией, изменяющей поток управления (например, jmp/call/ret). Имеет один вход и несколько выходов. Может улучшить производительность, по сравнению с BB и INS.

Мы остановились на втором варианте, так как он даст меньшее замедление, чем сбор покрытия на основе выполненных инструкций и избавит от необходимости реализовывать сложные абстракции для трассировки в QEMU.

10.2 Реализация сбора покрытия

Каждый раз, когда блок кода выполняется в эмуляторе, кроме самого диапазона адресов этого блока, в покрытие кода добавляется информация о модуле и процессе, к которым блок трансляции принадлежит. Происходит это благодаря реализованным плагинам [6], которые способны находить загруженные модули и процессы для них.

Помимо подписки на начало выполнения блока трансляции, необходимо также подписаться и на возникающие исключения, разделяющие блок трансляции на две части: до и после. Для этого добавим новый сигнал после вычисления PC инструкции, вызвавшей исключение, и подпишемся на него в инструменте. После получения такого сигнала, необходимо уменьшить размер блока трансляции, до требуемого. Откинутая же часть станет позже новым блоком.

Основное различие между сбором покрытия бинарного кода для ядра и для пользовательского режима заключается в размерности адреса базовых блоков. Самым очевидным решением является использовать большую размерность адреса для всех базовых блоков, но это сильно увеличит объём файла с покрытием. Поэтому было решено определять, какой код выполняется в режиме ядра и указывать для него смещение не от начала модуля, а от начала секции .text в нём.

Сбор покрытия бинарного кода для помеченных данных будет аналогичен сбору покрытия бинарного кода для всех выполненных базовых блоков, за исключением поиска тех самых помеченных данных. При чтении помеченных данных, последний полученный блок трансляции будет считаться работающим с помеченными данными, следовательно, будет добавлен к результирующему покрытию.

10.3 Анализ полученного покрытия бинарного кода

В данный момент используется метод, основанный на получении выполненных базовых блоков и объединения их с функциями в IDA Pro [38], с помощью плагина. В процессе работы с плагином пользователь может выбрать интересующие его процессы и модуль (если он не определился автоматически) и просмотреть выполненные и помеченные базовые блоки на

графе потока управления. В дальнейшем планируется интеграция этого плагина с инструментом Lighthouse [35].

10.4 Сравнение с существующими решениями

В сжатой форме результаты сравнения приведены в табл. 1.

Табл. 1. Сравнение разных инструментов для сбора покрытия кода.

Решение	Независимость от компилятора	Инструментирование на уровне ядра	Инструментирование за пределами виртуальной машины	Поддержка архитектур	Пометка данных	Информация о процессах и модулях	Удобный формат
kcov	X	✓	X	✓	X	X	✓/X
Llvm-cov	X	X	X	✓	X	X	✓/X
IPT	✓	✓	X	X	X	✓	X
DynamoRIO	✓	X	X	✓	X	✓/X	✓
Valgrind	✓	X	X	X	X	✓	✓
PIN	✓	X	X	✓	X	✓	✓
PEMU	✓	✓	✓	✓	X	✓	✓
Panda	✓	✓	✓	✓	✓	✓	X
Natch	✓	✓	✓	✓	✓	✓	✓

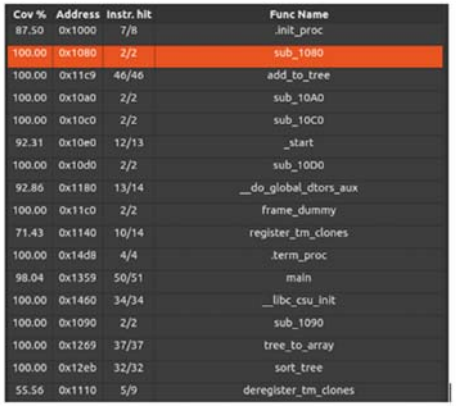

Рис. 3. Окно выбора функции из плагина отображения покрытия к IDA Pro
Fig. 3. Function selection window from the coverage display plug-in for IDA Pro

10.5 Примеры работы

На рис. 3-4 приведены окно выбора функции из плагина к IDA Pro и граф базовых блоков с подсвеченными блоками из того же плагина.

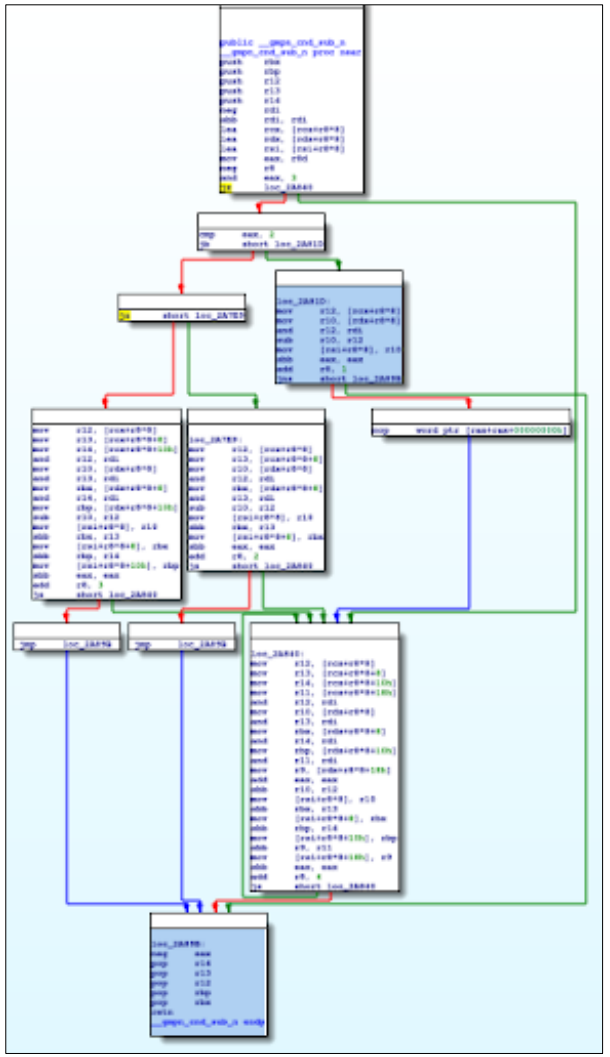

Рис. 4. Граф базовых блоков с подсвеченными блоками из плагина отображения покрытия к IDA Pro
Fig. 4. Graph of basic blocks with highlighted blocks from the coverage display plugin for IDA Pro

11. Графы взаимодействия процессов

Для анализа потоков данных между процессами строится граф их взаимодействия. Он иллюстрирует распространение помеченных данных по всей системе, показывая между какими процессами эти данные передавались. В графе комбинируется низкоуровневая

информация о распространении меток taint-анализа с высокоуровневой информацией о гостевой системе, получаемой в ходе интроспекции.

Близким аналогом предлагаемого решения является инструмент Panorama [39]. Это реализованная на базе эмулятора QEMU система обнаружения и анализа вредоносных программ в ОС Windows. Каждому помеченному байту Panorama сопоставляет небольшую структуру данных, хранящую источник данных и другую информацию, на базе которой строится граф. Взаимодействия отслеживаются между процессами, в которых произошла запись или пометка данных, и процессами, прочитавшими эти данные. Кроме процессов граф отображает модули анализируемой программы, файлы, сетевые соединения и другие источники ввода. Panorama распознает не только взаимодействия через ОЗУ, но и потоки данных через запись и чтение диска.

Инструменты расследования вторжений BackTracker [40], Taser [41], Protracer [42] и Rain [43] отслеживают информационные потоки между процессами в Linux за счет встраиваемых в ядро модулей. В отличие от Panorama, которая применяет полноценный динамический анализ байтовой гранулярности, вышеперечисленные инструменты опираются только на отслеживание системных вызовов, за счет чего имеют более грубый характер распространения меток и определения зависимостей.

В нашем решении для отслеживания передаваемых данных на эмуляторе выделяется дополнительный объем теневой памяти в размере двух байт на каждый байт ОЗУ гостевой системы. В эти два байта записывается идентификатор узла графа. Идентификаторам сопоставляются процессы, в течение работы которых записываются помеченные данные. Когда происходит чтение помеченных данных, проверяются идентификаторы читаемой памяти. Если они не соответствуют текущему процессу, то фиксируется передача данных от другого процесса.

Также в граф записывается информация о файлах и сокетах, в которые попадают помеченные данные. Взаимодействия с ними отслеживаются по системным вызовам. Имена файлов и параметры сокетов восстанавливаются соответствующими механизмами интроспекции.

Результат анализа записывается в json файл по завершению работы эмулятора. Этот файл затем используется инструментом SNatch для визуализации.

12. SNatch – интерфейс для визуализации результатов анализа

SNatch – это инструмент, предназначенный для постобработки, анализа и отображения данных, полученных от инструмента Natch. Состоит из двух частей: бэкенда и фронтенда. Первый выполняет разбор и обработку бинарных логов, сгенерированных Natch, и последующую генерацию сущностей для пользовательского анализа, на основании полученных данных. Фронтенд предоставляет графический пользовательский интерфейс, доступный через браузер, который позволяет изучать и взаимодействовать с полученными от бэкэнда данными для анализа.

Анализ организуется с помощью “проектов”, которые создаются пользователем через браузерный интерфейс. При создании нового проекта в бэкенд передаются сгенерированные Natch: лог помеченных данных, лог процессов, лог модулей, лог стеков вызовов, символьный лог и лог задач. Формируемые при разборе и анализе данные однозначно соотносятся с пользовательским проектом и используется для дальнейшей визуализации в фронтенд части.

В фронтенд части доступны следующие интерактивные модули отображения информации: граф вызовов, граф взаимодействия процессов, граф времени жизни процессов и дерево процессов.

Граф вызовов (рис. 5) традиционно представлен в виде древовидной структуры и содержит информацию о функциях, взаимодействовавших с помеченными данными. Интерактивные элементы позволяет сворачивать и разворачивать как отдельные ветви, так и дерево целиком.

Деревья разбиваются по процессам, а для каждой записи отображается смещение вызванной функции в модуле и название модуля, функции и номер строки в исходнике, если соответствующие данные были получены при создании проекта.

Рис. 5. Граф вызовов функций в SNatch
Fig. 5. Graph of function calls in SNatch

Граф взаимодействия процессов (рис. 6) представлен в виде ориентированного графа, где в вершинах расположены взаимодействующие элементы, а стрелки отображают направление и объем (толщина стрелки) передаваемых данных. На графе могут присутствовать вершины следующих типов: бинарный файл, интерпретатор и связанный с ним скрипт, сокет, внешняя сеть, терминал и текстовый файл. Благодаря шкале времени, можно поэтапно проследить процесс распространения помеченных данных внутри системы.

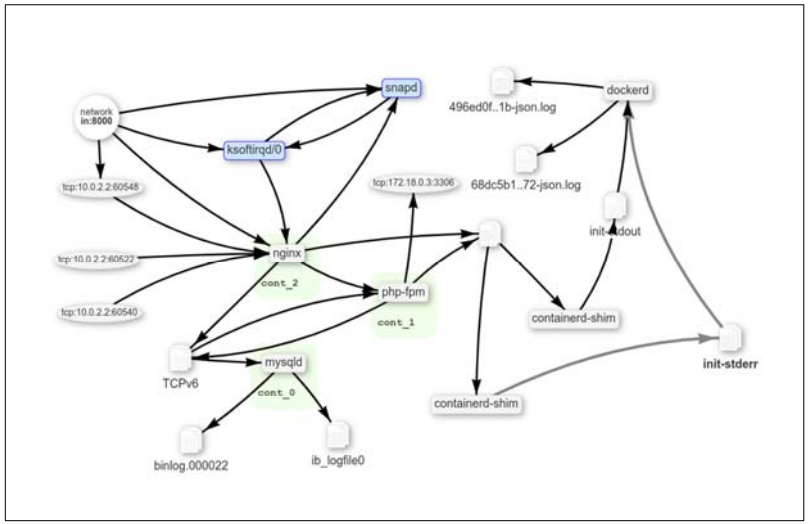

Рис. 6. Граф взаимодействия процессов в интерфейсе SNatch
Fig. 6. Process interaction graph in the SNatch interface

При этом отображение поддерживает несколько режимов в зависимости от привязки к моменту выполнения конкретного взаимодействия: active – активное взаимодействие на

выбранный момент времени; past – активное взаимодействие и произошедшие ранее; significant – активное взаимодействие и те вершины, которые еще будут использоваться на последующих шагах; и all – активное взаимодействие и все вершины, присутствующие в графе. Во всех режимах активные элементы имеют яркий цвет, прочие раскрашены в оттенки серого. Отдельно важно отметить поддержку отображения контейнеров — вершины, принадлежащие одному контейнеру, будут располагаться на плоскости рядом, выделены общим цветным блоком с именем контейнера. Интерактивные элементы данного графа позволяют как переключать описанные выше режимы, так и изменять время активного взаимодействия, а также свободно перемещать вершины графа по плоскости, приближать и отдалять граф.

Граф времени жизни процессов (рис. 7) представлен в виде временной шкалы, на которой изображены процессы в соответствии с временем и продолжительностью работы. Каждый из процессов расположен на отдельной строке графа, при этом принадлежащие одному контейнеру процессы располагаются по соседству и визуально выделены общим фоном. Граф поддерживает увеличение и уменьшение масштаба для поддержки анализа процессов короткой продолжительности, либо напротив, получения общего представления о возникавших в процессе работы системы процессах. При наведении мыши на процесс появляется всплывающий элемент, где отображается: имя процесса, идентификатор процесса, идентификаторы пользователей, путь к исполняемому файлу, строка запуска, родительский процесс и дочерние процессы. Оттенками красного на графе изображаются процессы, запущенные от root-пользователя, остальные - оттенками синего. Темными цветами выделяются процессы, которые оперировали помеченными данными. По двойному клику по таким процессам осуществляется переход на граф взаимодействия процессов в момент, когда выбранный процесс первый раз появляется на графе.

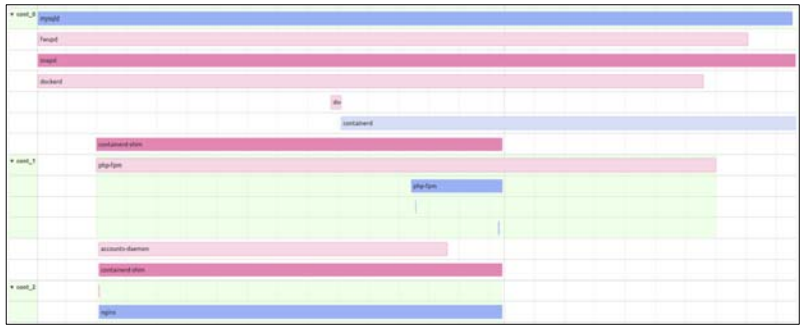

Рис. 7. Граф времени жизни процессов в интерфейсе SNatch
Fig. 7. Process lifetime graph in the SNatch interface

13. Примеры применения

Для демонстрации работоспособности Natch и SNatch рассмотрим несколько примеров.

Первый пример (рис. 8) продемонстрирует работу через пометку файлов. Сценарий следующий: есть две программы на языке C, одна из них читает файл (который и будет помечен), в первой строке которого хранится адрес сайта (в нашем случае www.google.ru), далее строка с этим адресом используется как параметр для вызова второй программы, которая запускает утилиту curl с переданным аргументом. Таким образом мы наблюдаем как помеченные данные передаются от процесса к процессу (в том числе и через командную строку) и в итоге утекают в сеть.

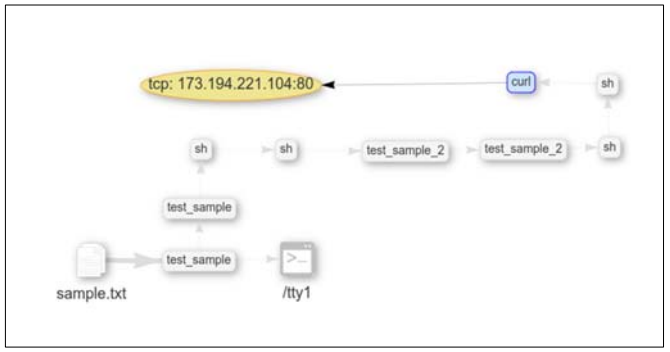

Рис. 8. Граф потоков данных между процессами в первом примере
Fig. 8. Graph of data flows between processes in the first example

Второй пример (рис. 9) тоже использует пометку файлов, но еще демонстрирует взаимодействие программ через сокеты. Сценарий примера: есть два Python скрипта – сервер и клиент. Для каждого подготовлен файл с сообщениями, которыми они будут обмениваться во время сессии (эти файлы помечаются). “Чат” клиента и сервера записывается в файл лог, который затем архивируется.

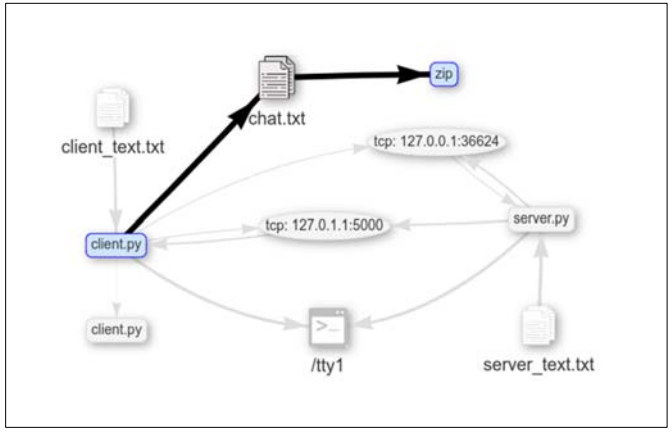

Рис. 9. Граф потоков данных между процессами во втором примере
Fig. 9. Graph of data flows between processes in the second example

14. Ограничения и планы развития

Текущая версия инструмента Natch способна работать для платформы x86_64 и ОС на основе ядер Linux и FreeBSD. Это ограничение возникло лишь из приоритетов развития, поэтому в дальнейшем будет добавлена поддержка процессорной архитектуры ARM и гостевой ОС Windows.

Другое ограничение – невозможность работы с вредоносным ПО – носит фундаментальный характер. Отслеживанию работы помеченных данных можно препятствовать, используя неявные каналы передачи информации, зависимости по управлению и т.п. Злоумышленник может создать программу, в которой поток данных отслеживаться не будет, поэтому Natch предназначен только для поиска поверхности атаки при разработке и сертификации, то есть когда разработчик не пытается противодействовать анализу поведения кода.

Извлекаемых методами интроспекции данных из виртуальной машины достаточно для построения более сложных и подробных отчётов, для навигации по исходному коду, для поиска обращений к заданным файлам. Поэтому графический инструмент SNatch планируется развивать для ещё большего упрощения работы аналитика.

Список литературы

[1] Климушенкова М.А., Бакулин М.Г. и др. О некоторых ограничениях полносистемного анализа помеченных данных. Труды ИСП РАН, том 28, вып. 6, 2016 г., стр. 11-26 / Klimushenkova M.A., Bakulin M.G. et al. On Some Limitations of Information Flow Tracking in Full-system Emulators. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 11-26 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-1.

[2] Фурсова Н.И., Довгалюк П.М. и др. Легковесный метод интроспекции виртуальных машин. Программирование, том. 43, вып. 5, 2017 г., стр. 39-47 / Fursova N.I., Dovgalyuk P.M. et al. A lightweight method for virtual machine introspection. Programming and Computer Software, vol. 43, issue 5, 2017, pp. 307–313.

[3] Davanian A., Qi Z. et al. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In Proc. of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2019, pp. 31-45.

[4] Dolan-Gavitt B., Leek T. et al. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In Proc. of the 20th ACM Conference on Computer and Communications Security (CCS), 2013, pp. 839-850.

[5] Qemu. A generic and open source machine emulator and virtualizer. URL: https://www.qemu.org/.

[6] Васильев И.А., Фурсова Н.И. и др. Модули для инструментирования исполняемого кода в симуляторе qemu. Проблемы информационной безопасности. Компьютерные системы, 2015 г., вып. 4, стр. 195-203 / Vasilev I., Fursova N. et al. Modules for instrumenting the executable code in QEMU simulator. Information Security Problems. Computer System, 2015, pp. 195-203 (in Russian).

[7] Dovgalyuk P. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proc. of the 2012 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 553-556.

[8] More A., Tapaswi S. Virtual machine introspection: towards bridging the semantic gap. Journal of Cloud Computing, vol. 3, 2014, article no. 16.

[9] Dovgalyuk P., Fursova N. et al. QEMU-based Framework for Non-Intrusive Virtual Machine Instrumentation and Introspection. In Proc. of the 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 944-948.

[10] Henderson A., Prakash A. et al. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform. In Proc. of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 248-258.

[11] Hizver J., Chiueh T.-C. Real-Time Deep Virtual Machine Introspection and Its Applications. ACM SIGPLAN Notices vol. 49, issue 7, 2014, pp. 3-14

[12] Dovgalyuk P., Vasiliev I. et al. Non-intrusive Virtual Machine Analysis and Reverse Debugging with SWAT. In Proc. of the IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020, pp. 196-203.

[13] Lin Z., Rhee J. et al. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2011, 18 p.

[14] Schneider C., Pfoh J., Eckert C. Bridging the semantic gap through static code analysis. In Proc. of the 5th European Workshop on Systems Security (EuroSec), 2012, 6 p.

[15] Volatility 3: The volatile memory extraction framework. Available at: https://github.com/volatilityfoundation/volatility3, accessed 06.12.2022.

[16] LibVMI. Available at: https://libvmi.com/, accessed 06.12.2022.

[17] Fu Y., Lin Z. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In Proc. of the IEEE Symposium on Security and Privacy. 2012, pp. 586-600.

[18] Saberi A., Fu Y., Lin Z. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memorization. In Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS'14), 2014, 15 p.

[19] Franzen F., Holl T. et al. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In Proc. of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), 2022, 18 p.

[20] Zhang S., Meng X., Wang L. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security, 2016, article no. 14, 13 p.

[21] Diaphora: A Free and Open Source Program Diffing Tool. Available at: http://diaphora.re/, accessed 06.12.2022.

[22] Feng Q., Prakash A. et al. Origen: Automatic extraction of offset-revealing instructions for crossversion memory analysis. In Proc. of the 11th ACM Asia Conference on Computer and Communications Security, 2016, pp. 11-22.

[23] Jiang X., Wang X. "Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots. Lecture Notes in Computer Science, vol. 4637, 2007, pp. 198-218.

[24] Dolan-Gavitt B., Srivastava A. et al. Robust signatures for kernel data structures. In Proc. of the 16th ACM Conference on Computer and Communications Security, 2009, pp. 566-577.

[25] Schwartz E.J., Avgerinos T., Brumley D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In Proc. of the IEEE Symposium on Security and Privacy, 2010, pp. 317-331.

[26] Bakulin M., Klimushenkova M., Egorov D. Dynamic Diluted Taint Analysis for Evaluating Detected Policy Violations. In Proc. of the 2017 Ivannikov ISPRAS Open Conference (ISPRAS), 2017, pp. 22-26.

[27] Luk C.-K., Cohn R. et al. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices, vol. 40, issue 6, 2005, pp. 190-200.

[28] Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Proc. of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 89-100

[29] Bruening D., Duesterwald E., Amarasinghe S. Design and implementation of a dynamic optimization framework for Windows. In Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), 2001, 12 p.

[30] Song D., Brumley D. et al. BitBlaze: A new approach to computer security via binary analysis. Lecture Notes in Computer Science, vol. 5352, 2008, pp. 1-25.

[31] Henderson A., Lok Y. et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. IEEE Transactions on Software Engineering, vol. 43, issue 2, 2017, pp. 164-184.

[32] Python scriptable Reverse Engineering Sandbox, a Virtual Machine instrumentation and inspection framework based on QEMU. Available at: https://github.com/Cisco-Talos/pyrebox, accessed 03.11.2022.

[33] Zeng J., Fu Y., Lin Z. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. ACM SIGPLAN Notices, vol. 50, issue 7, 2015, pp 147-160.

[34] Stallman R.M. and the GCC Developer Community. Using the GNU Compiler Collection (GCC). Available at: https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc.pdf, accessed 03.11.2022.

[35] Lighthouse - A Coverage Explorer for Reverse Engineers. Available at: https://github.com/gaasedelen/lighthouse/, accessed 03.11.2022

[36] Blasum H., Görgen F., Urban J. Gcov on an embedded system. In Proc. of the International Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07), 2007, 4 p.

[37] Husain A. Un-bee-lievable Performance: Fast Coverage-guided Fuzzing with Honeybee and Intel Processor Trace. Available at: https://blog.trailofbits.com/2021/03/19/un-bee-lievable-performance-fast-coverage-guided-fuzzing-with-honeybee-and-intel-processor-trace/, accessed 03.11.2022.

[38] IDA Pro, Available at: https://www.hexrays.com/ida-pro/, accessed 03.11.2022

[39] Yin H., Song D. et al. Panorama: capturing system-wide information flow for malware detection and analysis. In Proc. of the 14th ACM conference on Computer and communications security (CCS '07), 2007. Pp. 116-127.

[40] King S.T., Chen P.M. Backtracking intrusions. In Proc. of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 223–236.

[41] Goel A., Po K. et al. The taser intrusion recovery system. In Proc. of the 20th ACM Symposium on Operating Systems Principles, 2005, pp. 163-176.

[42] Ma S., Zhang X., Xu D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In Proc. of the Network and Distributed System Security Symposium, 2016, 15 p.

[43] Ji Y., Lee S. et al. RAIN: Refinable attack investigation with on-demand inter-process information flow tracking. In Proc. of the ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 377–390.

Информация об авторах / Information about authors

Павел Михайлович ДОВГАЛЮК – инженер, кандидат технических наук. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, отладчики, эмуляторы.

Pavel Mikhailovich DOVGALYUK – engineer, Ph.D. in Technical Sciences. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

Мария Анатольевна КЛИМУШЕНКОВА – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Maria Anatolyevna KLIMUSHENKOVA is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Наталья Игоревна ФУРСОВА – инженер, кандидат технических наук. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, эмуляторы.

Natalia Igorevna FURSOVA – engineer, Ph.D. in Technical Sciences. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, emulators.

Владислав Михайлович СТЕПАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Vladislav Mikhailovich STEPANOV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Иван Александрович ВАСИЛЬЕВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Ivan Aleksandrovich VASILIEV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Аркадий Алексеевич ИВАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Arkady Alekseevich IVANOV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Алексей Владимирович ИВАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Alexey Vladimirovich IVANOV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Максим Геннадьевич БАКУЛИН - разработчик программного обеспечения. Сфера научных интересов: эмуляция, динамический анализ помеченных данных, компьютерная безопасность.

Maksim Gennadievich BAKULIN is a sowtware engineer. Research interests: emulation, dynamic taint analysis, cyber security.

Данила Игоревич ЕГОРОВ - разработчик программного обеспечения. Сфера научных интересов: эмуляция, динамический анализ помеченных данных, компьютерная безопасность.

Danila Igorevich EGOROV is a sowtware engineer. Research interests: emulation, dynamic taint analysis, cyber security.