**Применение технологий виртуализации в процессе фаззинга программных компонентов**

Современные информационно-вычислительные системы представляют собой сложную совокупность гетерогенных функциональных блоков. Программные компоненты, входящие в их состав, построены с применением разнообразных технологических решений. Рассматривая существующие технические системы следует отметить высокую степень внедрения в них различного программного обеспечения. Ошибки и уязвимости имеющиеся в программных системах зачастую приводят к выходу из строя компьютерной системы, утечке конфиденциальной информации и т.п. негативным последствиям.

Одним из механизмов поиска уязвимостей в программных решениях является фаззинг. Основной задачей программных решений и отдельных модулей является обработка некоторой информации, а так же управление другими программными или аппаратными компонентами. Информационные потоки являются неотъемлемой их частью. Практически любой программный модуль, выполняющийся в некоторой аппаратной системе, имеет некоторые точки входа, в которых он ожидает получить определенные данные. Общей сутью классического подхода к фаззингу является внедрение в такие точки входа информации в процесс некоторых, потенциально способных вызвать нарушения в работе программного компонента, данных. Такие операции выполняются последовательно до тех пор, пока не возникнет исключительная ситуация. В наиболее типичных случаях инструменты фаззинга, например, генерируют файлы неверного формата, а затем передают их некоторому тестируемому программному решению. В случае выявления факта аварийного завершения работы программы или возникновения необрабатываемых в пользовательском коде, но, возможно, не фатальных для приложения исключений данные средства уведомляют пользователя о наличии уязвимости.

В ряде случаев подобного типа подходы являются затруднительными. Например, тестирование корректности обработки сообщений, переданных по сети с применением нестандартных закрытых шифрованных протоколов, пользовательским приложением будет затруднительным, поскольку для передачи сгенерированных потенциально опасных данных потребуется знать протокол и применяемые в нем механизмы шифрования.

Для решения данного вопроса активно применяется фаззинг оперативной памяти. В основе такого подхода лежит идея внедрения потенциально опасных данных напрямую в адресное пространство тестируемого процесса, а затем выполнение исследуемого блока кода. Т.о. фаззинг оперативной памяти нацелен на исследование отдельных блоков кода и тестирование логики их выполнения.

В настоящее время наиболее популярны следующие подходы к фаззингу оперативной памяти [1]:

1. Manipulation Loop Insertion (MLI) [1]. В данном подходе исследователь в начале выявляет интересующий его алгоритм вручную при помощи реверс-инжиниринга. Затем в адресное пространства процесса вносится алгоритм генерации потенциально опасных данных. Далее в начало и конец исследуемого алгоритма внедряются команды безусловного перехода на алгоритм генерации потенциально опасных данных. Данный подход не контролирует состояние глобальных переменных программы и внешних объектов системы, которые могли быть приведены в определенное состояние в ходе работы алгоритма. Таким образом, глобальные переменные и внешние объекты могут повлиять на качество тестирования.
2. Snapshot Restoration Mutation (SRM) [1]. Данный подход так же как и MLI требует определения исследуемого компонента вручную посредством реверс-инжиниринга. Однако в отличие от предыдущего метода состояние глобальных переменных контролируется путем создания снимков памяти. Так, в начале работы инструмент анализа создает снимок памяти процесса, затем генерирует потенциально опасный набор данных, внедряет его в исследуемый алгоритм. В начале новой итерации цикла анализа процесс возвращается в предыдущее состояние благодаря созданному ранее снимку памяти. Таким образом, данный подход несколько более совершенен и позволяет контролировать состояние глобальных переменных. Однако, важно понимать, что данный подход не позволяет контролировать внешние объекты. Так, если, например, в ходе работы алгоритма был открыт некоторый файл, из него произведено чтение последней записи, затем в него была произведена новая запись, а после файл был закрыт, то восстановление состояния процесса по сделанному ранее снимку памяти не позволит восстановить содержимое файла и может остаться незамеченным. Во время новой операции чтения в таком случае будет прочитано значение, записанное во время предыдущей итерации.

Серьезным недостатком фаззинга является недостаточный уровень контроля причины возникновения исключения. Так, исключение может возникнуть в ходе очередной итерации, но быть вызвано некоторыми внешними объектами. Более того, проведя исследования современных программных решений важно отметить распространение управляемых решений на базе технологий Microsoft .NET Framework. Программное обеспечение, построенное с применением этой технологии может содержать в себе Managed (управляемый – байт-код), не управляемый (Native – скомпилированный под конкретную аппаратную платформу) и Mixed (смешанный в рамках одной сборки) код. Методы MLI и SRM в общем случае могут успешно применяться для Native программных решений. Фаззинг Managed и Mixed программных решений с применением данных технологий в значительной степени осложнен. В частности из за того, что даже однопоточное (с точки зрения прикладного программиста на языках платформы Microsoft .NET Framework) является не однопоточным (рис. 1.). Программные решения предложенные в [1] и построенные на методах MLI и SRM приводят в неработоспособное состояние Managed и Mixed приложения.

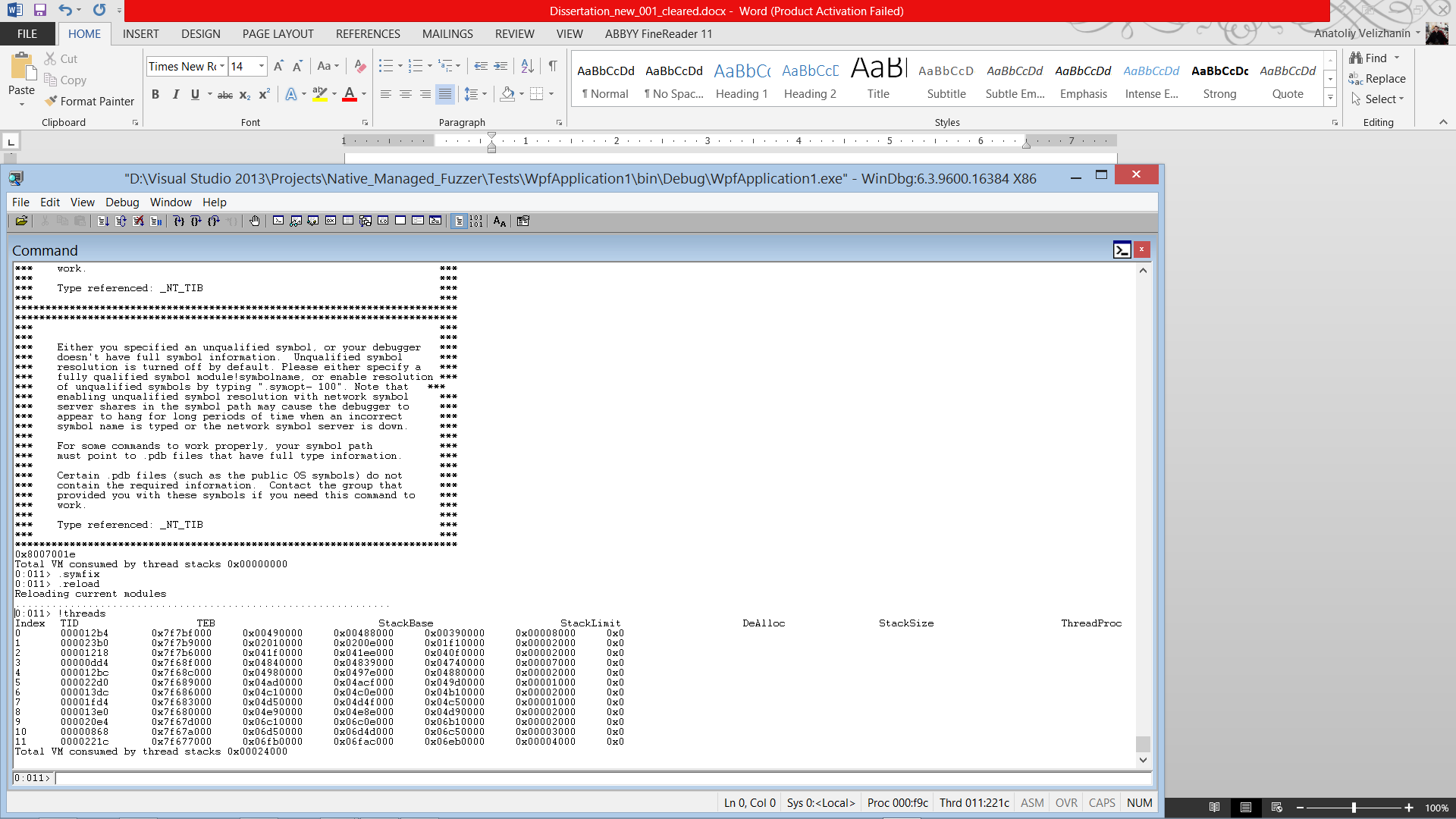


Рис. 8. Иллюстрация потоков, созданных автоматически в ходе запуска однопоточного (с точки зрения прикладного программиста) Managed приложения

Для решения задачи фаззинга современных программных решений предлагается применение технологий виртуализации, что позволит эмулировать среду в рамках операционной системы. Эмуляция целой гостевой системы позволяет не только проводить фаззинг Managed и Mixed программных решений, получивших большую популярность в настоящее время, но и позволяют минимизировать влияние внешних объектов на последующие итерации фаззинга.

Приведем предложенный алгоритм фаззинга оперативной памяти с применением технологии виртуализации реализован в программном решении RAMFuzzer.

1. Базовая подготовка системы Fuzzing-а. На данном этапе исследователь создает виртуальную машину (поддерживается система VirtualBox) на которой будет функционировать исследуемое программное обеспечение. Настройка отладчика WinDbg.
2. Подготовка пред-тестового состояния гостевой системы. На данном этапе необходимо запустить исследуемый программный модуль под отладчиком Microsoft WinDBG, установить Breakpoint перед вызовом интересующей нас с точки зрения исследования функции и вывести программу на данную точку останова (следующий шаг отладчика должен привести к вызову интересующей нас функции).
3. Создание базового снимка предварительно настроенной гостевой операционной системы. Данный снимок будет использоваться для восстановления состояния виртуальной машины на каждой итерации Fuzzing-а.
4. Необходимо настроить программное решение интегрированной среды Fuzzing-а на компьютерной системе хоста. В качестве набора настроек интерфейс взаимодействия с пользователем программы RAMFuzzer принимает
   1. Параметры виртуальной машины
      1. Файл менеджера виртуальной машины VirtualBox.
      2. Строка подключения к виртуальной машине.
      3. Имя виртуальной машины.
      4. Базовое имя снимка гостевой системы.
      5. Максимальное количество итераций.
      6. Проверка связи с виртуальной машиной.
      7. Откат виртуальной машины каждую итерацию Fuzzing-а.
   2. Настройки тестируемой программы
      1. Адрес точки старта Fuzzing-а. Данный адрес является адресом входа в интересующую нас функцию.
      2. Адрес точки выхода Fuzzing-а. Данный адрес соответствует точке выхода из интересующей нас функции. По достижению данной точки происходит переход на следующую итерацию тестирования программного модуля.
      3. Рабочая директория. Позволяет указать директорию, в которую будут сохранены базовые данные по каждой проведенной итерации Fuzzing-а.
      4. Исполняемый файл скрипта Fuzzing-а. Данная настройка хранит в себе полное имя исполняемого файла алгоритма Fuzzing-а. Поскольку интегрированная среда Fuzzing-а должна обладать максимальной гибкостью исследователь не может ограничиться некоторым набором шаблонов входных данных. Предусмотрев возможность указания исполняемого файла скрипта пользователем происходит не только передача ему части нагрузки, но и одновременно предоставляется гибкий интерфейс манипулирования входными данными.
5. Запуск процесса тестирования.

В настоящее время проводится совершенствование алгоритма и программного продукта. Одними из перспективных усовершенствованиями видится поддержка виртуализации на базе QEMU для эмуляции архитектур процессоров, отличных от системы хоста, а так же поддержка работы с отладчиком GDB для Unix-подобных операционных систем.

Применение технологии виртуализации так же сопряжено с рядом трудностей. В частности невысокая производительность систем виртуализации и низкая производительность аппаратной платформы могут привести к значительным временным затратам.

**Используемая литература**

[1] Fuzzing. Исследование уязвимостей методом грубой силы. Автор: Майкл Саттон, Адам Грин, Педрам Амини, Издательство: Символ-Плюс, ISBN 978-5-93286-147-9; 2009 г.