Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке

В.А. Падарян
<vartan@ispras.ru>
В.В. Каушан
<korpse@ispras.ru>
А.Н. Федотов
<fedotoff@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье рассматривается метод автоматизированного построения эксплойтов для уязвимости переполнения буфера на стеке и его применение к задаче оценки критичности ошибок в программном обеспечении. Метод, на основе динамического анализа и символьное выполнения, применяется к бинарным файлам программ без дополнительной отладочной информации. Описанный метод был реализован в виде инструмента для построения эксплойтов. Возможности инструмента были продемонстрированы на примере 8 уязвимостей в приложениях, работающих под управлением ОС Windows и Linux, 3 из которых не были исправлены на момент написания статьи.

Ключевые слова: классификация ошибок; эксплуатация уязвимостей; бинарный код; динамический анализ; символьное выполнение.

1. Введение

С развитием информационных технологий становятся все более значимыми как сама безопасность программ, так и средства ее обеспечивающие. Сложное программное обеспечение активно используется в работе критической инфраструктуры: контролирует движение транспорта, управляет в больницах различным медицинским оборудованием, регулирует работу электростанций. Рабочие сбои в функционировании такого ПО приводят к серьезным последствиям, а целенаправленная эксплуатация ошибок злоумышленниками способна привести к еще большему ущербу. Ошибки, использование которых может привести к намеренному нарушению целостности системы и нарушению её работы, называют уязвимостями. Многие крупные IT-компании (Microsoft, Google и т.д.) не только поддерживают исследования в области

поиска ошибок и уязвимостей ПО, но и внедряют самые передовые технологии в жизненный цикл разработки программ, с целью сокращения ошибок в выпускаемом ПО и снижения издержек, связанных с обнаружением и исправлением ошибок.

Искать ошибки и уязвимости можно как на уровне исходных текстов, так и анализируя бинарный код. Второй способ является более предпочтительным, так как абстракции языка высокого уровня скрывают особенности работы программы, важные для выявления ошибок и оценки их важности. Кроме того, исходные тексты зачастую недоступны, поэтому специалистам, оценивающим безопасность ПО, неизбежно приходится иметь дело с исполняемым (бинарным) кодом и применять соответствующие методы анализа [1]. В последние годы активно развивается подход к поиску ошибок на основе символьного выполнения.

Символьное выполнение было предложено в конце семидесятых годов двадцатого столетия для тестирования программного обеспечения [2]. Под символьным выполнением понимается процесс выполнения исследуемой программы, в которой конкретные значения переменных заменяются символьными значениями. Как правило, символьными значениями являются входные данные программы. Операции над символьными значениями порождают формулы, описывающие последовательность операций над символьными переменными и константами. Каждое условное ветвление, зависящее от символьных данных, добавляет в общую систему уравнение, описывающее прохождение управления по определенной ветке. Строящуюся систему уравнений называют предикатом пути, т.к. она описывает сценарий работы программы, в рамках которого была построена. Построенная система уравнений подается на вход программе-решателю, где символьные переменные выступают в качестве неизвестных величин в формулах. Результатом решения является конкретный набор значений для символьных переменных.

Первоначальная идея применения символьных вычислений была нацелена на улучшение покрытия программы тестами. Но в последнее время эту технику начали применять для направленного поиска определенного состояния программы. Перед вызовом решателя предикат пути дополняют уравнениями, описывающими искомое состояние программы, в рассматриваемом случае – это ситуация, когда срабатывает уязвимость. Ввиду большого количества классов уязвимостей и многочисленных факторов, влияющих на проявление той или иной уязвимости, попытки решить задачу формального описания уязвимостей на уровне бинарного кода предпринимались только для отдельных, частных случаев.

Уязвимость, как правило, обусловлена программной ошибкой. Однако не каждая программная ошибка порождает уязвимость. Современные средства фаззинга, применяемые в промышленном программировании, вырабатывают тысячи наборов входных данных, приводящих к аварийной остановке [3].

Актуален вопрос расстановки приоритетов для найденных ошибок. Целесообразно в первую очередь исправлять ошибки, которые могут быть проэксплуатированы. Наиболее опасными для пользователя и желаемыми для нарушителя являются ошибки, которые предоставляют возможность выполнить произвольный код.

В рамках данной работы предложен метод оценки найденных ошибок, основанный на символьном выполнении бинарного кода программ. Для заданного набора входных данных, приводящих к аварийной остановке программы, выполняется построение эксплойта (набора входных данных, эксплуатирующих уязвимость) для распространенного вида уязвимостей: переполнения буфера на стеке. Те ошибки, для которых удалось построить эксплойт, должны быть отнесены к категории критических и исправлены как можно скорее. Предложенный в работе метод автоматизируем; был разработан программный инструмент, реализующий этот метод. Он позволяет генерировать для соответствующих ошибок эксплойты, приводящие к выполнению кода полезной нагрузки, заданного пользователем.

Статья организована следующим образом. Во втором разделе рассмотрены методы, положенные в основу решения поставленной задачи. В третьем разделе представлены основные сведения об особенностях эксплуатации уязвимости переполнения буфера на стеке. В четвертом разделе описан сам метод, а отдельные аспекты его реализации и результаты практического применения приведены в пятом разделе. В последнем, шестом, разделе обсуждаются полученные результаты и дальнейшие направления исследований.

2. Используемые методы анализа

Анализ бинарного кода можно проводить с помощью методов статического и динамического анализа [1]. Символьное выполнение в рамках статического анализа ограничено применимо из-за высокой сложности получаемой системы уравнений, практический успех был заявлен только в рамках динамического [4] или смешанного анализа [5].

Представленные в данной статье исследования опираются на возможности среды анализа бинарного кода [6]. Основным предметом анализа являются трассы машинных инструкций, полученные в результате работы полносистемного эмулятора [7,8]. Трассы содержат состояния регистров, информацию о прерываниях и взаимодействии с периферийными устройствами, что позволяет восстанавливать статико-динамическое представление для всех работавших в системе программ и эффективно исследовать его свойства. Основным назначением среды анализа является автоматизация метода выделения алгоритмов из бинарного кода [9] и повышение уровня представления этих выделенных алгоритмов.

Поскольку известен набор входных данных, приводящий к аварийному завершению программы, можно получить трассу выполнения, которая

содержит аварийное завершение программы. Для генерации эксплойта достаточно рассматривать только те инструкции, которые относятся к обработке входных данных с момента их получения до момента аварийного завершения программы. Для отбора таких инструкций используется алгоритм динамического слайсинга трассы, дополненный анализом помеченных данных.

Современные процессорные архитектуры содержат множество различных инструкций со сложной семантикой и побочными эффектами. Распространенным приемом, обеспечивающим поддержку различных архитектур, является использование промежуточного представления. В рамках разрабатываемого метода используется промежуточное представление Pivot [10], позволяющее унифицировано описывать операционную семантику инструкций различных архитектур. Промежуточное представление отвечает требованием SSA-формы, что позволяет значительно упростить анализ. Основные операторы Pivot, используемые при построении предиката пути, следующие.

- Оператор `NOP` – не имеет никакого эффекта.
- Оператор `INIT` – инициализирует локальную переменную константным значением.
- Оператор `APPLY` – применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.
- Оператор `BRANCH` – осуществляет передачу управления.
- Оператор `LOAD` – производит загрузку значения из одного из адресных пространств.
- Оператор `STORE` – записывает значение в одно из адресных пространств.

Для описания памяти и регистров в Pivot используется модель единообразных адресных пространств. С точки зрения этой модели все адресуемые операнды целевой архитектуры, с которыми может работать машина (регистры, память, порты ввода-вывода), размещаются в линейных адресных пространствах, обращение к ним использует пару (идентификатор пространства, смещение). Для учета побочных эффектов работы операций используется модельное слово состояния, которое аналогично регистру флагов в x86.

3. Эксплуатация уязвимости переполнения буфера на стеке

Рассматривается ситуация, когда размер данных, записываемых в буфер на стеке, превышает размеры этого буфера. На рис. 1А показан кадр стека некоторой функции, в которой происходит переполнение буфера. Традиционно, в архитектуре x86 стек растет от старших адресов к младшим

(на рисунке – сверху вниз). Параметры размещаются на стеке в обратном порядке, далее вызов функции приводит к размещению адреса возврата на стеке, после чего функция может сохранить старое значение регистра `ebp` и выделить память для локальных переменных (в том числе для буфера). Данные в буфер записываются в порядке возрастания адресов (на рисунке – снизу вверх).

Если в буфер записать больше данных, чем размер этого буфера, это приведет к перезаписи ячеек выше по стеку, в том числе может привести к перезаписи старого значения `ebp`, адреса возврата и аргументов функции. Если злоумышленник может контролировать значения, записываемые в буфер, он может добиться того, что после переполнения адрес возврата будет содержать указатель на произвольный код, который может быть сформирован самим злоумышленником. Выполнение этого кода может привести к серьезным последствиям, в том числе к компрометации данных приложения, пользователя или операционной системы. Обычно, в качестве такого кода используется код, приводящий к вызову оболочки, называемый шелл-кодом. В реальности, использование такого кода настолько популярно, что часто любой код полезной нагрузки называют шелл-кодом. Злоумышленник может поместить код как ниже адреса возврата, так и выше него (рис. 1Б).

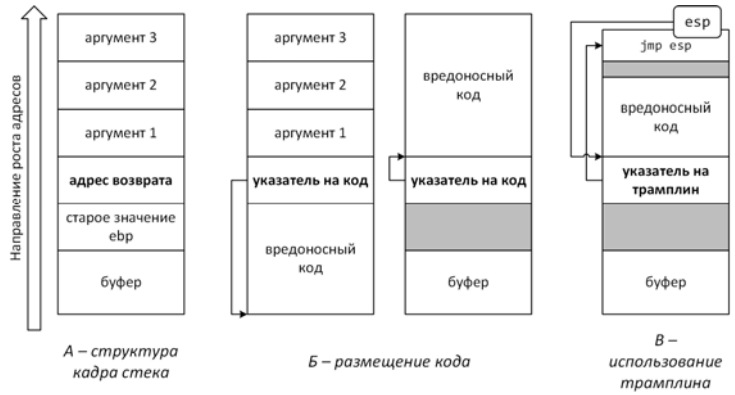

Рис. 1. Организация стека и способы размещения на нем внедряемого кода.

Следует отметить, что область памяти, выделенная под стек, может быть защищена от выполнения кода, и подобная попытка выполнения вредоносного кода может привести к аварийному завершению программы. В этом случае можно воспользоваться методикой Return-oriented programming [11], позволяющей составлять полезную нагрузку из уже имеющихся участков кода. Помимо того, при использовании механизма рандомизации адресного пространства, вредоносный код будет загружаться по разным адресам для разных попыток эксплуатации уязвимости, что не позволяет заранее

определить значение для записи на место адреса возврата. Но достаточно часто в момент выхода из функции один из регистров указывает на некоторую область памяти на стеке. Если эта область памяти доступна для размещения кода, то можно передать управление на этот код с помощью инструкции вида `jmp <reg>` или `call <reg>`, расположенной по заранее известному адресу. Инструкции такого вида в данной работе называются «трамплинами». При использовании трамплинов, после возврата из функции управление будет передано на трамплин, а оттуда – в размещенный на стеке исполняемый код (рис. 1В). Следует отметить, что использование трамплинов полезно не только в случае рандомизации, но и когда начальный адрес размещаемого шелл-кода содержит нулевой байт. Довольно часто переполнение буфера происходит во время копирования нуль-терминированной строки, которая не может содержать нулевые символы, а, следовательно, не позволяет переписать ячейку адреса возврата нужным значением.

4. Схема работы

Процесс генерации эксплойта разделен на четыре последовательных этапа, один из них является необязательным (рис. 2).

Сначала происходит выделение подтрассы, состоящей только из инструкций обрабатывающих входные данные. Для этого используется алгоритм слайсинга, а также информация о точке получения входных данных и точке аварийного завершения. Для полученной подтрассы строится предикат пути. В качестве опционального шага проводится поиск трамплинов. После этого происходит собственно построение эксплойта для заданного пользователем шелл-кода.

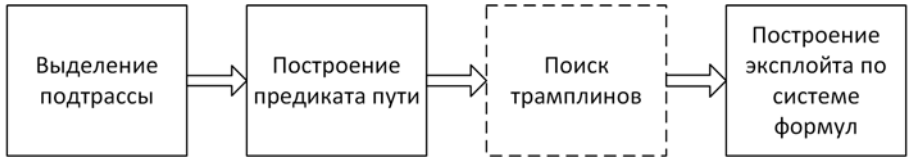

Рис. 2. Декомпозиция метода на четыре этапа

4.1 Выделение подтрассы

Выделение подтрассы делается для ограничения числа рассматриваемых машинных команд. В подтрассу отбираются только те команды, которые прямо или косвенно обрабатывают введенные в программу данные; ее построение выполняется алгоритмом слайсинга трассы [12]. Для работы слайсинга необходимы следующие данные: диапазон шагов трассы, на котором алгоритм отслеживает данные и начальный набор отслеживаемых данных. Начальному шагу соответствует точка получения входных данных, а конечному – точка аварийного завершения.

Начальный шаг и буфер с входными данными

Искать буфер с входными данными и шаг трассы, на котором буфер оказывается заполненным, аналитик может одним из нескольких способов.

Многие программы для получения данных используют библиотечные функции. Зная источник данных, например, сеть, файл и т.п., аналитик может найти в трассе вызовы соответствующих функций: например, для получения данных из сети чаще всего используется функция `recv`, а для чтения данных из файла – `ReadFile`. Входные данные также могут передаваться в качестве параметров командной строки, в таком случае, аналитику нужно найти вызов функции `main`.

Поиск шага аварийного завершения и ячейки перезаписи адреса возврата

Данная задача разделяется на две подзадачи:

- поиск ячейки, в которой хранится адрес возврата, перезаписываемый во время переполнения буфера;
- поиск шага трассы, на котором произойдет переход по перезаписанному адресу.

Для поиска ячейки используется алгоритм слайсинга, критерием является найденный буфер с входными данными. Для каждой отобранной инструкции адрес операнда-приемника сравнивается с адресами ячеек, хранящих адреса возврата функций для текущего стека вызовов. Если эти адреса пересекаются, это означает, что произошла перезапись адреса возврата одной из функций в стеке вызовов. В этом случае ячейка памяти, хранящая перезаписанный адрес возврата, считается искомой ячейкой.

Далее с помощью описанного ниже алгоритма выполняется поиск шагов трассы, в которых, вероятно, произошло аварийное завершение. Для каждого из этих шагов сравнивается значение регистра `ESP` с адресом ячейки, в которой хранится адрес возврата (полученной на предыдущем этапе). Если значения совпадают, то данный шаг считается шагом, на котором происходит аварийное завершение исследуемой программы.

Информация о возможных аварийных завершениях

Понятие аварийного завершения программы привязано к операционной системе. Так как предложенный метод рассчитан на работу в произвольном операционном окружении, без привязки к конкретной процессорной архитектуре, поиск аварийного завершения использует обобщенную модель процессора общего назначения и исходит только из найденных сценариев работы в рамках этой модели. Можно отметить, что аварийное завершение программы почти всегда происходит в результате исключительной ситуации, произошедшей в этой программе. В свою очередь, исключительная ситуация

приводит к возникновению прерывания процессора. Прерывания являются сущностями, не привязанными к конкретной платформе и архитектуре, поэтому работа алгоритма поиска аварийного завершения построена на анализе обработки прерываний. Необходимо различать прерывания, возникающие из-за исключительной ситуации в программе от прерываний, возникающих из-за работы устройств ввода/вывода или других событий процессора. Кроме того, так как трасса отражает выполнение всех программ в исследуемой системе, она содержит не только инструкции и исключительные ситуации, относящиеся к исследуемому приложению, но и для всех остальных приложений.

Используемый в данной работе алгоритм находит множество точек трассы, в которых, скорее всего, происходили исключительные ситуации. Так как аварийное завершение программы происходит в результате исключительной ситуации и при этом исключительная ситуация порождает прерывание, разумно искать точки аварийного завершения программы среди точек трассы, в которых возникают прерывания. Кроме того, исключительная ситуация нарушает естественный вход выполнения программы: вместо следующей инструкции после выхода из прерывания будет выполнен код обработчика исключений или завершение процесса. Это наблюдение позволяет значительно сократить множество точек трассы, рассматриваемых в дальнейшем.

Для каждого прерывания в трассе определяют адреса инструкций в точке входа в прерывание и в точке выхода из него. Далее рассматривается инструкция, выполняемая перед прерыванием. Здесь возможны два варианта: инструкции передачи управления и все остальные. Для инструкций передачи управления необходимо дополнительно вычислить адрес назначения: следующей должна выполниться инструкция по этому адресу (либо по адресу, следующему за инструкцией передачи управления в случае инструкций условного перехода). Для всех остальных инструкций сравнивается адрес после прерывания и адрес, следующий за выполненной инструкцией. Если они равны – выполнение не было нарушено. Аналогичные проверки проводятся для инструкций передачи управления, только сравнение идет с адресом назначения.

Результатом является множество точек трассы, в которых потенциально возникли исключительные ситуации. С помощью приведенного выше способа, среди этих точек перебором находится точка аварийного завершения исследуемой программы.

4.2 Построение предиката пути

Для построения предиката пути используется слайсинг трассы. Критерием слайсинга является уже найденный буфер с входными данными и шаг трассы, на котором происходит их получение. Просмотр трассы ограничен шагом, на котором происходит аварийное завершение исследуемой программы.

Результатом является слайс трассы и набор отслеживаемых ячеек на каждом шаге этого слайса. По этой информации строится предикат пути.

В данном случае, предикат пути представляет собой уравнения на языке SMT [13], описывающие преобразования, которые происходили с входными данными, на пути до определенной точки программы. Предикат пути необходим для того, чтобы описать ограничения на входные данные, полученные на пути к точке аварийного завершения.

В процессе построения предиката пути происходит двойная трансляция: сначала машинная инструкция транслируется в промежуточное представление, затем это представление переводится в соответствующие ему SMT-уравнения. Пространство памяти и регистров представляется в виде двух массивов битовых векторов. Так выглядит объявление массивов памяти и регистров на языке SMT:

```
(declare-const r_0 (Array (_ BitVec 16)  (_ BitVec 8)));
(declare-const v_0 (Array (_ BitVec 64)  (_ BitVec 8)));
```

Здесь размер адреса в пространстве регистров (r) составляет 16 бит, а в пространстве виртуальной памяти (v) – 64 бита. Гранулярность данных в обоих пространствах составляет 8 бит.

Процесс формирования уравнений предлагается рассмотреть на конкретном примере двух последовательных машинных команд `CMP BYTE PTR [80553450h], 00h` и `JB 806EE97Ch` архитектуры Intel x86. На рис. 3 представлен результат бинарной трансляции.

Трансляция Pivot-инструкций происходит последовательно, начиная с первой. Оператор `INIT` транслируется в константное выражение SMT. Оператор `APPLY` в эквивалентную операцию SMT-решателя. Например, для второго оператора `APPLY` инструкции `CMP` будет сгенерировано следующее выражение:

```
((_ sign_extend 64) #x80553450)
```

Оператор `Load` производит загрузку значения элемента адресного пространства (ячейка памяти, регистр) в локальную переменную. Необходимо определить, является ли элемент отслеживаемым или нет. Для этого происходит поиск элемента в наборе отслеживаемых элементов на момент отбора инструкции.

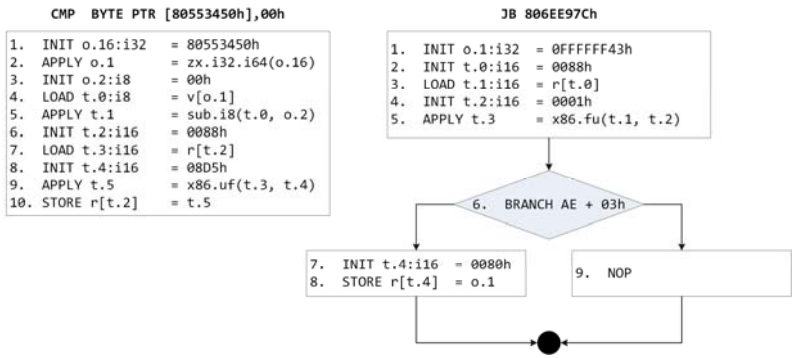

Рис. 3. Трансляция инструкций CMP и JB архитектуры Intel x86

Если элемент присутствует в наборе, то этот элемент считается символьным, в противном случае подставляется его конкретное значение. Для получения конкретных значений используется алгоритм восстановления буфера. Существуют ситуации, когда восстановить значение не удалось, в таком случае, элемент считается символьным. Пусть байт памяти по адресу `0x80553450` является символьным, тогда для оператора `Load` будет сгенерировано следующее выражение:

```
(select v_0 ((_ sign_extend 64)  #x80553450))
```

В итоге, для пятого оператора `APPLY` будет получено:

```
(bvsub (select v_0 ((_ sign_extend 64)  #x80553450)) #x00)
```

Инструкции с шестой по десятую не обрабатываются, поскольку они отвечают за обновление регистра флагов. Для того чтобы не создавать избыточность в уравнениях, происходит ленивое вычисление флагов. Результат трансляции машинной инструкции `CMP` – это выражение для пятого оператора и некоторая дополнительная информация необходимая для выставления флагов.

Теперь рассмотрим формирование уравнений для инструкции `JB 806EE97Ch`. Инструкции с 2-ой по 5-ую не обрабатываются, т.к. регистр флагов `r:[0x88]` не является символьным. Инструкции с номерами 1,7,8,9 не обрабатываются, т.к. регистр счетчика команд `r:[0x80]` также не является символьным. Таким образом, обрабатывается единственная шестая инструкция `Branch`.

Для условия оператора `Branch` выполняется вычисление нужных флагов и загрузка их на модельное слово состояния. В данном случае условие – `AE` (больше или равно для беззнаковых чисел). Вычисляется флаг `CF`, который загружается в модельное слово состояния. Затем составляется уравнение на выполнение инструкции `Branch`. В данном случае необходимо, чтобы значение флага `CF` было равно нулю. Затем добавляется уравнение, отражающее выполнение или невыполнение машинной инструкции условного перехода в трассе. Пусть условный переход был выполнен в трассе. Тогда для выполнения перехода `JB`, нужно добавить уравнение вида `(CF=0) = false`,

тем самым указав, что инструкция `Branch AE` не выполняется. Построенное уравнение становится частью предиката пути.

4.3 Поиск трамплинов

Для поиска трамплинов необходимо указать модули, которые относятся к адресному пространству приложения, и будут размещены по заведомо известным адресам. В коде указанных модулей происходит поиск инструкций-трамплинов. Эти инструкции имеют вид `jmp Reg` или `call Reg`. Они передают управление по адресу, записанному в регистре. Сначала стоит определить области памяти, в которых потенциально может быть размещен шелл-код. Из набора отслеживаемых (помеченных) ячеек памяти в момент аварийного завершения программы, отбираются ячейки, образующие непрерывные буферы достаточного размера для размещения шелл-кода. Если такие ячейки нашлись, то далее отбираются регистры, значения которых указывают на эти ячейки. Если таких регистров нет, то поиск трамплинов на этом завершается. В случае успеха, для отобранных регистров происходит поиск инструкций-трамплинов. Код операции и операнда-регистра таких инструкций занимает два байта, которые должны быть найдены в любых исполняемых секциях модулей. Следует отметить, что требуемые байты могут быть расположены, в том числе, и на границе двух разных инструкций. С ростом объема просматриваемых секций вероятность найти, как минимум, один трамплин растет достаточно быстро: для 500Кб она достигает 0.999.

4.4 Описание эксплойта и решение системы уравнений

Для получения эксплойта к предикату пути необходимо добавить уравнения описывающие этот эксплойт. Уравнения для перехвата управления после переполнения буфера на стеке можно разделить на два вида:

- уравнения для размещения шелл-кода в области памяти, находящейся под контролем нарушителя;
- передача управления в область памяти, находящейся под контролем нарушителя.

Для областей памяти, являющихся отслеживаемыми на момент аварийного завершения программы, выберем такие, размер которых больше или равен размеру шелл-кода. Если таких областей нет, то считаем, что данная уязвимость не эксплуатируема, в противном случае составим следующее уравнение для одной области. Пусть шелл-код представляет собой строку `"ABCD"`, диапазон адресов: от `1000` до `1003` включительно. Уравнение имеет вид:

`v(1000,1) =0x41 ∧ v(1001,1)=0x42 ∧ v(1002,1)=0x43 ∧ v(1003,1) =0x44`, где ∧ - логическое "и".

Для описания передачи управления в область памяти, находящейся под контролем нарушителя, необходимо составить уравнение, описывающее то,

что в области памяти, где хранится адрес возврата из функции, будет храниться адрес шелл-кода.

Пусть `x` – это адрес, по которому хранится адрес возврата из функции , `l` – адрес шелл-кода или адрес трамплина. Тогда формула имеет вид: `v(x,1) = l[0] ∧ v(x+1,1) = l[1] ∧ v(x+2,1) = l[2] ∧ v(x+3,1)= l[3]`, где ∧ - логическое "и".

Объединяя уравнения размещения и передачи управления с предикатом пути, получим набор уравнений, которых достаточно, чтобы описать процесс формирования рабочего эксплойта. Далее уравнения подаются на вход решателю, и если система уравнений совместна ее решение является рабочим эксплойтом.

5. Реализация метода и результаты практического применения

Предложенный метод был реализован в виде модуля-расширения среды анализа бинарного кода, он использует такие ее возможности, как повышение уровня представления, модель процессора общего назначения, слайс трассы.

Для решения системы используется сторонний SMT-решатель, который был интегрирован в разработанный модуль-расширение. На сегодняшний день существует большое количество доступных решателей: MiniSat, OpenSMT, STP, Yices, Z3 и др. В разработанном инструменте используется SMT-решатель Z3, в силу следующих достоинств:

- инкрементальный подход к решению уравнений;
- поддержка большого количества типов данных, в том числе и машинных;
- доступно API на языке Си, которое позволяет работать с уравнениями в виде объектов в памяти, что гораздо эффективней, нежели работа с текстом;
- исходный код доступен по лицензии MSR-LA;
- высокая скорость работы по сравнению с другими решателями.

Разработанный инструмент был опробован на нескольких примерах. В качестве гостевых ОС использовались 32-битные версии Windows XP SP2, Windows XP SP3, Arch Linux (по состоянию на апрель 2014) и МСВС 3.0. Использовались приложения с известными уязвимостями, а также приложения из актуального дистрибутива Arch Linux, ошибки в которых были найдены с помощью фаззинга. Список анализируемых приложений приведен в табл. 1.

Табл. 1. Список анализируемых приложений.

Операционная система	Приложение	Уязвимость
Linux	corehttp 0.5.4	CVE:2007-4060

MCBC	libpng (konqueror)	CVE:2004-0597
Windows XP SP3	SuperPlayer 3500	EDB-ID:27041
Linux	iwconfig v26	CVE:2003-0947
Windows XP SP2	lhhtpd 0.1	CVE:2002-1549
Linux	get_driver (sysfsutils)	
Linux	mkfs.jfs (jfsutils)	
Linux	alsa_in (jack)	

Для фаззинга использовался метод, описанный в [14]. Фаззер запускает исследуемое приложение со всеми возможными однобуквенными параметрами командной строки (от `-a` до `-z` и от `-A` до `-Z`) и еще одним параметром длиной 6676 байт. Для 6607 установленных приложений фаззер получил 748 аварийных завершений для 42 различных приложений. Из этих 42 приложений были выбраны 3, распространяемые в рамках популярных пакетов программного обеспечения: `sysfsutils`, `jfsutils`, `jack`.

В табл. 2 приведены результаты работы алгоритма построения эксплойта: размер слайса обработки входных данных, размер блока входных данных, время генерации системы уравнений и время ее решения.

Табл. 2. Результаты работы алгоритма построения эксплойта.

Операционная система	Приложение	Размер слайса	Размер данных, байт	Время решения, с	Суммарное время работы, с
Linux	corehttp 0.5.4	18293	565	1024	1367
MCBC	libpng (konqueror)	2493	536	8	128
Windows XP SP3	SuperPlayer 3500	4855	594	<1	66
Linux	iwconfig v26	124	80	<1	7
Windows XP SP2	lhhtpd 0.1	20174	320	18	245
Linux	get_driver	152	272	2	41
Linux	mkfs.jfs	209	407	3	23
Linux	alsa_in	241	58	<1	40

Следует отметить, что для успешной эксплуатации уязвимости переполнения буфера на стеке с использованием современного дистрибутива ОС Linux были предприняты меры по отключению различных механизмов защиты.

- Отключена рандомизация адресного пространства.
- Исследуемые приложения компилированы с флагами `-fno-stack-protector` и `-U_FORTIFY_SOURCE`, что привело к отключению механизма защиты стека «канарейкой» и использования безопасных функций, предоставляемых компилятором `gcc`.
- С помощью утилиты `execstack` для исследуемых приложений было выставлено разрешение на выполнение кода на стеке.

Для других операционных систем (Windows XP и MCBC) дополнительные действия по отключению механизмов защиты не предпринимались. При построении эксплойтов для приложений из табл. 1, работающих под Windows XP, использовались трамплины из-за того, что адреса памяти, выделенные под стек, содержали нулевой байт, и поэтому адрес шелл-кода не мог быть напрямую записан в адрес возврата. Для приложений, работающих под Linux, эта проблема отсутствует, но использование трамплинов для обхода рандомизации не принесло результатов, так как единственные нерандомизированные участки кода принадлежали самим приложениям: из-за малого размера приложений алгоритм поиска трамплинов не дал результатов.

В список результатов не были включены приложения, для которых не удалось сгенерировать работающий эксплойт. В этих приложениях в кадре стека, помимо переполняемого буфера, были расположены другие переменные, перезапись которых приводила к преждевременному аварийному завершению без выполнения полезной нагрузки. Чаще всего, такие переменные содержали указатели на некоторую область памяти, после перезаписи таких переменных разыменование расположенных там указателей приводило к ошибке доступа к памяти. В других случаях перезаписывалась переменная, содержащая переменную цикла, что приводило к чтению данных по некорректному адресу. Для корректной работы эксплойта в этих случаях требуется перезапись соответствующих ячеек памяти корректными значениями.

6. Заключение

В статье представлен метод построения эксплойтов для обнаруженных ошибок. Метод основан на символьном выполнении бинарного кода, он позволяет преодолевать рандомизацию адресного пространства с помощью трамплинов, автоматизируются действия, которые не могут быть проведены без участия пользователя. Метод был реализован в виде программного инструмента, являющегося частью среды анализа бинарного кода. Его применение помогает разработчику понять, какие ошибки следует исправлять в первую очередь, как представляющие наибольшую угрозу для безопасности ПО.

Наиболее близкие результаты были показаны в работах коллектива из университета Карнеги – Меллон. В работе AEG [14] была представлена первая система для автоматической генерации эксплойтов. Поиск потенциально

эксплуатируемых уязвимостей ведется в исходном коде, а построение эксплойта происходит на уровне бинарного кода. Система MAYHEM [4] является развитием системы AEG для поиска уязвимостей и использует в своей работе только бинарный код. Их возможности позволяют перебирать состояния программы во время ее динамического символьного выполнения и обнаруживать срабатывание программных дефектов. К сожалению, все разработанные этим коллективом инструменты недоступны. Также следует отметить, что в приведенных публикациях авторы решают задачи в аналогичных ограничениях: отключаются некоторые механизмы защиты кода.

Помимо того, известны и другие системы символьного выполнения, работающие с бинарным кодом: BitFuzz [15], FuzzBall [16], S2E [17], SAGE [18], Avalanche [19] и др. Большинство из них в первую очередь нацелены на перебор состояний программы и не располагают средствами построения эксплойтов.

Представленные в данной работе результаты предлагают законченный метод, позволяющий оценить найденные ошибки. Построение эксплойта гарантирует безошибочную классификацию ошибки как критической. Дальнейшие работы предполагают построение эксплойта с использованием ROP-компиляции, позволяющей успешно преодолевать защиту в виде неисполняемого стека, эксплуатацию других типов уязвимостей, и практическое апробирование метода на других процессорных архитектурах.

Список литературы

[1]. А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. // Труды Института системного программирования РАН, том 22, 2012 г. стр. 131-152.

[2]. King J.C. Symbolic execution and program testing. //*Commun. ACM.* – 1976. – №19.

[3]. Miller, C., Caballero, J., Johnson, N. M., Kang, M. G., McCamant, S., Poosankam, P., Song, D. Crash analysis with BitBlaze // *at BlackHat USA, 2010*.

[4]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code // IEEE Symposium on Security and Privacy. – 2012.

[5]. Avgerinos, T., Rebert, A., Cha, S. K., & Brumley, D. (2014, May). Enhancing symbolic execution with veritesting. // *In ICSE*, May 2014, pp. 1083-1094.

[6]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[7]. П.М. Довгалюк, Н.И. Фурсова, Д.С. Дмитриев. Перспективы применения детерминированного воспроизведения работы виртуальной машины при решении задач компьютерной безопасности. // Материалы конференции РусКрипто’2013. Москва, 27 — 30 марта 2013.

[8]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. // Труды

Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[9]. Андрей Тихонов, Арутюн Аветисян, Вартан Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3, 2008. Стр. 66-71

[10]. Падарян В. А., Соловьев М. А., Кононов А. И. Моделирование операционной семантики машинных инструкций. // Программирование, № 3, 2011 г. Стр. 50-64.

[11]. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. // In Proc. of the USENIX Security Symposium, 2011.

[12]. Андрей Тихонов, Вартан Падарян. Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения. // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». 2009. стр. 131

[13]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal // Proceedings of PDPAR'03, July 2003

[14]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D. Brumley. AEG: Automatic exploit generation // *Commun. ACM*. – 2014.– №2.

[15]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. // In Proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010.

[16]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.

[17]. G. C. Vitaly Chipounov, Volodymyr Kuznetsov. S2E: A platform for in-vivo multi-path analysis of software systems. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 265–278.

[18]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. // In Proc. of the Network and Distributed System Security Symposium, Feb. 2008.

[19]. Исаев, И. К., Сидоров, Д. В., Герасимов, А. Ю., Ермаков, М. К. (2011). Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды Института системного программирования РАН, том 21, 2011 г., стр. 55-70.

Automated exploit generation method for stack buffer overflow vulnerabilities

V. A. Padaryan
<vartan@ispras.ru>
V. V. Kaushan
<korpse@ispras.ru>
A. N. Fedotov
<fedotoff@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. In this paper automated method for exploit generation is presented. This method allows to construct exploits for stack buffer overflow vulnerabilities and also to prioritize software bugs. It is applied to program binaries, without requiring debug information. The method is based on dynamic analysis and symbolic execution. We present a tool implementing the method. We used this tool to generate exploits for 8 vulnerabilities in both Linux and Windows programs, 3 of which were undocumented at the time this paper was written.

Keywords: bug classification; vulnerability exploitation; binary code; dynamic analysis; symbolic execution.

References

[1]. Tikhonov A.YU., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda. [Combined (static and dynamic) analysis of binary code]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 22, 2012, pp. 131-152 (in Russian).
[2]. King J.C. Symbolic execution and program testing. *Commun. ACM.* – 1976. – №19.
[3]. Miller, C., Caballero, J., Johnson, N. M., Kang, M. G., McCamant, S., Poosankam, P., Song, D. Crash analysis with BitBlaze. *at BlackHat USA, 2010*.
[4]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. *IEEE Symposium on Security and Privacy,* 2012.
[5]. Avgerinos, T., Rebert, A., Cha, S. K., & Brumley, D. (2014, May). Enhancing symbolic execution with veritesting. *In ICSE*, May 2014, pp. 1083-1094.
[6]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Metody i programmnye sredstva, podderzhivayushhie kombinirovannyj analiz binarnogo koda [Methods and software tools for combined binary code analysis]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2014, vol. 26, no. 1, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8
[7]. Dovgalyuk P.M., Fursova N.I., Dmitriev D.S. Perspektivyi primeneniya determinirovannogo vosproizvedeniya rabotyi virtualnoy mashinyi pri reshenii zadach kompyuternoy bezopasnosti. [Prospects of using virtual machine deterministic replay in solving computer security problems]. *Materialyi konferentsii RusKripto'2013 [The Proceedings RusCrypto'2013]*, 2014 (In Russian).
[8]. Dovgalyuk P.M., Makarov V.A., Padaryan, M.S. Romaneev, V.A., Fursova N.I. Primenenie programmnykh ehmulyatorov v zadachakh analiza binarnogo koda [Application of software emulators for the binary code analysis]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2014, vol. 26, no. 1, pp. 277-296 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-9.
[9]. Tikhonov A.YU., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. *Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Informations security aspects. Computer sistems]*, 2008, №3. pp. 66-71 (in Russian)
[10]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Modelirovanie operatsionnoy semantiki mashinnyih instruktsiy. [Simulation of operational semantics of machine instructions]. *Programming and Computer Software*, May 2011, Volume 37, Issue 3, pp 161 – 170 , DOI 10.1134/S0361768811030030 (In Russian)
[11]. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. *In Proc. of the USENIX Security Symposium*, 2011.
[12]. Tikhonov A.YU., Padaryan V.A., Primenenie programmnogo slaysinga dlya analiza binarnogo koda, predstavlennogo trassami vyipolneniya.[Using program slicing for bynary code represented by execution traces] *Materialyi XVIII Obscherossiyskoy nauchno-tehnicheskoy konferentsii «Metodyi i tehnicheskie sredstva obespecheniya bezopasnosti informatsii». [The Proceedings of XVIII Russian science technical conference "Methods and technical information security tools"]* 2009. pp 131 (In Russian).
[13]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. *Proceedings of PDPAR'03*, July 2003
[14]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D. Brumley. AEG: Automatic exploit generation. *Commun. ACM.* – 2014.– №2.
[15]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. *In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, Mar. 2012.
[16]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. *In Proc. of the ACM Conference on Computer and Communications Security*, Chicago, IL, October 2010.
[17]. G. C. Vitaly Chipounov, Volodymyr Kuznetsov. S2E: A platform for in-vivo multi-path analysis of software systems. *In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.
[18]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. *In Proc. of the Network and Distributed System Security Symposium*, Feb. 2008.
[19]. Isaev, I. K., Sidorov, D. V., Gerasimov, A. YU., Ermakov, M. K. (2011). Primenenie dinamicheskogo analiza dlya avtomaticheskogo obnaruzheniya oshibok v programmakh ispol'zuyushhikh setevye sokety [Using dynamic analysis for automatic bug detection in software that use network sockets]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2011, vol. 21, pp. 55-70 (In Russian).