Использование ABI для интроспекции виртуальных машин

Н.И. Фурсова <natalia.fursova@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
И.А. Васильев <ivan.vasiliev@ispras.ru>
Новгородский государственный университет имени Ярослава Мудрого, 173000, Россия, г. Великий Новгород, ул. Лазаревская, дом 11

Аннотация. В статье предлагается подход к интроспекции виртуальных машин с использованием двоичного интерфейса приложений. Основная цель метода - получать информацию о работе системы, имея минимальные знания об ее внутреннем устройстве. Наша система основана на эмуляторе QEMU и имеет модульное строение, единицей в котором является плагин.

Существующие подходы (RTKDSM, DECAF) получают данные из операционной системы с помощью структур ядра. Эти инструменты вынуждены хранить большое количество профилей с данными, потому что все адреса и смещения в структурах меняются от версии к версии. Мы предлагаем использовать редко изменяющиеся части двоичного интерфейса приложений, такие как соглашения о вызовах и номера и параметры системных вызовов. Основная идея метода - перехватывать системные функции и считывать параметры и возвращаемые значения.

Для осуществления системного вызова у процессора есть специальная инструкция. Расширив возможности QEMU механизмом инструментирования, мы имеем возможность отслеживать каждую выполняющуюся инструкцию и отфильтровывать нужную. При возникновении системного вызова мы передаем управление в детектор системных вызовов, который проверяет номер произошедшего вызова и, в соответствии с ним, принимает решение какому плагину перенаправить это задание.

В механизме перехвата системных вызовов важно не только определить что вызов произошел, но и корректно определить его завершение, чтобы считать значения выходных параметров и возвращаемое значение. Для окончания системного вызова тоже есть специальные инструкции, но нам так же нужно верно сопоставить начало вызова с его концом, для чего мы определяем текущий контекст.

Таким образом мы реализовали мониторинг файловых операций, процессов и создали прототип монитора API функций.

Мы планируем расширить набор плагинов для анализа и мониторинга. Наша система будет извлекать информацию о загруженных модулях, приложениях, а также отладочную информацию.

Ключевые слова: интроспекция; виртуальные машины; динамический анализ; системные вызовы.

DOI: 10.15514/ISPRAS-2015-27(6)-11

Для цитирования: Фурсова Н.И., Довгалюк П.М., Васильев И.А. Использование ABI для интроспекции виртуальных машин. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 159-168. DOI: 10.15514/ISPRAS-2015-27(6)-11.

1. Введение

Динамический анализ - это важная технология для исследования программного обеспечения (ПО). Он используется для профилирования, анализа вредоносного кода, обнаружения вторжений, тестирования ПО и для многого другого.

Полносистемный анализ дает представление о всей системе. Он может помочь анализировать деятельность ядра ОС и взаимодействие между процессами, коммуникацию с аппаратным обеспечением или определить поведение вредоносного кода без влияния на работу системы.

Для реализации анализа мы используем интроспекцию. Интроспекция - это извлечение данных из операционной системы, которые она использует для своей работы и которые скрыты от пользователя. Такими данными могут быть идентификаторы процессов и потоков, значения переменных, содержимое памяти. Большое количество этих данных сосредоточены в структурах ядра системы.

Основная идея нашего метода - использование минимальных знаний о системе, которые не включают структуры ядра. Мы будем получать информацию с разных уровней работы системы, используя лишь данные с открытой структурой. Наш подход предполагает модульное строение, единицей в котором является плагин, выполняющий определенные функции.

Наша работа основана на мультиплатформенном симуляторе QEMU [1]. Мы расширили QEMU новой функциональностью, позволяющую загружать внешние плагины и производить динамическое инструментирование. Мы так же создали несколько плагинов для мониторинга системных вызовов, файловых операций и процессов.

2. Обзор существующих подходов

Рассмотрим несколько исследований на тему инструментирования и полносистемного анализа.

Real-time kernel data structure monitoring (RTKDSM) использует возможности анализа Volatility, является фреймворком с открытым исходным кодом, упрощает и автоматизирует анализ состояний выполняющейся виртуальной машины [3]. Система RTKDSM выполняет мониторинг состояния операционной системы в виртуальной машине в реальном времени. Она использует Xen как монитор виртуальной машины и некоторые плагины

Volatility. Volatility специализируется на анализе дампов памяти. RTKDSM анализирует память гостевой системы напрямую, не используя дампы.

Основанная на дампах архитектура Volatility не эффективна в режиме мониторинга, поскольку нужно анализировать целый дамп каждый раз, как только понадобится информация. Таким образом, RTKDSM использует Volatility только для поиска структур данных исследуемых операционных систем. После того как адреса структур получены, RTKDSM использует собственный агент мониторинга, который отслеживает изменения в этих структурах.

DECAF это платформо-независимый полносистемный фреймворк для динамического анализа [2]. DECAF расшифровывается как Dynamic Executable Code Analysis Framework. Он разработан на основе QEMU и предоставляет интерфейс для программирования анализирующих плагинов. DECAF восстанавливает семантику уровня операционной системы с помощью интроспекции. Восстановление происходит, когда такой запрос приходит с уровня аппаратного обеспечения.

Инструмент RTKDSM основан на Xen, что ограничивает его работу архитектурой x86. DECAF является мощным средством динамического анализа, имеет модульную структуру и открытый исходный код, что позволяет пользователям дополнять его нужными функциями, однако имеет некоторые недостатки. Во-первых, DECAF основан на QEMU версии 1.0, что не позволяет запускать новые операционные системы, такие как Windows 7-8. Во-вторых, метод получения данных у этого инструмента - исследование структур ядра системы, из-за чего ему необходимо генерировать и хранить профили для каждой версии исследуемой операционной системы.

3. Поход и уникальность

Мы предлагаем новый подход к интроспекции, который не требует модификаций гостевой операционной системы и приложений и сделает зависимые от операционной системы части минимальными. Например, DECAF вынужден иметь профиль на каждую версию операционной системы с указанием всех адресов структур и смещений в этих структурах, что не очень удобно, потому что адреса и смещения отличаются для разных версий и сборок ядра ОС.

Мы предлагаем использовать для извлечения данных из операционной системы некоторые части ABI (Application Binary Interface). ABI - это интерфейс, описывающий взаимодействие между операционной системой, библиотеками и приложениями. Как правило, в ABI входят такие сведения как соглашения о вызовах, формат исполняемых и библиотечных файлов, количество и формат системных вызовов и другое. ABI редко подвергается изменениям что позволяет получать данные, используя минимальное количество параметров. Это свойство важно, если речь идет о встроенных системах, в которые невозможно загрузить приложение, которое считает из

системы все необходимые смещения (таким образом работает DECAF). Для работы нашего метода используются такие части ABI как соглашения о вызовах и системные вызовы (их номера и параметры).

Одной из важных составляющих нашей системы является перехватчик системных вызовов. Системный вызов это запрос из операционной системы к ядру. Для этих запросов обычно предусмотрены специальные инструкции (например, syscall для x86/64 или svc #0 для ARM). Эти инструкции переводят процессор в режим ядра и выполняют соответствующий системному вызову код. Перехватчик системных вызовов это плагин, процесс работы которого представлен на рисунке 1.

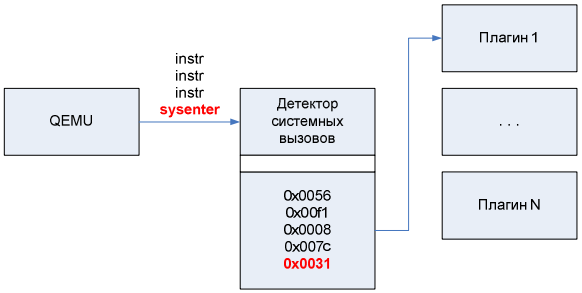

Рис. 1. Перехватчик системных вызовов.

С помощью перехвата системных вызовов мы имеем возможность отслеживать файловые операции, создание новых процессов и потоков, создание объектов ядра, операций отображения в память и другие.

Для выполнения требований о минимальном вмешательстве мы используем виртуальную машину и исследуем ее память. Также используются зависимые от ABI части кода для преобразования платформо-зависимых данных в платформо-независимое представление.

Мы используем наш подход для реализации алгоритмов анализа. Каждый алгоритм представляется в виде плагина. В итоге мы получаем систему с модульным строением, где каждая единица отвечает за один из видов анализа. Плагины могут работать на разных уровнях, таких как уровень операционной системы, уровень аппаратного обеспечения. Информация извлекается из исполняемого кода.

Плагины на разных уровнях взаимодействуют посредством сообщений. Сообщения соответствуют событиям, которые обозначают внешние коммуникации или изменения в состоянии виртуальной машины.

Аппаратный уровень включает события и данные, генерируемые симулятором, например, может быть вызвана функция обратного вызова, когда произошло прерывание, трансляция инструкции, выполнение инструкции, получение сетевого пакета, чтение/запись ячеек памяти и так далее. Эти события могут быть использованы для построения плагинов,

которые исследуют системные вызовы, адресные пространства и контекст исполнения.

Следующий уровень включает плагины для мониторинга файлов, процессов, потоков и других объектов операционной системы. Файловые операции могут быть исследованы путем перехвата соответствующего системного вызова, который рассматривался на предыдущем уровне. Создание процессов может исследоваться таким же способом.

Уровень приложений содержит плагины, которые исследуют модули, приложения и другие объекты уровня пользователя. Плагины уровня исходных кодов извлекают символьную и отладочную информацию для предоставления понятной пользователю информации об отладке и анализе приложения.

3.1 Мониторинг файловых операций

Мы создали плагин для мониторинга файлов. Этот плагин не зависит от гостевой операционной системы и гостевого аппаратного обеспечения.

Так как Windows и Linux имеют разные наборы системных вызовов, мы создаем плагин для перехвата системных вызовов для каждой операционной системы.

Для анализа файловых операций перехватываются следующие функции: NtCreateFile, NtOpenFile, NtReadFile, NtWriteFile, NtClose в Windows и creat, open, read, write, close в Linux.

Когда плагин обнаруживает платформо-зависимый системный вызов, он преобразовывает параметры и возвращает данные системного вызова в платформо-независимой форме.

Мы извлекаем следующие параметры операций: Create/Open file (возвращается хэндл файла, имя файла, тип доступа), Close file (хэндл файла), Read/Write file (хэндл файла, адрес буфера, количество прочитанных/записанный байт). Пример полученного абстрактного журнала представлен на рисунке 2.

```
open(name \SystemRoot\bootstat.dat, mode READ | WRITE,
      handle 0x1c)
read(handle 0x1c, buffer 0x15f88c, length 0x4)
write(handle 0x1c, buffer 0x15f8cb, length 0x1)
```

Рис. 2.Фрагмент абстрактного журнала.

Так же для каждой операционной системы записывается подробный лог всех вызывавшихся функций с полным набором параметров. Платформо-независимый вариант нужен для реализации единого механизма анализа лога, работающего со всеми операционными системами.

Плагин для системных вызовов использует две функции обратного вызова для обработки системного вызова. Первая функция обратного вызова требуется,

чтобы проверить какой системный вызов произошел и соответствует ли он нужному. Вторая функция обратного вызова выполняется, когда системная функция завершилась. Он читает возвращаемые значения и выходные параметры функций.

Механизм вызова функции обратного вызова по завершению выполнения функции с целью получения возвращаемых значений является одной из задач нашей работы. Для получения возвращаемого значения системного вызова мы вызываем функцию обратного вызова после выполнения системной функции. Для выполнения гостевого кода QEMU использует динамическую трансляцию. Мы не можем просто вставить любой код после выполнения инструкции в этом же блоке трансляции, потому что выполнение текущего блока закончится и начнется выполнение следующего блока, что лишит нас возможности отследить завершение системного вызова.

Для решения этой задачи мы используем контекст выполнения. Функции обратного вызова вызываются перед каждой инструкцией, и мы можем запомнить некоторые данные на старте системного вызова и затем сверять их с текущими данными на выходе. Для определения начала и окончания выполнения системного вызова в архитектуре x86 используются регистры esp и cr3.

3.2 Мониторинг процессов

Системные вызовы для процессов разные у разных операционных систем, поэтому большинство их параметров мы не можем унифицировать. Мы трассируем наиболее важные данные, доступные во всех реализациях системных вызовов.

Процессы в Linux могут быть созданы с помощью функций fork или clone. Windows использует NtCreateProcess, которая сама вызывает NtCreateThread. Поскольку функции для создания процессов разные, то привести их к общему виду является нетривиальной задачей. На данном этапе для каждой операционной системы свой плагин и абстрактный журнал для процессов не пишется.

Одним из очень важных параметров при создании процессов и работе с ними является идентификатор процесса. В Windows на уровне системных вызовов в большинстве функций идентификатор заменен хэндлом процесса, в Linux же возвращается настоящий идентификатор процесса. Первой идеей было использовать хэндл вместо идентификатора, однако выяснилось, что хэндл ненадежный источник: он может закрыться, а процесс продолжит жить или он может быть продублирован, поэтому было решено найти идентификатор процесса и для Windows. Функция NtCreateThread заполняет структуру ClientID, которая содержит идентификатор процесса и потока.

Для того, чтобы использовать идентификаторы для анализа системы и приложений необходимо связать их с адресным пространством. На данный момент мы работаем с архитектурой x86 и используем для этой цели регистр

cr3. Процесс создается из контекста другого процесса, поэтому на первом этапе мы сможем связать только cr3 создающего процесса с идентификатором создаваемого. Эта схема представлена на рисунке 3.

Рис. 3.Связь cr3 и идентификатора процесса.

Для того чтобы решить проблему с контекстом было решено использовать системный вызов NtQueryProcessInfo, который тоже при определенных условиях может запрашивать структуру ClientID. Если эта системная функция вызывается для текущего процесса, то мы можем корректно соотнести адресное пространство с идентификатором процесса. На данный момент непонятно, достаточно ли этих данных, поэтому мы рассмотрели возможность использования альтернативных путей получения этой информации, например, встраиваемые системные вызовы. Это значит, что мы будем вызывать некоторые системные вызовы в тот момент, когда необходимо будет получить информацию о процессе, например, NtQueryProcessInfo/getpid. Однако такой подход возможен только с использованием записи/воспроизведения [4], потому что встраивания изменят состояние системы.

3.3 Мониторинг API функций

Следующим источником получения данных являются API функции. Исследуя их выполнение, мы можем получать доступ к более высокоуровневой информации.

Для того, чтобы получить доступ к этим функциям нужно определить базовый адрес, по которому загрузится dll, а также необходимо знать смещения в файле, по которым расположены адреса функций из библиотеки.

Мы пробовали решить эту задачу на примере библиотеки kernel32.dll с помощью системных вызовов. В загрузке dll файлов обычно принимают участие три системных вызова: NtOpenFile, NtCreateSection, NtMapViewOfSection. Выходным параметром последней системной функции является исковый базовый адрес. Теперь, зная адрес, можно определить смещения интересующих функций и, используя выражение "базовый адрес + смещение" получить адрес функции. Принцип работы с адресами может быть такой же, как с системными вызовами, только вместо опкодов инструкций рассматривать адреса функций и, при переходе на заданный адрес, передавать управление в нужный плагин.

4. Заключение

В статье описан подход для интроспекции виртуальных машин. Мы предлагаем анализировать операционные системы и приложения в виртуальной машине с минимальными вмешательствами в их работу. Для

этого мы используем некоторые аспекты ABI, а именно коды системных вызовов и соглашения о передаче их параметров. Мы создали основу нашей системы - перехватчик системных вызовов для файловых операций и процессов. Перехватчик системных вызовов анализирует инструкции симулятора и определяет в потоке системные вызовы. Монитор файлов может быть использован для трассировки файловых операций в операционных системах Windows и Linux.

Мы планируем расширить набор плагинов для анализа и мониторинга. Плагины должны предоставлять возможности для интроспекции для операционных систем. Наша система будет извлекать информацию о загруженных модулях, приложениях, вызовах API функций, а также отладочную информацию.

Одной из возможностей улучшения инструмента анализа является анализ оффлайн, основанный на записи сценария работы виртуальной машины. У нас уже есть реализация записи/воспроизведения в QEMU [4]. Анализ будет выполняться в процессе воспроизведения и не окажет никакого влияния на систему, потому что все входные данные записаны в файле журнала. Соединив запись/воспроизведение с подходом к интроспекции, мы получим мощный инструмент анализа без оказания влияния на работу системы.

Список литературы

[1]. F. Bellard, QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41-41, Berkeley, CA, USA, 2005. USENIX Association.

[2]. A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. 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, pages 248-258, New York, NY, USA, 2014. ACM.

[3]. J. Hizver, T-c Chiueh. Real-time Deep Virtual Machine Introspection and Its Applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, pages = 3-14, York, NY, USA, 2014. ACM.

[4]. P. Dovgalyuk, Pavel. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pages 553-556, CSMR '12, Washington, DC, USA, 2012. IEEE Computer Society.

Using ABI for Virtual Machines Introspection

N.I. Fursova <natalia.fursova@ispras.ru>
P.M. Dovgalyuk <pavel.dovgaluk@ispras.ru>
I.A. Vasiliev <ivan.vasiliev@ispras.ru>
Yaroslav-the-Wise Novgorod State University,
173000, Russia, Velikiy Novgorod, Lasarevskaya street, 11

Abstract. The paper proposes an approach to introspection of virtual machines using the applications binary interface. The purpose of the method is to get information about the system, while having a minimum knowledge about its internal structure. Our system is based on QEMU emulator and has a modular structure.
Existing approaches (RTKDSM, DECAF) receive data from the operating system using the kernel structures. Those instruments have to store a large number of data profiles, because all addresses and offsets in the kernel structures vary from version to version. We offer the use of the rarely changing application binary interfaces, such as calling conventions and the numbers and parameters of system calls. The idea of the method is to intercept system functions and read parameters and return values.
Processor uses a special instruction to implement a system call. We expand QEMU with instrumentation engine, so we are able to monitor each executing instruction and to filter desired ones. In the event of a system call, we pass the control to the detector of system calls, that checks the number of occurred call and according to it decides to which plugin the job should be redirected to. In the mechanism of system calls interception, it is important not only to determine that the call occurred, but also to correctly determine its completion. That is needed to obtain the values of output parameters and return values.
To determine the end of the system call, the system also has special instructions, but we need to collate the beginning of the call to its end correctly. And to do so we are using the current context.
Thus, we have implemented monitoring of file operations and processes, and created a prototype of API functions monitor. We plan to expand the set of plugins for analysis and monitoring.

Keywords: introspection; virtual machines; dynamic analysis; system calls.

DOI: 10.15514/ISPRAS-2015-27(6)-11

For citation: Fursova N.I., Dovgalyuk P.M., Vasiliev I.A. Using ABI for Virtual Machines Introspection**.** Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 159-168 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-11

References

[1]. F. Bellard, QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41-41, Berkeley, CA, USA, 2005. USENIX Association.

[2]. A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. 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, pages 248-258, New York, NY, USA, 2014. ACM.

[3]. J. Hizver, T-c Chiueh. Real-time Deep Virtual Machine Introspection and Its Applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, pages = 3-14, York, NY, USA, 2014. ACM.

[4]. P. Dovgalyuk, Pavel. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pages 553-556, CSMR '12, Washington, DC, USA, 2012. IEEE Computer Society.