DOI: 10.15514/ISPRAS-2024-36(3)-9

Декларативный подход к задаче интроспекции виртуальной машины

[1] *В.М. Степанов,* ORCID: 0000-0003-2141-3333 <*vladislav.stepanov@ispras.ru*>
[1,2] *П.М. Довгалюк,* ORCID: 0000-0003-2483-5718 <*pavel.dovgalyuk@ispras.ru*>
[1] *Н.И. Фурсова,* ORCID: 0000-0001-6817-4670 <*natalia.fursova@ispras.ru*>

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

Аннотация. В инструментах анализа снимков памяти и интроспекции виртуальной машины большое внимание уделяется решению проблемы семантического разрыва. Важную роль в этой задаче играет наличие отладочных символов и знаний о смещениях внутри объектов ядра. Набор сведений о смещениях, как правило, называют профилем ОС. Методы генерации таких профилей опираются на внедряемые в систему агенты, отладочные символы, компиляцию исходного кода или бинарный анализ. Использование исключительно бинарного анализа для этой задачи делает возможным исследование с минимальными знаниями о выполняемой ОС. В статье представлен подход генерации профиля ОС, опирающийся на перехват системных событий в ходе работы виртуальной машины. В его основе лежит сопоставление данных, получаемых при разборе бинарного интерфейса приложений (ABI), с данными, извлекаемыми из предполагаемых мест расположения структур ядра. Преимуществом решения является его масштабируемость под поддержку новых ОС. В то время как другие существующие подходы выстраивают алгоритмы поиска на основе перехвата функций ядра Linux, обращающихся к искомым полям, текущий подход предлагает использовать проверки, схожие для разных семейств ОС. Представлен также способ описания эвристических алгоритмов для генерации профиля, который упрощает работу с ними, и делает их более устойчивыми к изменениям между версиями ОС.

Ключевые слова: виртуальные машины; мониторинг; QEMU; интроспекция.

Для цитирования: Степанов В.М., Довгалюк П.М., Фурсова Н.И. Декларативный подход к задаче интроспекции виртуальной машины. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 123–138. DOI: 10.15514/ISPRAS–2024–36(3)–9.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022, https://rscf.ru/project/24-11-20022/.

Declarative approach to virtual machine introspection

[1] V.*M. Stepanov*, ORCID: 0000-0003-2141-3333 <*vladislav.stepanov@ispras.ru*>
[1,2] *P.M. Dovgalyuk*, ORCID: 0000-0003-2483-5718 <*pavel.dovgalyuk@ispras.ru*>
[1] *N.I. Fursova*, ORCID: 0000-0001-6817-4670 <*natalia.fursova@ispras.ru*>

[1] *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. The prominent problem in memory dump analysis and virtual machine introspection approaches is a semantic gap. Availability of debug symbols or knowledge about kernel data structures offsets is very important for retrieving high-level information from binary code. A set of information about kernel data structures field offsets is called an OS profile. Methods of generating such profiles are based on guest agents, debug symbols, source code compilation or binary analysis. Using only binary analysis makes it possible to do research with a minimal knowledge about analyzed guest OS. In this paper we present a novel approach for OS profile generating. It is based on system call tracing and comparison between data obtained from application binary interface and data extracted from expected locations of kernel structures. The advantage of this solution is scalability for supporting different guest systems. While other existing approaches use heuristics based on handling Linux kernel functions that access the fields, the current approach suggests using heuristics that are similar across different OS families. We also suggest a method of describing heuristic algorithms for profile generation that simplifies understanding of them and makes them more resistant to changes between OS versions.

Keywords: virtual machines; monitoring; QEMU; introspection.

For citation: Stepanov V.M., Dovgalyuk P.M., Fursova N.I. Declarative approach to virtual machine introspection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 36, issue 3, 2024. pp. 123-138 (in Russian). DOI: 10.15514/ISPRAS-2024-36(3)-9.

Acknowledgements. The work was partially supported by the Russian Science Foundation (grant No. 24-11-20022, https://rscf.ru/project/24-11-20022/).

1. Введение

Технология интроспекции виртуальной машины является важным элементом множества инструментов динамического анализа, построенных на полносистемном эмуляторе. С ее помощью из извлекаемого во время работы эмулятора бинарного кода восстанавливается высокоуровневая информация о состоянии и деятельности объектов гостевой операционной системы. Традиционные инструменты анализа снимков памяти, такие как Volatility [1] и Rekall [2], применяют отладочные символы ядра ОС для определения адресов памяти, на которых располагаются те или иные параметры. Инструменты интроспекции по таким параметрам определяют выполняемые в системе процессы, открытые файлы и сетевые соединения. За счет инструментирования исполняемого кода они отслеживают вызовы функций и системные вызовы, по которым восстанавливают информацию о происходящих в системе событиях и изменениях. При всем этом механизмы интроспекции могут работать исключительно на уровне гипервизора без каких-либо внедряемых внутрь виртуальной машины агентов.

Необходимость наличия отладочных символов к ядру является ограничением применимости инструментов в ряде случаев. Разработчики дистрибутивов Linux могут не предоставлять их для своих продуктов, либо не хранить их для старых версий. Также отладочные символы могут оказаться недоступны, если ядро было скомпилировано пользователем. В связи с этим существует спрос на инструменты бинарного анализа, способные работать при отсутствии полной информации о ядре исследуемой ОС. Как одно из решений проблемы, есть подходы интроспекции, опирающиеся на бинарный интерфейс приложений (ABI) в качестве

основного источника информации о системе [3]. Основная идея здесь заключается в том, что ABI при изменении версий ядра достаточно редко меняется, поэтому одна реализация подхода может функционировать на целом семействе ОС. На этом построена трассировка системных вызовов. Для ее работы достаточно определить, какие машинные инструкции выполняются для системных вызовов, в каком регистре хранятся и каким функциям соответствуют их номера в той или иной системе, а также чем являются и как передаются их аргументы и возвращаемые значения. Все эти данные могут быть описаны в виде небольшого числа конфигурационных файлов, заточенных под разные ОС.

На основе отслеживания определенных системных вызовов реализуется мониторинг событий, связанных с файлами, сокетами и процессами. Такой мониторинг имеет ограничения, ведь не все интересующие аналитика параметры могут быть получены из аргументов этих вызовов. К тому же модули ядра могут взаимодействовать с файлами и другими требующими отслеживания объектами с помощью прямых вызовов функций, отслеживание которых в ходе интроспекции виртуальной машины, как правило, требует отладочные символы [4]. Как решение, способное комбинировать преимущества зависимых и независимых от отладочных символов подходов, существуют методы анализа структур данных ядра, способные определить необходимые для интроспекции смещения полей.

В DECAF [5] и PANDA [6] применяется модуль ядра, который внедряется в гостевую систему Linux, и в процессе работы определяет смещения структур данных. Такой модуль собирается специально под каждую ОС, и в случае систем без доступа на запись файлов его внедрение может оказаться недоступным. Другое решение заключается в применении эвристик, основанных на бинарном анализе. Удобной для пользователя инструмента анализа особенностью такого похода является отсутствие необходимости выполнять подготовительные работы с образом виртуальной машины. Недостатком же подхода может стать его неточность, связанная с ложными срабатываниями разработанных эвристических проверок.

Основная идея применения эвристик для поиска смещений полей в структурах данных заключается в использовании определенных закономерностей, таких как порядок, размеры и типы данных полей. Закономерности могут определяться вручную разработчиком инструмента, либо могут находиться автоматизировано на основе каких-либо данных о системе. Недостатком ручных подходов является их плохая масштабируемость, ведь на каждый извлекаемый из ядра параметр разработчиком должен быть определен алгоритм его нахождения. В то же время, при изменениях между версиями ядра старые методы могут потребовать доработки. Автоматизированные подходы, как правило, нацелены на извлечение большого количества параметров и адаптацию под происходящие в ядре изменения, однако в существующих решениях возможности такой адаптации могут быть ограничены.

В этой статье рассмотрены методы разбора структур данных, которые не опираются на внедряемые агенты и отладочные символы ядра, а также дано описание подхода к построению профиля интроспекции, который применяется в инструменте динамического анализа помеченных данных Natch.

2. Обзор существующих решений

RAMPARSER [7] представляет собой инструмент для анализа снимков памяти. Восстанавливает смещения полей структур данных, опираясь на вручную заготовленные разработчиком эвристики. Суть эвристик заключаются в перехвате определенных выбранных функций, которые взаимодействуют с искомыми полями. В этих функциях отслеживаются инструкции доступа к памяти, по которым определяются адреса полей и вычисляются их смещения. Недостатком подхода является сложность реализации эвристик, так как разработчик должен учитывать, что код функций может меняться в зависимости от версии ядра. Также для перехвата используемых в эвристиках функций в обязательном порядке

необходимо наличие для ядра файла System.map. Схожим образом работает инструмент RamAnalyzer [8], но для перехвата функций он извлекает информацию об отладочных символах из бинарного кода системы через механизм kallsyms.

LogicMem [9] выполняет анализ снимков памяти с ОС Linux, находит в них список из структур данных task_struct и разбирает параметры объектов ядра. Для определения смещений полей используются логические правила на языке Prolog. Подход показывает высокую точность результатов, и при этом опирается только на бинарный анализ. Недостатком является тот факт, что используемые в правилах эвристики привязываются к порядку расположения полей, из-за чего данный подход не будет работать при включенной рандомизации структур данных. Тем же недостатком обладает подход, использующий сигнатуры для поиска скрытых вредоносной программой объектов ядра [10].

Подход Hybrid-bridge [11] опирается на запускаемый внутри виртуальной машины инструмент анализа, который реализует обращения к объектам ядра. Выполняемые в ходе работы гостевого инструмента операции доступа к памяти преобразуется к коду, выполняемому на уровне гипервизора. Ограничением подхода является необходимость наличия возможности установки программ в гостевую систему.

ORIGEN [12] использует возможности фреймворка BinDiff [13] по сопоставлению между собой функций из бинарного кода разных версий одних и тех же программ. Сначала определяются смещения полей в структурах данных некоторого эталонного ядра. На основе динамического анализа отслеживаются инструкции операций доступа к этим полям. Затем выполняется сопоставление бинарного кода эталонного ядра с анализируемым, за счет чего в целевом ядре находятся эквивалентные инструкции операций доступа. На основе этих инструкций восстанавливаются смещения искомых полей. На практике возможности этого подхода продемонстрированы только для небольшого количества параметров.

Подход Katana [14] предлагает использовать для определения нужных для интроспекции смещений taint анализ. Прежде всего, на основе исходного кода ОС Linux строится база данных с информацией о функциях, выполняющих к полям операции доступа. В то же время, определяются потоки данных между аргументами функций и операциями доступа к искомым параметрам. Далее берется снимок памяти ядра и путем его сканирования определяется таблица символов с информацией об экспортированных функциях и глобальных переменных. С помощью эмулятора в контекст анализируемого снимка добавляется вызов функции kallsyms_on_each_symbol с заранее заготовленной callback функцией, что позволяет собрать информацию обо всех функциях ядра Linux кроме статически скомпилированных. После этого Katana сопоставляет машинный код определенных функций с ранее сгенерированной базой данных. К входным параметрам и возвращаемым значениям функций применяется taint-анализ, и по операциям доступа с помеченными параметрами определяются смещения искомых полей структур данных ядра.

К преимуществам Katana относятся адаптируемость подхода к изменениям в коде ядра, устойчивость к рандомизации структур данных, а также построение последовательности действий по поиску нужного смещения без ручных усилий со стороны разработчика, что обеспечивает высокую масштабируемость решения.

3. Модель декларативного описания эвристик интроспекции

Рассмотрим подход к разбору структур данных ядра, который лег в реализацию инструмента определения поверхности атаки Natch. Решение основано на эвристических алгоритмах, выполняемых в ходе динамического анализа. Алгоритмы подбираются вручную под каждый извлекаемый параметр. При этом был реализован способ представления эвристик, который упрощает их внедрение и повышает масштабируемость решения под новые запросы. В отличие от подхода Katana, который автоматизирует процесс построения алгоритмов нахождения полей, данный подход не опирается на исходный код ОС и извлекаемые из ядра

имена функций. Это позволяет адаптировать его под отличные от Linux системы, такие как Windows и FreeBSD, используя в основе одни и те же механизмы и схожие эвристики.

Восстановление смещений полей структур данных выполняется в ходе работы программного эмулятора QEMU [15]. На протяжении загрузки гостевой ОС перехватываются системные вызовы, на основе которых работает независимая от отладочных символов интроспекция. Получаемые от такой интроспекции данные используются для проверки интроспекции, основанной на структурах данных. Для описания порядка подбора и проверок смещений реализуется модель, обладающая следующими особенностями:

(1) Для выполнения вычислений эмулятор останавливается на определенных событиях, задаваемых моделью. При этом один алгоритм подбора смещений может содержать в себе несколько остановок.

(2) Смещения полей структур данных перебираются в пределах определенных диапазонов или наборов возможных значений.

(3) Корректность подбираемых смещений подтверждается в ходе специальных проверок. Проверочные функции могут оперировать данными системных вызовов и вычисляемыми параметрами предыдущих связанных с ними проверок.

(4) Между подбором смещения указателя на структуру данных и подбором смещений полей этой структуры может быть задана такая связь, что перебор смещений полей будет повторно выполняться на каждое возможное смещение указателя.

(5) Могут быть заданы альтернативные пути извлечения параметров для случаев, когда некоторые поля структур данных отсутствуют в определенных версиях ОС.

3.1 Перебор смещений и проверочные функции

Допустим, существует механизм разбора структур данных, который в ходе анализа отвечает на запросы от мониторов файлов, процессов и модулей. Эти мониторы реализуют интроспекцию виртуальной машины, совмещая информацию, получаемую от системных вызовов, с запросами различных параметров из структур данных ядра. Тогда генерация профиля для гостевой ОС может быть реализована следующим образом. Определяются тесты, которые проверяют результат выполнения запросов к механизму разбора структур данных и сравнивают его с параметрами, получаемыми из других источников. Далее в ходе настроечного запуска эмулятора совместно с выполнением тестов реализуется перебор смещений полей. Смещения перебираются либо по наборам заранее известных значений, либо в определенных диапазонах с некоторым шагом. Например, для поля с указателем может быть задан диапазон перебора от нуля до предполагаемого максимального размера структуры данных с шагом 8 байт. При успешном прохождении тестов набор смещений, с которыми удалось их пройти, сохраняется и заносится в статистику для дальнейшего уточнения.

Тесты должны выполняться отдельно на каждый набор подбираемых смещений. Как итог, на одном остановленном состоянии виртуальной машины проверки могут выполняться много раз. В связи с этим встает вопрос оптимизации. Опишем три ключевые особенности, которые позволяют выполнять перебор быстрее:

(1) Проверки делятся на этапы таким образом, чтобы независимые от перебираемых смещений вычисления выполнялись единожды непосредственно перед перебором. Вычисленные параметры передаются от одного этапа к следующему.

(2) Так как при извлечении некоторых параметров возникает необходимость использовать сразу несколько смещений полей, функции разбора структур данных предусматривают возврат ошибок с указанием элемента, из-за которого ошибка возникла. Значения смещений полей, которые вызывают ошибку, пропускаются, что ускоряет их общий перебор.

(3) Также в ходе перебора может возникать ситуация, когда на одном и том же состоянии виртуальной машины происходит множество обращений подряд к одним и тем же переменным из гостевой памяти. В связи с этим выполняется кэширование значений параметров. Если смещения на пути к параметру не меняются, то выдается ранее сохраненное для текущего состояния значение.

Последние две оптимизации связаны с тем концептом, что функционал разбора структур данных разрабатывается под нужды конечного инструмента интроспекции, а не под нужды конкретных проверок эвристик. Это означает, что если в ходе запроса на получение имени файла требуется обратиться к множеству полей структур данных, то во время проверок этот запрос будет использован без изменений. Теоретически, для проверочных тестов можно реализовать множество мелких запросов на каждое искомое поле структуры данных, но это в значительной мере усложняет описание эвристик. В таком случае при изменениях алгоритмов разбора структур данных под новые версии гостевого ядра возникает необходимость добавлять новые проверки. Применение же более сложных запросов делает проверяемые эвристики более устойчивыми к изменениям. Проверочные функции, которые опираются на сложные запросы, могут даже без изменений использоваться в генерации профилей разных семейств ОС.

Объясним это на примере. Для задач мониторинга состояния гостевой ОС Linux разработаны механизмы получения идентификаторов процесса и имен файлов из структур данных ядра. В то же время, имеются способы получения той же информации на основе системных вызовов. Идентификатор текущего процесса может быть получен при перехвате вызова getpid, а имена файлов могут быть получены и сопоставлены номерам файловых дескрипторов на основе вызова open.

В первом случае проверка эвристики будет выполняться на моменте завершения getpid. В начале извлекается pid из регистра с возвращаемым значением, затем выполняется перебор смещений, необходимых для получения pid текущего процесса из структур ядра. С каждым набором смещений выполняется проверка, в ходе которой запрашивается предполагаемый идентификатор текущего процесса из ядра, после чего полученное значение сравнивается со значением pid из системного вызова.

В случае с файлами проверка выполняется на моменте завершения вызова open. На предварительном этапе имя файла извлекается из аргумента системного вызова, а номер файлового дескриптора из возвращаемого значения. Далее выполняется перебор смещений, необходимых для получения имени файла по его номеру дескриптора. На каждый набор смещений имя файла извлекается из ядра и сравнивается с именем, полученным от системного вызова. При успехе найденные смещения сохраняются, а в случае неудачных попыток, подбираемые смещения корректируются в соответствии с возвращаемой ошибкой.

3.2 Статические проверки

Далее раскроем принцип описания эвристик. Для этого будем использовать ориентированный граф, узлы которого могут задавать точки останова, подбираемые смещения и функции проверок (рис. 1). Прежде всего в таком графе всегда присутствует корневой узел с указанием точки останова. Такой точкой может быть начало или завершение системного вызова, либо любое другое событие, которое перехватывается в эмуляторе. Стрелки между узлами задают последовательность действий. Если от одного узла исходит несколько стрелок, то между этими стрелками ставится знак конъюнкции или дизъюнкции. При этом, задаваемые узлами действия выполняются в порядке обхода графа в глубину. Если встречается узел с перебором, то следующие узлы повторно выполняются на каждое подобранное значение смещения. На узлах с функциями проверки происходит отсеивание некорректных смещений. Другими словами, когда функция при выполнении выдает значение “ложь”, текущий набор смещений отбрасывается, и выполнение возвращается к последнему

узлу перебора. В случае же, когда функция проверки выдает значение “истина”, то выполняются следующие далее узлы, либо если это происходит на концевом узле графа, то подобранные на пути к данному узлу смещения сохраняются, а родительская ветвь, ведущая от ближайшего разделения или начала графа отмечается успешно пройденной.

Задаваемые функции проверок могут оперировать вычисленными переменными из родительских узлов, а также извлекать параметры из гостевой памяти, используя подобранные в родительских узлах смещения. При подборе независимых друг от друга параметров используется разделение на разные ветки графа со знаком конъюнкции. При обходе таких разделений, каждая следующая ветвь графа будет выполняться только в том случае, если предыдущие были пройдены как минимум с одним набором смещений. Если хоть одна ветвь не была успешно пройдена, то выполнение отбрасывается к последнему узлу перебора перед ветвлением. Если все ветви успешно пройдены, то узлы, предшествующие разделению, тоже считаются успешно пройденными.

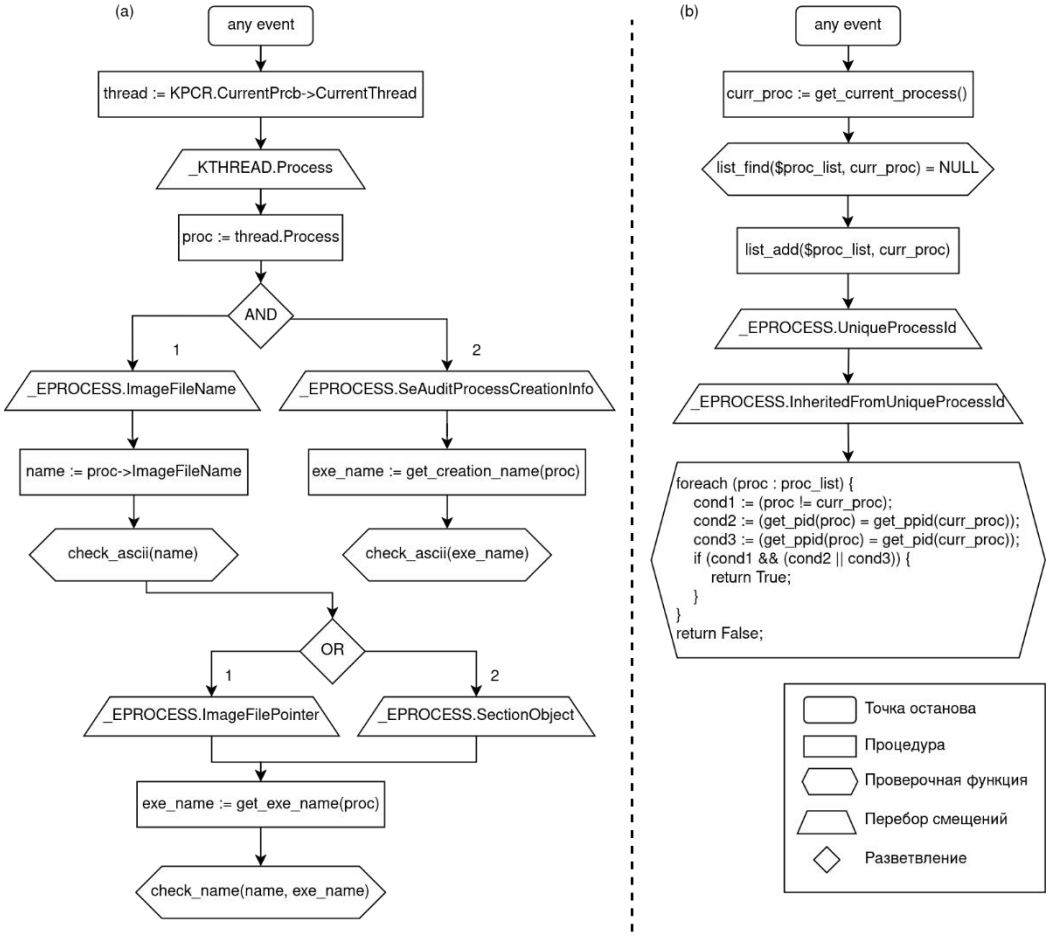

Рис. 1. Эвристики для имен и идентификаторов процессов в ОС Windows
Fig. 1. Process name and ID heuristics for Windows

Дизъюнкция при описании эвристик применяется в тех случаях, когда для разных версий ядра ОС необходимо находить разные параметры. Другими словами, мы пробуем извлечь нужную информацию из гостевой памяти одним способом, и, если не получается, применяем

другой. Ветвь, предшествующая разделению с дизъюнкцией, считается успешно пройденной, если успешно пройдена хотя бы одна из исходящих ветвей графа.

В итоге обход графа может приводить к одному из двух результатов. Если в соответствии с задаваемыми ветвлениями правилами корневой узел графа получает пометку успешного прохождения, то найден как минимум один набор смещений параметров, который удовлетворяет описанной графом эвристике. В обратном случае найти удовлетворяющие заданным условиям смещения не удается, поэтому проверки будут повторены на следующей точке останова.

В случае успешного прохождения эвристики должен образоваться граф из сохраненных значений смещений, прошедших проверки. Путем обхода этого графа восстанавливаются все возможные наборы смещений, удовлетворяющие заданным эвристикой условиям. На узлах конъюнкции сохраненные значения смещений из всех исходящих веток объединяются во всевозможные комбинации. В узлах дизъюнкции берутся сохраненные значения смещений только из одной ветки за раз. Получаемые наборы смещений далее заносятся в статистику, после чего описанная графом эвристика повторно проверяется на других попаданиях в точку останова. Когда один и тот же набор смещений успешно пройдет проверки определенное заданное количество раз, этот набор будет выдан как конечный результат этих проверок.

3.3 Динамические проверки

Выше мы описали как происходит обработка графа, в котором есть только один узел точки останова. Теперь раскроем принцип построения динамических проверок эвристик. Их суть заключается в выполнении разных этапов проверки эвристики на разных состояниях виртуальной машины. На графе это задается путем добавления дополнительных узлов точки останова.

Когда выполнение задаваемых узлами графа действий упирается в узел с точкой останова, подобранные на пути к этому узлу смещения сохраняются, как это делается при достижении концевых узлов графа. Сохраняются также вычисленные параметры, передаваемые от родительских узлов. Выполнение же следующих узлов откладывается до момента попадания в заданную точку останова. Но перед этим проверяется, что все ветви графа, соответствующие текущей точке останова и предшествующие узлам с другой точкой останова успешно проходят проверки как минимум с одним набором смещений. Если это так, то формирование итогов по проверке эвристики откладывается до полного обхода графа, либо отклонения его результатов. Далее при входе в соответствующие точки останова восстанавливаются сохраненные состояния с найденными смещениями и вычисленными параметрами, после чего обход графа продолжается со следующего узла.

Дадим пример эвристики, основанной на описанном выше принципе. В ОС Linux в структурах данных ядра есть параметр, который хранит состояние завершения процесса. Это поле exit_state в task_struct. На основе отслеживания его значения удается определить момент, когда процесс заканчивает свою работу. В большинстве случаев, но не всегда, возможно также определить завершение процесса по системному вызову exit. Основываясь на этой информации, определим следующий алгоритм (рис. 2а). Корневым узлом графа является точка останова на начале системного вызова exit. Далее идет перебор смещения искомого поля, а затем функция, которая извлекает состояние завершения из ядра и проверяет, что на этот момент процесс еще не завершен. Следующий узел задает другую точку останова, которая будет активирована при прерывании или обновлении таблицы страниц в гостевой системе. А за точкой останова следует узел с еще одной проверкой состояния процесса, но на этот раз проверка считается пройденной, если процесс оказывается завершен. Таким образом, с помощью проверок на разных точках останова формируется эвристика, опирающиеся на отслеживание изменений в искомых параметрах.

Другой пример динамической проверки, это нахождение смещения поля f_pos в структуре file (рис. 2b). Это поле, в котором хранится информация о текущей позиции чтения или записи файла. Корневым узлом графа является точка останова на начале системного вызова read. Далее идет узел с перебором искомого смещения, а за ним выполняется процедура с сохранением текущего значения f_pos. Второй точкой останова здесь является завершение вызова read. Проверяется, что значение f_pos между двумя состояниями изменилось на величину прочитанных в ходе системного вызова данных.

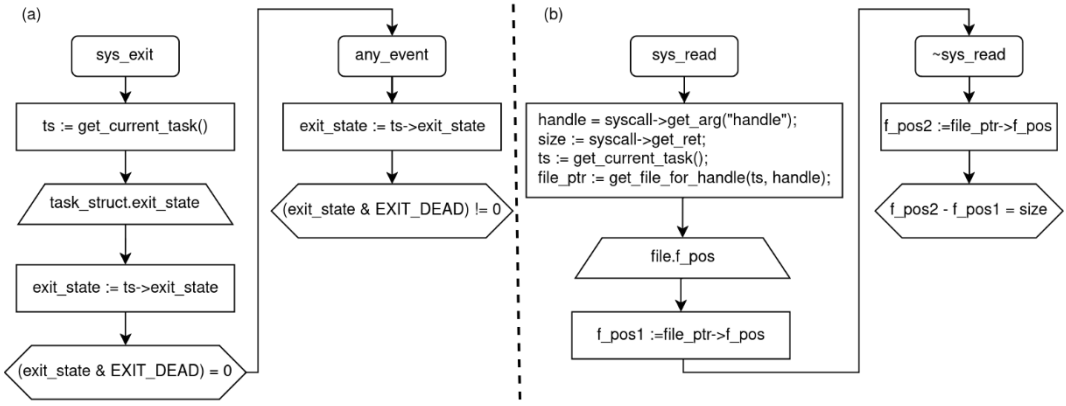

Рис. 2. Эвристики с завершением процесса и чтением файла.
Fig. 2. Heuristics for process termination and file reading

3.4 Построение сложных эвристик

Оперируя всеми вышеизложенными механизмами, гораздо проще описать сложные эвристические алгоритмы. С помощью узлов графа и связей между ними реализуется элемент декларативного программирования, благодаря которому можно абстрагироваться от реализации переборов смещений и передач параметров между разными этапами проверок. Граф в первую очередь описывает ожидаемый результат при корректно подобранных смещениях искомых параметров. Поэтому возможен и такой вариант использования эвристик, когда смещения параметров берутся из файла конфигурации, и затем выполняется обход графа с проверками, но пропуском узлов перебора. Таким образом, на основе того же представления реализуется тестирование задаваемых конфигурационным файлом смещений на соответствие выполняемому ядру в образе виртуальной машины.

При формировании сложных многоэтапных эвристик актуален еще один механизм. Это возможность повторно перебирать смещения одних и тех же параметров. В этом есть потребность, если целью первичного подбора является определение смещений указателей на пути к параметру, но смещение самого параметра следует уточнить на других этапах проверок. Вторичный перебор встраивается в параллельно идущие относительно первичного перебора ветви графа. Это объясняется тем, что на узлах, следующих за узлом перебора, в ходе выполнения смещение поля уже определено единственным значением. Следовательно, вторичный перебор может быть встроен только в ту ветвь графа, где подбираемый параметр еще не известен. Во вторичном переборе участвуют только найденные ранее смещения. Прежде всего для такого узла находится общий с предыдущим перебором родительский узел. Здесь формируется ограничение для области вторично подбираемых смещений. Во вторичный перебор подставляются только те наборы смещений из ветви с предыдущим перебором, для которых смещения, предшествующие общему узлу, совпадают с актуальными на момент вторичного перебора смещениями.

Для пояснения используем пример с ОС Linux (рис. 3). На моменте завершения системного вызова getpid выполняется поиск смещения для идентификатора процесса, поля tgid. Вместе

с этим подбирается смещение параметра current_task, т. е. указателя на текущий task_struct, а также смещения его полей comm, parent и group_leader. На основе наличия четырех полей в структуре подтверждается, что указатель действительно содержит адрес task_struct. Чтобы подтвердить, что указатель меняет значение при переключении процесса, добавляется дополнительная проверка с точкой останова в другом вызове getpid. На этом этапе должно определиться единственное смещение параметра current_task, но для других параметров, как правило, находится несколько вариантов смещений. Связано это с тем, что в task_struct некоторые поля принимают одинаковые значения. К ним относятся tgid и pid, первый из которых является идентификатором процесса, а второй идентификатором потока. Разделить их можно при остановке в другом состоянии виртуальной машины, где их значения не будут равны.

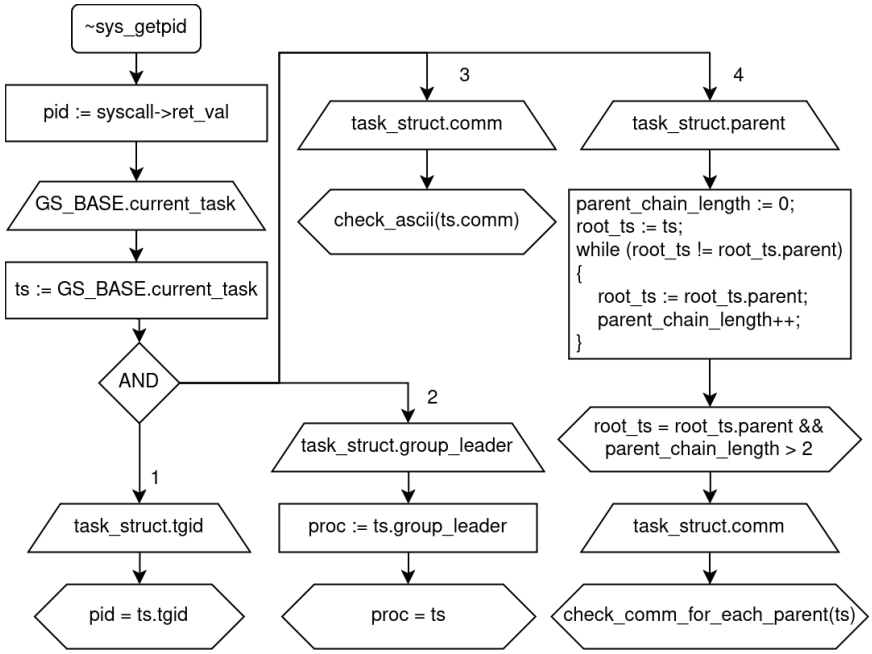

Рис. 3. Эвристики для основных полей task_struct
Fig. 3. Heuristics for main task_struct fields

Для определения того, какой из идентификаторов чем является удобно использовать параметр group_leader. Для потоков этот параметр указывает на основной task_struct процесса, которому принадлежит поток. Соответственно, на отдельной точке останова, следующей за проверкой параметра group_leader, добавляется вторичный перебор смещения tgid и перебор смещения pid. Далее идет проверка, что значения tgid и pid в текущем task_struct различаются и текущее значение pid отличается от значения того же поля в главном task_struct процесса. Что касается общего узла между первичным и вторичным перебором tgid, то таким узлом в данном случае является проверка параметра current_task. Исходя из этого, при вторичном переборе могут использоваться смещения tgid, найденные только для текущего значения current_task.

В поле comm хранится имя процесса. При первичном переборе его смещения проверяется только наличие в нем валидной строки из символов в кодировке ASCII. Вторичный перебор встраивается в проверку параметра parent, который является указателем на родительский task_struct. Проверяется, что для всех родительских процессов на заданном смещении есть валидная строка. При построении более сложной эвристики с файловыми структурами данных, распознаваемыми в ходе системного вызова open, перебор поля comm добавляется в

третий раз. На этом этапе строка comm сравнивается с полным именем процесса, которое извлекается из файла, указываемого полем exe_file структуры mm_struct.

Путем построения более сложных эвристических алгоритмов удается значительно повысить их точность. Благодаря этому удается не полагаться на подсчет статистики для наборов смещений, либо значительно сократить количество необходимых повторений этих наборов, что важно при восстановлении структур данных ядра для легковесных дистрибутивов, в ходе загрузки которых происходит очень малое количество системных событий.

4. Описание эвристик

Восстановление структур данных ядра было реализовано для гостевых ОС Linux и Windows. Оно предназначено в первую очередь для получения информации о выполняемых потоках и процессах (идентификаторы, имена, иерархия порождений, состояния завершения), открываемых файлах (имена, точки монтирования, позиции чтения/записи) и областях виртуальной памяти (отображаемые в память файлы, диапазоны адресов, флаги и др.). Для Linux также реализованы алгоритмы по извлечению команды запуска, идентификаторов пользователей для процессов и метаданных для сокетов.

4.1 Эвристики Linux

Для задач интроспекции виртуальной машины наиболее важной структурой данных ядра Linux является task_struct. Прежде всего необходим способ получения соответствующего текущему потоку исполнения адреса task_struct, который доступен из любого состояния виртуальной машины. В зависимости от версии ядра и эмулируемой платформы эта задача решается по-разному. В ядре Linux 2.4.0 это значение извлекается из стека ядра. В более современных версиях на платформе x86_64 адрес находится на определенном подбираемом смещении относительно регистра GS.

В разделах 3.3 и 3.4 в качестве примеров были разобраны алгоритмы нахождения основных параметров task_struct, они представлены на рис. 2 и 3. Процесс разбора файловых структур описывался в разделе 3.1, схема его алгоритма представлена на рис. 4a. Далее разберем проверки, выполняемые для нахождения смещений в структурах данных mm_struct.

Проверка поля mm_struct.exe_file не зависит от параметров системных вызовов, поэтому может быть выполнена на любой точке останова. Это поле является указателем на структуру file, параметры которой могут быть разобраны, если определяемые в ходе файловых проверок смещения уже известны. Суть проверки на рис. 4b заключается в том, что на основе поля exe_file восстанавливается полное имя исполняемого файла текущего процесса, после чего это поле сравнивается с именем процесса, получаемым из параметра task_struct.comm.

Алгоритм нахождения смещений, необходимых для получения информации об используемых процессом областях памяти представлен на рис. 4c. Он выполняется на моменте завершения системного вызова mmap. На ядрах с версией ниже 6.0.0 информация об областях памяти может быть получена путем чтения списка структур vm_area_struct. Для этого необходимо подобрать смещения полей mm_struct.mmap и vm_area_struct.vm_next. На более новых ядрах эти поля были удалены, но появилось поле mm_struct.mm_mt, которое указывает на новую структуру данных maple_tree. В связи с этим на схеме присутствует узел ветвления со знаком дизъюнкции, в соответствии с которым будут сделаны попытки получить искомые данные и старым, и новым методом.

В первую очередь проверяется первая структура областей памяти. Она должна описывать отображение исполняемого файла в память. Поле vm_file для нее будет равно значению mm_struct.exe_file. Поля vm_start и vm_end в структуре vm_area_struct хранят адреса начала и конца области памяти. При этом, что список, что дерево maple_tree организованы таким образом, что их элементы можно обойти в порядке возрастания адресов. На этой особенности и построен подбор смещений полей с адресами. Поле vm_flags определяется на основе

характерных значений, которые в него записываются. И как завершающая проверка, выполняется поиск элемента vm_area_struct, описывающего новое отображение файла, создаваемое системным вызовом mmap.

Кратко опишем другие реализованные проверки эвристик для Linux. Определение смещений на пути к параметрам сокетов выполняется на моменте завершения системного вызова bind.

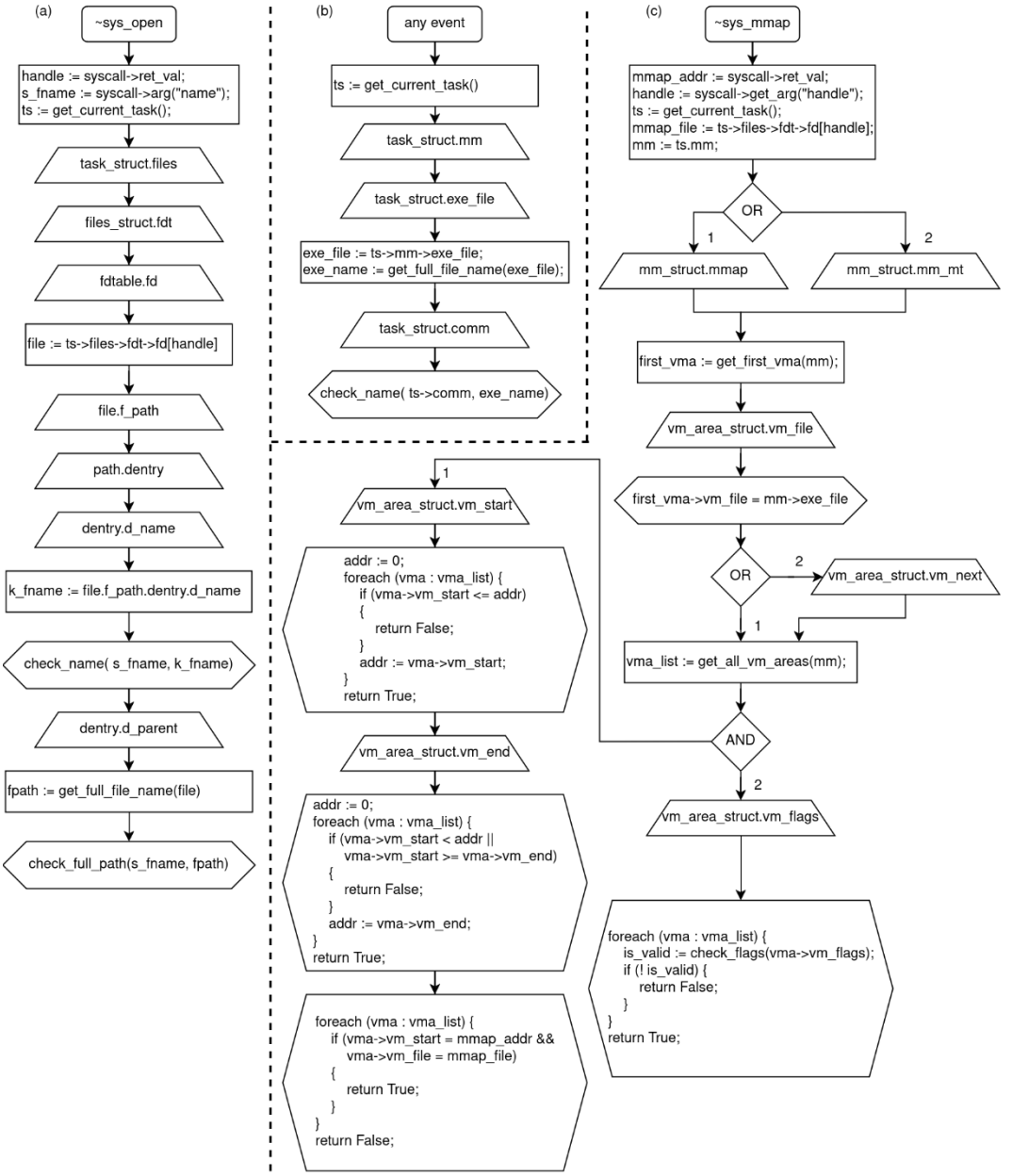

Рис. 4. Эвристики для структур files_struct и mm_struct
Fig. 4. Heuristics for files_struct and mm_struct structures

Проверяется, что сопоставляемое unix сокету имя соответствует тому, которое можно извлечь из соответствующих ему файловых структур ядра. Смещения полей, необходимых

для извлечения имени точки монтирования, определяются по системному вызову open. В начале ожидается вызов open, в аргументе которого полное имя файла длиннее того, что извлекается из ядра с помощью файловых структур по номеру дескриптора. Далее перебираются необходимые смещения структур монтирования и проверяется, что из них восстанавливается недостающая часть имени. Параметры uid находятся по системным вызовам getuid и geteuid. Поля, по которым восстанавливаются аргументы командной строки и переменные окружения могут проверяться на любом системном событии. Для их определения среди структур vm_area_struct находится та, которая содержит диапазон адресов стека. Далее поля args_start, args_end, env_start, env_end в mm_struct находятся по тому признаку, что хранимые в них значения лежат в диапазоне адресов стека и следуют друг за другом.

4.2 Эвристики Windows

В ОС Windows основной структурой данных ядра является структура KPCR, адрес которой для текущего выполняемого ядра всегда записывается в регистр GS. Через нее реализуется доступ к структуре данных ETHREAD для текущего потока и EPROCESS для текущего процесса. В целом, здесь нет необходимости для каждого поля перебирать значения смещений от нуля и до максимального размера структуры, так как эти смещения не могут быть рандомизированы, а количество дистрибутивов Windows невелико. Поэтому для каждого поля задается набор возможных значений, взятых из разных версий ядра. Эта особенность позволяет получать высокую точность подбора при достаточно простых эвристиках.

Проверка имен процессов (рис. 1a) работает по схожему с Linux способу, то есть проверяется наличие строки в кодировке ASCII на предполагаемом смещении. Короткое имя процесса в Windows хранится в поле EPROCESS.ImageFileName, а полное имя исполняемого файла восстанавливаются либо через EPROCESS.ImageFilePointer, либо через EPROCESS.SectionObject в зависимости от версии ядра.

В отличие от Linux, в Windows нет системного вызова getpid, поэтому необходим иной подход к определению смещения поля с идентификатором процесса. Также, в структуре EPROCESS нет указателя на такую же структуру с параметрами родительского процесса, но есть поле с идентификатором родительского процесса. Исходя из этого формулируется следующая проверка (рис. 1b). Собирается коллекция адресов EPROCESS для выполняемых процессов. Для каждого нового процесса чей адрес добавляется в коллекцию, выполняется перебор смещений полей с идентификаторами. На каждую пару смещений выполняется попытка найти среди ранее выполняемых процессов родительский или дочерний процесс текущего.

Информация об областях памяти собирается путем разбора бинарного дерева поиска, на которое указывает поле EPROCESS.VadRoot. С этими параметрами существует вариативность, связанная с тем, что в старых версиях Windows узлы дерева описывались структурами MMADDRESS_NODE, а в более новых ядрах используется структура RTL_BALANCED_NODE. В остальном принцип проверки этих параметров похож на тот, что используется для Linux. Проверяется, что адреса начала и конца для областей памяти возрастают при последовательном обходе элементов.

Проверка смещений файловых структур выполняется по системному вызову open, сравниваются имя файла из входного аргумента и имя файла из ядра. Здесь особенность заключается в том, что необходимо подбирать не только смещения полей на пути к метаданным файла, но и значения некоторых констант, используемых в вычислениях. Это связано с тем, что доступ к таблице файловых дескрипторов в Windows реализуется через поле HandleTable.TableCode, которое само по себе не является указателем, но его можно преобразовать к указателю с помощью побитового накладывания масок и сдвигов.

Собственно константы, применяющиеся в этих преобразованиях, различаются в разных версиях ядра, поэтому по ходу проверок эвристик выполняется их подбор.

Завершение процессов определяется по параметру EPROCESS.Flags. Для нахождения его смещения проверка выполняется на момент начала системного вызова TerminateThread. По аналогии с проверкой параметра exit_state из Linux отслеживается, что процесс жив на момент вызова, но на следующем состоянии виртуальной машины в это поле уже должен быть записан флаг смерти процесса.

5. Заключение

Работоспособность решения была проверена на дистрибутивах Ubuntu, Debian, Fedora, Centos и других с версиями ядра Linux от 2.4 до 6.8, а также на ОС Windows с версиями ядра NT от 6.1 до 10.0.

Важной особенностью разработанного подхода является сходство проверяемых эвристик на ОС Linux и Windows. Фактически, некоторые из используемых для них проверок одинаковы для обеих ОС, и различаются только перебираемые смещения. В ходе дальнейших исследований может быть добавлена поддержка генерации профиля для ядра ОС FreeBSD. Есть основания полагать, что под нее достаточно адаптировать уже разработанные для Linux проверки.

Также преимуществом подхода является его простота в применении. На практике генерация профиля для готового образа виртуальной машины должна занимать около пары минут, что, как правило, быстрее скачивания отладочных символов с искомыми смещениями с сайта разработчика дистрибутива или внедрения агентов в гостевое ядро.

Ограничением применяемого подхода является его зависимость от наличия системных событий в ходе настроечного запуска эмулятора. Для разбора структур данных, описывающих сокеты, необходим перехват сетевых системных вызовов, а их в ходе загрузки ОС может не оказаться. Возможным решением этой проблемы, а также направлением будущей работы, может стать механизм внедрения в гостевой код нужных системных вызовов с проверкой ответных действий ядра ОС.

Список литературы / References

[1]. Volatility 3: The volatile memory extraction framework. Url: https://github.com/volatilityfoundation/volatility3, дата обращения 12.04.2024.

[2]. Rekall Memory Forensic Framework. Url: https://github.com/google/rekall, дата обращения 12.04.2024.

[3]. Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 944–948. DOI: https://doi.org/10.1145/3106237.3122817.

[4]. DRAKVUF. Black-box Binary Analysis System. Url: https://drakvuf.com, дата обращения 12.04.2024.

[5]. Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 248–258. DOI: https://doi.org/10.1145/2610384.2610407.

[6]. Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5). Association for Computing Machinery, New York, NY, USA, Article 4, 1–11. DOI: https://doi.org/10.1145/2843859.2843867.

[7]. Richard Golden, Andrew Case, and Lodovico Marziale. 2010. Dynamic Recreation of Kernel Data Structures for Live Forensics. Digital Investigation 7 (2010), 32–40.

[8]. Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security 2016, 1 (2016), 14.

[9]. Zhenxiao Qi, Yu Qu, and Heng Yin. 2022. LogicMem: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In Network and Distributed System Security Symposium (NDSS).
[10]. Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In Proceedings of the 16th ACM Conference on Computer and Communications Security. 566–577.
[11]. Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memorization. In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14).
[12]. Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. Origen: Automatic extraction of offset-revealing instructions for crossversion memory analysis. In Proceedings of the 11th ACM on Asia Conference.
[13]. Bindiff. Url: https://www.zynamics.com/bindiff.html, дата обращения 12.04.2024.
[14]. Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus. ACM, New York, NY, USA, 18 pages.
[15]. Bellard, F.: QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX. Annual Technical Conference, 2005, pp. 41–46.

Информация об авторах / Information about authors

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.

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

Pavel Mikhailovich DOVGALYUK – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

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

Natalia Igorevna FURSOVA – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, emulators.