Автоматическое обнаружение использования неинициализированных значений в рамках полносистемной эмуляции

Н. А. Белов <zodiac@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Описанный в данной статье метод позволяет автоматически обнаруживать использование неинициализированных значений в рамках полносистемной эмуляции. Это актуально для такого низкоуровневого программного обеспечения, как, например, BIOS или начальный загрузчик, выполняющие функции инициализации оборудования и загрузки операционной системы. Ошибки в данных программных системах наиболее опасны и приводят к неработоспособности всей системы целиком. Программное обеспечение подобного рода затруднительно тестировать на реальной аппаратуре, поэтому для этих целей используются эмуляторы различных архитектур. В рамках работы был разработан метод использования теневой памяти (памяти, содержащей информацию об исходной памяти) для хранения и отслеживания состояния регистров и ячеек гостевой памяти. Также были сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках. Разработанный метод был реализован и протестирован на гостевой системе архитектуры x86 в полносистемном эмуляторе QEMU.

Ключевые слова: обнаружение неинициализированных значений; полносистемная эмуляция; инструментирование.

DOI: 10.15514/ISPRAS-2016-28(5)-1

Для цитирования: Белов Н.А. Автоматическое обнаружение использования неинициализированных значений в рамках полносистемной эмуляции. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 11-26. DOI: 10.15514/ISPRAS-2016-28(5)-1

1. Введение

В современном мире происходит постоянное усложнение компьютерных систем, будь то персональные компьютеры, мобильные телефоны, некоторые модели электронных наручных часов, различное сетевое оборудование или

другие программно-аппаратные комплексы. Поскольку каждая такая система использует множество своего различного низкоуровневого программного обеспечения, такого как BIOS, UEFI, начальные загрузчики и прошивки, выполняющие, к примеру, функции инициализации и тестирования оборудования и дальнейшей загрузки операционной системы, то ошибки в данных программных системах наиболее опасны и приводят к неработоспособности всей системы целиком.

Однако, программное обеспечение подобного рода затруднительно тестировать на реальной аппаратуре, так как нет доступа ко всему контексту выполнения (регистрам и памяти), а использовать аппаратный отладчик дорого и неудобно, поэтому для этих целей используются специальные программы – эмуляторы [1].

2. Обзор и метод решения задачи

В данной статье рассматривается разработка и реализация метода автоматического обнаружения использования неинициализированных значений в рамках полносистемной эмуляции.

Для решения данной задачи был разработан метод хранения и отслеживания состояния регистров и ячеек памяти гостевой системы (инициализированы они или нет), а также сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках.

После этого разработанный метод был реализован и протестирован в полносистемном эмуляторе QEMU [2].

2.1 Обзор существующих методов

Рассмотрим работы, в которых решались похожие задачи инструментирования внутреннего представления и поиска ошибок при работе с памятью.

1. Android Memory Checker Component входил в состав операционной системы Android и использовался для тестирования только пользовательских приложений внутри системы. Метод был реализован на основе эмулятора QEMU и специальной версии библиотеки libc.so. Он позволял находить такие ошибки при работе с памятью, как запись и чтение за пределами выделенного блока, попытки использования некорректных указателей для освобождения или перераспределения памяти, утечки памяти.

   В 2014 году данное средство было исключено из состава Android в связи со сложностью поддержки [3].
2. Valgrind – свободное программное обеспечение, предназначенное для профилирования, отладки использования памяти и обнаружения ее утечек. Memcheck является его основным модулем, обеспечивающим проверку корректности работы тестируемого приложения с памятью. Он может обнаруживать различные виды ошибок, преимущественно

возникающих в программах на Си и Си++. Memcheck обнаруживает множество ошибок, но минусом данного инструмента является то, что он может проверять только пользовательские приложения [4].

Таким образом, оба инструмента не позволяют тестировать низкоуровневое программное обеспечение и используются только для тестирования пользовательских приложений внутри загруженной операционной системы.

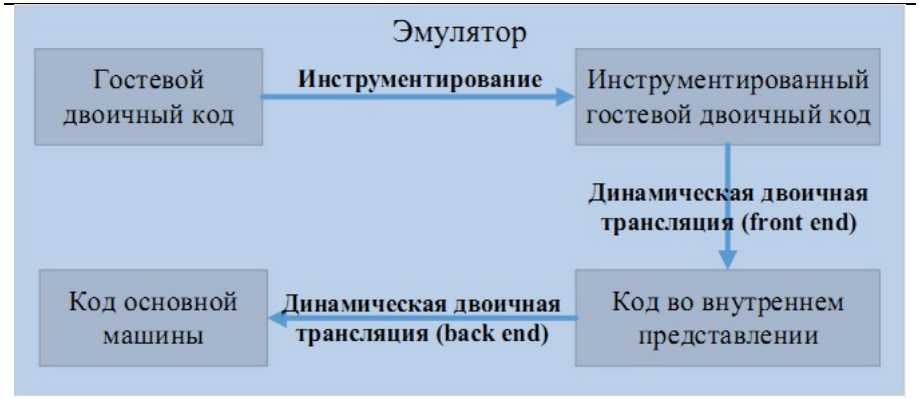
2.2 Инструментирование внутреннего представления

Ключевой метод, лежащий в основе работы программной системы Valgrind, – это динамическая двоичная трансляция. Данный метод используется для перевода двоичного кода из одного представления в другое «на лету» и имеет широкое распространение. Например, он применяется для полносистемной эмуляции [5]. Также с его помощью можно инструментировать различное программное обеспечение.

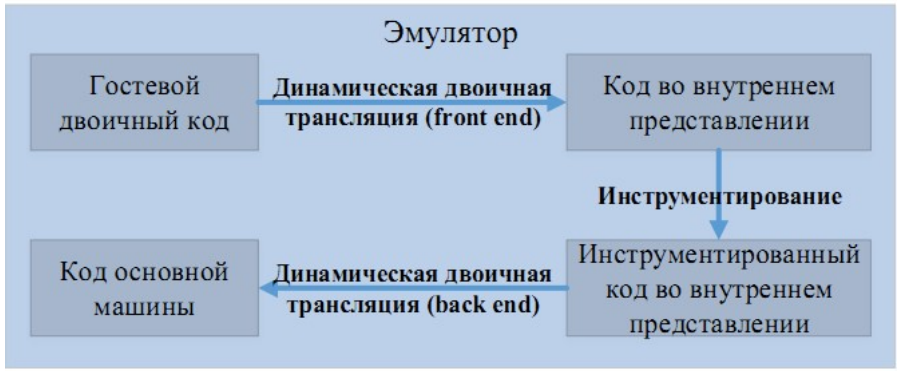
Существует два различных подхода к реализации инструментирования исходного двоичного кода при полносистемной эмуляции, различия которых изображены на рис. 1.

1. **Инструментирование внутри гостевой среды.** В данном случае инструментирование выполняется программным обеспечением внутри гостевой системы, используя ее адресное пространство. Однако, подход имеет очень сильное замедление и трансляция выполняется дважды: во время инструментирования необходимого процесса и во время эмуляции.
2. **Инструментирование вне гостевой среды.** В отличие от предыдущего случая, при таком подходе инструментирование производится эмулятором. Инструментирующий код использует адресное пространство основной машины.

В данной статье мы будем использовать инструментирование вне гостевой среды, так как поставленная задача предполагает возможность тестирования низкоуровневого программного обеспечения, когда невозможно использовать адресное пространство гостевой системы для инструментирующего кода и сопутствующих данных.



Инструментирование внутри гостевой среды



Инструментирование вне гостевой среды

Рис. 1. Различие между методами инструментирования кода

Fig. 1. Differences between code instrumentation methods

2.3 Теневая память

Теневая память – это память, значения которой содержат какую-либо информацию об исходной памяти. При решении поставленной задачи будем использовать теневую память, содержащую информацию об инициализированности значений по соответствующим адресам физической памяти виртуальной машины. Теневая память будет покрывать каждый бит гостевой памяти, причем будем использовать значение «единица» для обозначения того, что бит инициализирован, и «ноль» в противном случае.

Определим операции над значениями теневой памяти аналогично тому, как это сделано в Memcheck [6]. Будем использовать обозначение ***d*** для фактического значения гостевой памяти и ***s*** для соответствующего ему

теневого значения, а ***X*** и ***Y*** для размера операнда и результата в байтах (размер «ноль» используется для однобитового операнда).

1. ***DifD(s1, s2)*** («инициализировано если хотя бы одно инициализировано»). Операция возвращает результат, каждый бит которого установлен в значение «инициализирован», если хотя бы один соответствующий бит операнда имеет значение «инициализирован». Операция реализуется через побитовое логическое «ИЛИ».
2. ***UifU(s1, s2)*** («неинициализировано если хотя бы одно неинициализировано»). Операция возвращает результат, каждый бит которого установлен в значение «неинициализирован», если хотя бы один соответствующий бит операнда имеет значение «неинициализирован». Операция реализуется через побитовое логическое «И».
3. ***ImproveAND(d, s)*** и ***ImproveOR(d, s)***. Операции используются для вычисления теневого значения результата побитовых логических операций «И» и «ИЛИ» соответственно. Значение очередного бита результата определяется в зависимости от соответствующих битов ***d*** и ***s*** по следующим правилам:
   а) для операции ***ImproveAND***: бит имеет значение «инициализирован», если ***d = 0*** и ***s*** имеет значение «инициализировано», так как если один из аргументов логической операции «И» инициализированный нулевой бит, то второй аргумент не повлияет на результат,
   б) для операции ImproveOR: бит имеет значение «инициализирован», если d = 1 и s имеет значение «инициализировано» (из тех же соображений, что и для операции ImproveAND),
   в) для обеих операций в остальных случаях бит получает значение «неинициализирован».
4. ***Left(s)***. Операция симулирует наиболее неблагоприятный вариант распространения статуса неинициализированности значения через флаг переноса во время сложения или вычитания целых чисел. Результатом операции является значение v, в котором установлены все биты от самого старшего до установленного самого младшего. Например, ***Left(0001 0100) = 1111 1100***. Операция реализуется через операции побитового логического «ИЛИ», побитового отрицания и смены знака (***NEG***).
5. ***PCastXY(s)***. Данная группа операций представляет собой операции изменения размера операнда по следующему правилу: если все биты операнда имеют значение «инициализирован», то все биты результата будут иметь значение «инициализирован», иначе все биты результата имеют значение «неинициализирован». Например, ***PCast12(0001 0100) = 0000 0000 0000 0000*** и ***PCast12(1111 1111) = 1111 1111 1111 1111***.

   Данное преобразование используется в различных приближениях, в которых проверка статуса инициализированности требует рассмотрения полного значения.
6. ***ZWidenXY(s)***. Группа операций беззнакового расширения аргумента. Для заполнения новых битов используется значение «инициализирован».
7. ***SWidenXY(s)***. Группа операций знакового расширения аргумента. Для заполнения новых битов используется значение старшего бита аргумента.

Начальная инициализация теневой памяти выполняется следующим образом:

1) вся память и регистры помечаются как неинициализированные;
2) участки ***ROM***-памяти помечаются как инициализированные;
3) регистр указателя стека помечается как инициализированный;
4) регистр счетчика инструкций помечается как инициализированный.

2.4 Обнаружение неинициализированных значений

При каждом обнаружении использования неинициализированного значения существуют две стратегии уведомления об ошибке: сообщить об этом сразу или распространить статус инициализированности значений на результат, а затем при наступлении определенных условий проверить его.

Первый вариант будет давать множество ложноположительных срабатываний, так как даже программы на Си, полностью соответствующие стандарту, регулярно копируют неинициализированные значения из памяти и обратно. Рассмотрим пример такого кода, приведенный на рис. 2.

```
1 struct S { int i; char c; };
2 struct S s1, s2;
3 s1.i = 42;
4 s1.c = 'z';
5 s2 = s1;
```

Рис. 2. Пример кода на языке Си

Fig. 2. Code example in C

Все распространенные компиляторы (например, GCC) выровняют размер ***struct S*** до целого числа слов, поэтому структура ***s1*** будет занимать восемь байт, из которых только пять будут инициализированными. Для присваивания ***s1 = s2*** компилятор GCC сгенерирует ассемблерный код, представленный на рис. 3.

```
1 mov eax,DWORD PTR [esp+0x8]
2 mov edx,DWORD PTR [esp+0xc]
3 mov DWORD PTR [esp],eax
4 mov DWORD PTR [esp+0x4],edx
```

*Рис. 3. Ассемблерный код для присваивания **s1 = s2***

*Fig. 3. Assembly code for **s1 = s2** assignment*

Таким образом, из структуры ***s1*** в структуру ***s2*** будут скопированы все восемь байт, несмотря на их смысл, что должно привести к сообщению об ошибке использования неинициализированных значений. В нашей работе будем сообщать о таких ошибках только при наступлении определенных событий, а в остальных случаях передавать неинициализированные значения в результат.

Выделим случаи, при которых необходимо сообщать об использовании неинициализированных значений:

1) неинициализированное значение является адресом для операций загрузки или выгрузки значений из памяти;
2) выполняется условный переход на основании неинициализированного значения;
3) выполняется переход на неинициализированный участок памяти.

2.5 Обновление значений теневой памяти

Как уже было сказано, немедленное информирование об ошибке требуется в малом числе случаев, в остальных же необходимо обновить значение теневой памяти после выполнения исходной операции. Для этого введем следующие правила.

1. **Константы.** Все константы всегда считаются инициализированными значениями.
2. **Операции копирования данных.** Выполняем копирование теневого значения источника в теневое значение принимающего аргумента.
3. **Сложение и вычитание.** Пусть дано ***d3 = Add(d1, d2)*** или ***d3 = Sub(d1, d2)***, тогда очередной бит результата будет иметь значение «инициализирован», когда биты обоих аргументов «инициализированы». Однако, бит результата может быть также «неинициализирован» в том случае, если заем из старших или перенос из младших битов оказался неинициализированным значением. Таким образом, теневой результат определяется как ***s3 = Left(UifU(s1, s2))***.

   Такие же правила используются и для умножения, хотя это не совсем точно. Произведение двух множителей с **N** и **M** последовательными инициализированными младшими битами имеет **N + M** инициализированных младших бит, а не **min(N, M)**, как получается при использовании данного метода. Однако, как показывает опыт разработчиков Memcheck, смысла усложнять используемое решение

нет, так как ложных срабатываний при умножении возникает ничтожно мало.

4. **Операция смены знака (NEG).** Результат теневого значения аналогичен результату выполнения операции ***Sub(0, d)***.
5. **Сложение с переносом и вычитание с займом.** В данной операции будем использовать дополнительный однобитовый аргумент ***vfl***, являющийся теневым значением регистра флагов (например, в архитектуре x86 регистра ***EFLAGS***), который отвечает за перенос или займ. Если этот флаг имеет значение «неинициализирован», то значение всей операции также будет неинициализировано. Таким образом, теневой результат операции определяется как ***s3 = UifU(Left(UifU(s1, s2)), PCast0X(vfl))***, где ***X*** – длина ***s1***, ***s2*** и ***s3***.
6. **Операция побитового исключающего «ИЛИ».** Для ***d3 = Xor(d1, d2)*** результат теневой памяти будет равен ***s3 = UifU(s1, s2)***.
7. **Операция побитового отрицания.** Результат теневого значения аналогичен результату выполнения операции ***Xor(0xFF…FF, d)***.
8. **Побитовые логические «И» и «ИЛИ».** Данные операции требуют проверки фактических значений аргументов, а также их теневых значений. Пусть даны операции ***d3 = And(d1, d2)*** и ***d3 = Or(d1, d2)***, тогда теневой результат операций определяется соответственно как ***s3 = DifD(UifU(d1, d2), DifD(ImproveAND(d1, s1), ImproveAND(d2, s2)))*** и ***s3 = DifD(UifU(d1, d2), DifD(ImproveOR(d1, s1), ImproveOR(d2, s2)))***.
9. **Битовые сдвиги.** Существуют следующие операции битовых сдвигов: сдвиг влево (***shl***), беззнаковый и знаковый сдвиги вправо (***shr*** и ***sar*** соответственно), циклические сдвиги влево и вправо (***rotl*** и ***rotr*** соответственно). Во всех случаях если значение, на которое производится сдвиг, неинициализировано, то весь результат также будет неинциализирован. Иначе для получения теневого результата выполним над теневым значением аргумента такую же операцию, как и для оригинальных значений. Таким образом, для ***d3 = OP(d1, d2)***, где ***d2*** – число, на которое происходит сдвиг, а ***X*** и ***Y*** – длина ***d1***/***d3*** и ***d2*** соответственно, получаем теневой результат ***s3 = UifU(PCastYX(s2), OP(s1, d2))***.

   Для всех операций теневое значение аргумента обрабатывается как и сам аргумент. Для операций ***rotl*** и ***rotr*** теневое значение должно быть сдвинуто абсолютно так же, как и оригинальное. Операции ***shl*** и ***shr*** используют для заполнения нулевые значения, поэтому новые биты теневого результата должны получить значение «инициализировано». Операция ***sar*** копирует старший бит аргумента, поэтому в теневой результат также необходимо копировать старший бит теневого аргумента.

10. **Операции изменения размера.** Операции изменения размера выполняются для теневого результата с помощью операций ***SWiden*** и ***ZWiden*** (для знаковых и беззнаковых расширений соответственно).
11. ***Операции, влияющие на флаги.*** В архитектуре x86 большинство арифметических инструкций устанавливают флаги в регистре ***EFLAGS***. Будем использовать один бит ***vfl*** для хранения состояния инициализированности данного регистра. Для каждой арифметической операции, устанавливающей флаги, будем сначала вычислять теневой результат по описанным выше правилам, а затем получать значение флага как значение функции ***PCastX0*** от теневого результата.

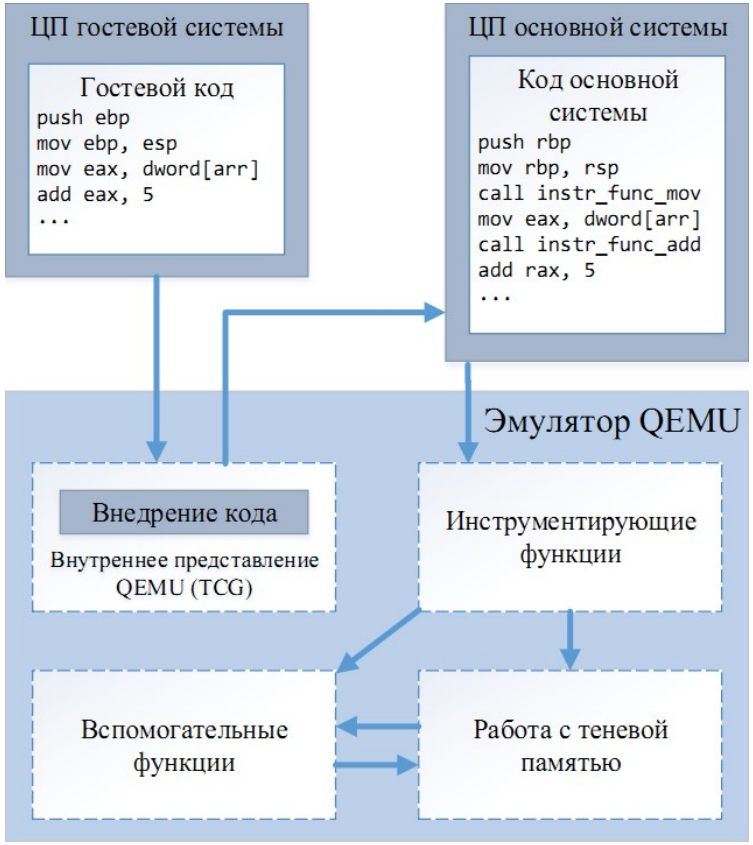


Рис. 4. Взаимодействие частей разработанного метода

Fig. 4. Interaction between parts of developed method

3. Описание реализации метода

В данном разделе будет изложен способ реализации каждого этапа, описанного в предыдущем разделе, в эмуляторе QEMU. Для этого рассмотрим алгоритм взаимодействия частей разрабатываемого метода, представленный на рис. 4.

На данном рисунке мы видим, что внедрение инструментирующего кода будет происходить на стадии трансляции гостевого кода в код внутреннего представления эмулятора. Затем при выполнении кода основной машиной будут вызываться специальные инструментирующие функции по одной перед выполнением каждой инструментируемой операции. Вычисление адреса теневой памяти и операции над ней выделим во множество «вспомогательных функций».

3.1 Внедрение кода

Как уже было сказано ранее, эмулятор QEMU для своей работы использует динамический транслятор TCG, хранящий внутреннее представление в виде двух массивов: коды операций и аргументы операций. Для выполнения инструментирования будем использовать специальные функции – ***helpers*** (здесь и далее для обозначения таких функций будет использоваться термин «функции-помощники»), которые выполняются в адресном пространстве основной машины и специальным образом работают с теневой памятью в зависимости от инструментируемой операции.

Таким образом, внедрение кода будет происходить в три шага.

1. **Вычисление размера внедряемого инструментирующего кода: количества операций и количества аргументов.** Для каждой инструментируемой операции будем хранить два параметра внедряемого кода: количество операций и их суммарное число аргументов. Затем создадим цикл по массиву операций внутреннего представления, в котором будем проверять необходимо ли инструментировать данную операцию. При положительном ответе будем увеличивать соответствующие счетчики: ***total_count*** и ***total_size***.
2. **Сдвиг массивов внутреннего представления для освобождения места под инструментирующий код.** В каждом из двух массивов внутреннего представления сдвинем последовательно все элементы таким образом, чтобы в начале оказалось свободное место под ***total_count*** и ***total_size*** элементов соответственно.
3. **Внедрение кода.** Сформируем массив операций и массив аргументов следующим образом. Для каждой исходной операции будем смотреть, необходимо ли ее инструментировать. В массивы будем записывать исходную операцию и ее аргументы. В том случае, если требуется

инструментирование данной операции, то перед этим еще будем записывать инструментирующий код и его аргументы.

3.2 Теневая память в эмуляторе QEMU

В данном подразделе рассмотрим существующий способ организации гостевой памяти в эмуляторе QEMU, реализацию для него теневой памяти, а также теневую память для переменных внутреннего представления.

Организация памяти в QEMU и теневая память.

Все страницы гостевой памяти, используемые в процессе работы эмулятора, выделяются непрерывными блоками и хранятся в связном списке. Для организации теневой памяти поступим аналогичным образом.

Каждый такой блок гостевой памяти выделяется вызовом функции ***qemu_ram_alloc()***, которая принимает размер в байтах и выделяет память через системный вызов ***mmap()*** на основной машине в процессе QEMU. Существует также несколько других функций, которые позволяют выделять блоки гостевой памяти, но все они, включая функцию ***qemu_ram_alloc()***, для добавления блока в список используют функцию ***ram_block_add()***. Добавим в ее исходный код вызов функции, которая будет создавать блок теневой памяти, соответствующий новому блоку гостевой памяти.

Для ускорения работы по поиску виртуального адреса основной машины, соответствующего виртуальному адресу гостевой машины, эмулятор использует буфер ассоциативной трансляции (*Translation lookaside buffer*, *TLB*), который хранит смещение гостевого виртуального адреса к виртуальному адресу гостевой машины. Поэтому нам необходимо иметь функцию, вычисляющую указатель на теневую память, соответствующую данному виртуальному адресу.

Для работы с теневой памятью реализуем функции проверки гостевой памяти на инициализированность и изменения значения теневой памяти для соответствующего участка гостевой памяти.

Теневая память для переменных внутреннего представления.

Внутренним представлением эмулятора поддерживаются переменные, которые подразделяются на три типа: глобальные (регистры), временные («*temporary*») и локальные временные («*local*»). Глобальные переменные существуют всегда и определяются разработчиком эмулируемой архитектуры. Временная переменная существует только в пределах базового блока (последовательности инструкций, имеющей одну точку входа и одну точку выхода) и уничтожается после его окончания. Локальная временная переменная существует только в пределах блока трансляции (последовательности инструкций, которую эмулятор транслирует за раз). Временные и локальные временные переменные выделяются для каждой функции по отдельности.

QEMU для хранения всех описанных выше типов переменных использует единый массив, поэтому теневую память для них реализуем аналогично. Теневую память каждой переменной будем описывать двумя параметрами: адрес, с которого было загружено значение, и теневое значение. Инициализируем полученный массив в соответствии с правилами, описанными в главе «Обзор и метод решения задачи».

Для работы с теневой памятью переменных внутреннего представления реализуем функции проверки переменной на инициализированность и изменения значения теневой памяти для соответствующей переменной.

При выполнении трансляции гостевого кода во внутреннее представление эмулятор не использует в качестве аргументов для операций переменные, значения которых не объявлены. Более того, в эмуляторе имеются такие проверки, поэтому нет необходимости каждый раз сбрасывать значение теневой памяти для переменных в исходное состояние.

3.3 Обнаружение перехода на неинициализированную область памяти

Переход на неинициализированную область памяти осуществляется следующим образом. В QEMU используется функция ***get_page_addr_code()***, вызывающаяся каждый раз при трансляции гостевого кода (при включенном режиме «singlestep», когда за один раз транслируется одна инструкция). Эта функция имеет одним из своих аргументов гостевой адрес, поэтому добавим в ее исходный код вызов нашей функции, проверяющей инициализированность значения по этому адресу и в случае необходимости выводящей ошибку.

3.4 Инструментирование операций загрузки и сохранения

В эмуляторе QEMU существует две инструкции внутреннего представления, предназначенные для копирования между эмулируемой памятью гостевой машины и переменными:

1. ***qemu_ld_i32*** загружает из эмулируемой памяти гостевой машины значение по адресу ***addr*** в переменную ***ret***.
2. ***qemu_st_i32*** сохраняет значение переменной ***val*** в эмулируемую память гостевой машины по адресу ***addr***.

Инструментирование инструкции загрузки.

Инструментирующая функция-помощник для инструкции загрузки выполняет следующие действия:

1) проверяет значение аргумента ***addr*** на инициализированность, при необходимости уведомляя об ошибке;
2) копирует значение теневой памяти по адресу ***addr*** в теневую память для аргумента ***ret***.

Инструментирование инструкции сохранения.

Инструментирующая функция-помощник для инструкции сохранения работает аналогично инструкции загрузки и выполняет следующие действия:

1) проверяет значение аргумента ***addr*** на инициализированность, при необходимости уведомляя об ошибке;
2) копирует значение теневой памяти для аргумента ***ret*** в теневую память по адресу ***addr***.

3.5 Инструментирование инструкции условного перехода

Инструкция условного перехода во внутреннем представлении имеет следующий прототип: ***brcond_i32 arg1, arg2, cond, int***. Данная инструкция выполняет переход на метку с индексом ***int***, если условие перехода, задаваемое аргументом ***cond*** – истина.

Инструментирующая функция-помощник для инструкции условного перехода проверяет значение аргументов ***arg1*** и ***arg2*** на инициализированность и, если хотя бы один из них неинициализирован, уведомляет об ошибке.

3.6 Инструментирование остальных операций

Инструментирование остальных операций (операции копирования значения, арифметические и логические операции, операции сдвига и так далее) осуществляется согласно правилам, описанным в главе «Обзор и метод решения задачи».

Каждой функции-помощнику передаются индексы переменных внутреннего представления, являющихся аргументами исходной операции, после чего в теневую память результата записывается вычисленное теневое значение.

4. Тестирование

Тестирование производилось на гостевой архитектуре x86. Была использована операционная система Baremetal OS – небольшой код, инициализирующий процессор и передающий управление пользовательской программе.

Код операционной системы был изменен для создания условий, позволяющих протестировать все три случая обнаружения использования неинициализированных значений, описанных в главе 4.3.

Тестирование производилось следующим образом.

1. **Неинициализированное значение является адресом для операций загрузки или сохранения значений из памяти.** Был написан код, загружающий значение по адресу, указывающему на заведомо неинициализированное значение, а затем выполнена загрузка по адресу, на который указывает значение.
2. **Выполняется условный переход на основании неинициализированного значения.** Был написан код, загружающий значение по адресу, указывающему на заведомо

неинициализированное значение, а затем выполняющий условный переход на основе этого значения.

3. **Выполняется переход на неинициализированный участок памяти.** Был написан код, выполняющий переход на заведомо неинициализированное значение адреса.

5. Заключение

В данной статье был описан метод автоматического обнаружения использования неинициализированных значений в рамках полносистемной эмуляции. В разработанном методе используется технология теневой памяти для хранения и отслеживания состояния регистров и ячеек гостевой памяти.

Также в статье были сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках.

Разработанный метод был реализован и протестирован на гостевой системе архитектуры x86 в полносистемном эмуляторе QEMU. Для тестирования было написано три простых примера на каждый случай обнаружения использования неинициализированных значений. Метод показал свою корректную работу на всех примерах.

Список литературы

[1]. Smith J., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, 656 p.

[2]. QEMU Emulator User Documentation (online). Доступно по ссылке: http://qemu.weilnetz.de/qemu-doc.html, 13.11.2014.

[3]. Android Memory Checker Component (online). Доступно по ссылке: https://github.com/android/platform_external_qemu/blob/791e96ffc61d52eae80f94129a93ff67474f3ff9/docs/ANDROID-MEMCHECK.TXT, 3.12.2014.

[4]. Memcheck: a memory error detector (online). Доступно по ссылке: http://valgrind.org/docs/manual/mc-manual.html, 16.11.2014.

[5]. Bellard F. QEMU, a Fast and Portable Dynamic Translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 41.

[6]. Seward J., Nethercote N. Using Valgrind to Detect Undefined Value Errors with Bit-precision. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 2.

Automatic uninitialized value usage detection during full-system emulation

N.A. Belov <zodiac @ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Developed method, which is described in this paper, is capable of automated detection of uninitialized values within the scope of full-system emulation. This method is of immediate interest for low-level software, such as BIOS or initial loader, which initializes hardware and loads the operating system. Errors in this kind of software are the most dangerous and lead to system shutdown. This sort of software is difficult to test on real hardware, consequently emulators of different architectures are used for these tasks. In the context of this work a new method of using shadow memory for storing and tracking register states and guest system memory cells. Criteria for detection of uninitialized variables usage and error reporting were defined. For example, these situations fall under the criteria: uninitialized value is the address for loading and unloading values from and to the memory, conditional jump is performed based on uninitialized value or to an uninitialized memory chunk. Developed method was implemented and tested in the guest system of x86 architecture in full-system emulator QEMU. System consists of few instructions, which initialize a processor and transfers control to a user application. Testing was performed on three simple examples for each of the criteria for unitialized values detection. Developed method demonstrated correct results on all examples.

Keywords: automatic uninitialized value usage detection; full-system emulation; instrumentation.

DOI: 10.15514/ISPRAS-2016-28(5)-1

For citation: Belov N.A. Automatic uninitialized value usage detection during full-system emulation. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 11-26 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-1

References

[1]. Smith J., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, 656 p.
[2]. QEMU Emulator User Documentation (online publication). Available at: http://qemu.weilnetz.de/qemu-doc.html, accessed 13.11.2014.

[3]. Android Memory Checker Component (online publication). Available at: https://github.com/android/platform_external_qemu/blob/791e96ffc61d52eae80f94129a93ff67474f3ff9/docs/ANDROID-MEMCHECK.TXT, accessed 3.12.2014.
[4]. Memcheck: a memory error detector (online publication). Available at: http://valgrind.org/docs/manual/mc-manual.html, accessed 16.11.2014.
[5]. Bellard F. QEMU, a Fast and Portable Dynamic Translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 41.
[6]. Seward J., Nethercote N. Using Valgrind to Detect Undefined Value Errors with Bit-precision. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 2.