Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Разработка средства проверки корректности адресов возврата на платформе S2E

Курсовая работа студента 345 группы

Еварда Вадима Евгеньевича

Научный руководитель Зеленчук И.В.,

заведующий лабораторией

РУНЦ «Информационная безопасность»

Уральский федеральный университет

Санкт-Петербург

2012

Содержание

Введение

В коде существенного количества приложений встречаются ошибки в логике, в работе с памятью, файлами и т.п. Недостатки, с помощью которых можно нарушить целостность приложения и вызвать неправильную работу, называются уязвимостями. Значительную опасность представляют ошибки в работе с памятью в программах на языках Си и C++, так как в силу некоторых особенностей распространённых аппаратных платформ (x86, ARM) могут привести к самой опасной уязвимости — исполнению кода злоумышленника при обработке данных, оформленных им специальным образом.

Проблема поиска уязвимостей стоит как перед разработчиками приложения, старающимися их не допустить, так и перед атакующим, которому нужны достаточно универсальные и удобные механизмы для эксплуатации типичных ошибок.

Два принципиальных подхода к нахождению такого рода ошибок называют статическим и динамическим анализом.

Статический анализ производится без действительного исполнения исследуемого приложения. Современные инструментальные средства реализуют два основных типа проверок: на соответствие принятому стилю оформления (например, лишняя «;» после условия оператора if приведёт к генерации совершенно не предусмотренного кода) и на соблюдение кодом «намерений» программиста (достаточный размер приёмного массива, проверка пользовательских данных перед передачей их в вызовы ОС и т.п.) [8]. К недостаткам статического анализа относят высокую долю ложных срабатываний и некритических ошибок, не представляющих опасности для исполнения программы, а также практическая бесполезность при отсутствии исходных кодов, особенно в случае обфусцированного и зашифрованного приложения.

При *динамическом анализе* исследуется исполнение программы на реальном или виртуальном оборудовании. В том или ином виде он оказывается единственным выходом для проприетарных приложений. К основным его недостаткам относят существенное замедление исполнения программы и, в случае инструментирования, потенциальные существенные изменения работы программы. С другой стороны, выявляемые в ходе динамического анализа ошибки обычно можно проследить до первоисточника; именно они с наибольшей вероятностью проявились бы при реальном исполнении программы, и доля ложных

срабатываний чрезвычайно мала.

Подходы к тому, на каких входных данных проводить динамический анализ не снабжённых формальной моделью программ, эволюционировали от ручного тестирования и написания простейших модульных тестов к генерации тестов и фаззингу. Одним из перспективных направлений в данной области является символьное исполнение.

Символьное исполнение кода — интерпретация программы, при которой некоторым «входным» данным (например, файлы, сеть, переменные окружения) присваивается не конкретное, но символьное значение, и при ветвлении в потоке исполнения, вызванном обработкой символьных данных, исследуются все возможные пути. Такой анализ очень затратен, и одной из вариаций, призванных его ускорить, является избирательное символьное исполнение [4,5].

1. Задача работы

1.1. Сущность уязвимости

Архитектура x86 предусматривает единый аппаратный стек для сохранения адресов возврата при исполнении инструкции call и для хранения локальных данных функции. Это означает, что некорректная работа с локальными данными может привести к повреждению сохранённого адреса возврата, по которому при исполнении инструкции ret будет передано управление.

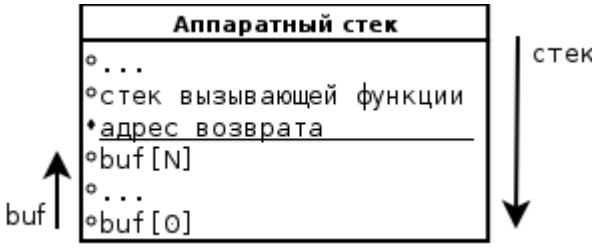

Рис. 1: Вызов функции (без параметров)

Распространённым примером такого рода ошибки является переполнение стекового буфера, когда в расположенный на стеке массив копируются данные большего размера, чем он может вместить. Например, это может происходить из-за того, что входные данные, которые помещаются в массив, *обычно* не превышают заданный объём. Если на практике они могут могут оказаться большего размера и ввод подконтролен атакующему (например, ошибка в библиотеке libpng может проявить себя при помещении браузером «жертвы» страницы с заготовленным зловредным изображением), то при переполнении буфера в стеке разместятся нужные нападающему данные. Совершив возврат по подменённому адресу, программа попадёт под контроль атакующего и может начать исполнять машинный код, попавший в стек вместе с теми же входными данными.

Современные процессоры, ОС и компиляторы предлагают способы защиты от эксплуатации таких уязвимостей, но они не универсальны (обходятся более сложными техниками [9] или перебором, который к тому же иногда можно существенно сократить) и вносят существенные накладные расходы.

1.2. Платформа S2E

Целью проекта S2E [1,2], развиваемого в Ecole Polytechnique Fédérale de Lausanne, является разработка платформы для построения средств динамического анализа, обладающих следующими свойствами:

- эффективный анализ семейств путей исполнения;
- достижение наибольшего правдоподобия путём запуска тестов на реальном стеке приложений;
- способность непосредственного анализа бинарных исполняемых файлов (без исходных кодов).

Система S2E основана на разработках виртуальной машины QEMU и среды символьного исполнения LLVM-биткода KLEE [4].

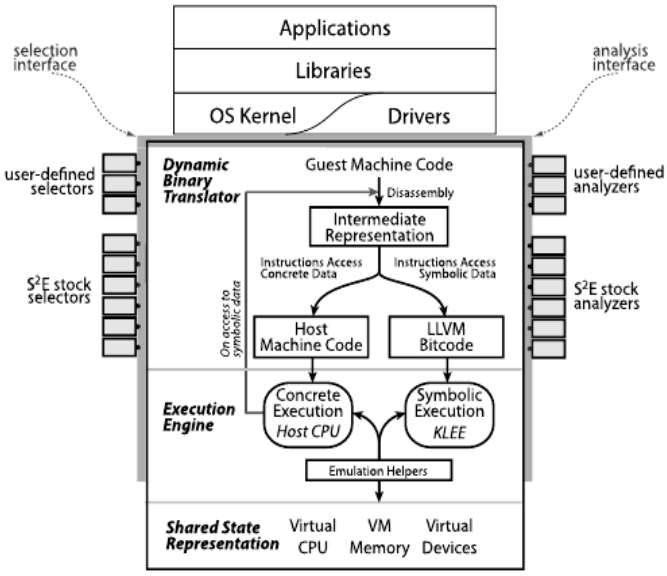

Рис. 2: Архитектура S2E

Состояние виртуальной машины делится на конкретное и символьное. Средствами QEMU код архитектуры x86 преобразуется в промежуточное представление. В случае работы только с конкретными данными промежуточный код исполняется так же, как в обычном эмуляторе QEMU, а работа с символьными данными поручается модифицированной среде KLEE.

1.3. Постановка задачи

Целью настоящей курсовой работы является разработка на базе S2E средства, позволяющего обнаружить несовпадение сохранённого при вызове адреса возврата с тем, который извлекается оттуда командой ret, а также условие, при котором такое несовпадение произойдёт.

Несмотря на то, что программа может осознанно совершать действия по замене сохранённого адреса возврата, в данной рамках данной работы все такие действия воспринимаются как ошибочные и требующие внимания.

2. Обзор похожих проектов

Данная работа задумывалась как основа для альтернативы закрытому AEG [6] – инструментальному средству, обнаруживающему уязвимости и автоматически генерирующему для них эксплоиты, приводящие к исполнению кода атакующего. AEG использует KLEE для исследования потока исполнения приложения и требует наличия исходных кодов для отладочных x86- и LLVM-сборок. В простых случаях для работы AEG достаточно кода, сгенерированного декомпилятором.

Обнаружение неочевидного (то есть недоступного статическим анализаторам) повреждения адреса возврата в стеке требует наличия тестовых данных и наблюдения за исполнением приложения при их обработке либо же анализа падения. Из средств, исследующих отчёты о сбоях на возможность их эксплуатации, следует упомянуть «!exploitable» для Windows, CrashWrangler для OS X и дополнение «exploitable» к GDB. Они не включают в себя генерацию тестовых данных, оставляя эту задачу другим инструментам.

Подход, при котором входные для тестируемого приложения данные генерируются по заданным правилам, называют фаззингом. Существует множество средств, реализующих фаззинг входных файлов, сетевых протоколов и т.п. (Peach, SPIKE и другие). Из методов фаззинга как наиболее перспективный выделяют мутирующий, при котором незначительно изменяются корректные входные данные. По сравнению с символьным исполнением, фаззинг не гарантирует исследование всех возможных путей исполнения программы, но и не приводит к существенному замедлению работы тестируемого приложения.

В рамках закрытого проекта LESE [7], применяющего технику избирательного символьного исполнения, был разработан инструмент, находящий известные уязвимости в старых версиях wu-ftpd, BIND, Sendmail.

3. Реализация

Для поддержки пользовательских расширений платформа S2E использует механизм, предоставленный libsigc++ — библиотекой типобезопасных сигналов для C++. Привязывая свои обработчики к сигналам, определённым в базовом дополнении CorePlugin и некоторых других, разработчики средств, построенных на основе S2E, реализуют необходимую функциональность.

Код x86 при трансляции во внутреннее представление QEMU разбивается на *блоки трансляции*. Этим названием объединяется последовательность микроинструкций QEMU, завершающаяся инструкцией, которая изменяет поток управления (например, переходы, вызовы функций и возврат из них). Обычно трансляция блока совершается только в первый проход пути исполнения по нему. По завершении процесса испускается сигнал CorePlugin::onTranslateBlockEnd, в котором можно определить обработчик события окончания исполнения блока.

Дополнение ModuleExecutionDetector, позволяющее сконцентрироваться на одном приложении, экспортирует аналогичный сигнал onModuleTranslateBlockEnd, испускаемый только по завершении трансляции блока, относящегося к выбранному приложению. В обработчике данного сигнала мы проверяем, завершился ли блок инструкцией вызова функции или возврата, и в этих случаях привязываем к сигналу исполнения данных блоков обработчики, реализующие логику проверки.

Для работы ModuleExecutionDetector необходимо указать ему параметры адресного пространства процесса: физические и виртуальные адреса, количество занимаемой памяти и т.п. Для гостевых систем Linux/x86, для которых производилось тестирование, стандартным средством получения данных параметров является подгружаемая через LD_PRELOAD библиотека init_env.so, разработанная авторами S2E. Она переопределяет функцию __libc_start_main(), отвечающую за передачу управления функции main() при запуске ELF-приложения.

Данный код является частью glibc и помещается в исполняемый файл редактором связей (linker). Поскольку данный код достаточно сложен, манипулирует адресами возвратов и не относится у исследуемому приложению напрямую, а также может и вовсе отсутствовать при сборке другими инструментами, было решено ограничить действие инструмента временем исполнения настоящего тела программы. В случае простого приложения на C оно совпадает с исполнением функции main(). Для этого были введены дополнительные машинные инструкции, сигнализирующие о начале и конце исполнения программы. Исполнение их из

гостевого кода сигнализирует о начале и конце настоящего тела программы.

Проверку реализуем при помощи своего (программного) стека, содержащего только адреса возврата. При исполнении инструкции вызова мы копируем сохранённый адрес возврата в свой стек, а при парном ему выходе проверяем, что переход произойдёт по сохранённому адресу.

Так как при ветвлении, вызванном символьными данными, состояние потока исполнения раздваивается (fork), каждой ветке (в терминологии KLEE называемой State) необходима собственная копия стека адресов возврата. Поэтому стек сохраняется в наследнике класса PluginState, копируемом при каждом ветвлении.

4. Результаты

В данной работе описано создание инструментального средства RetChecker, построенного на платформе S2E и предназначенного для выявления условий, при которых происходит перезапись адреса возврата.

Протестировано приложение, которое в зависимости от аргументов командной строки переполняет или не переполняет стековый буфер. Дополнение TestCaseGenerator выводит пример, приводящий к краху приложения.

Подготовлена среда для исследования программ, разбирающих сложные форматы данных (изображения, звукозаписи, архивы), в которых нередко находятся опасные уязвимости.

5. Дальнейшее развитие

Дальнейшим развитием работы может являться исследование того, как приложения обрабатывают блок недоверенных входных данных перед тем, как он переполнит стековый буфер, генерация proof-of-concept эксплоитов для известных простых уязвимостей, например, для собранной командой LESE коллекцией уязвимых сетевых служб. Представляется перспективной разработка модуля для платформы построения эксплоитов Metasploit, генерирующего нужный эксплоит по описанию, составленному RetChecker.

Разработка экспериментальной ветви ARM для S2E может помочь в исследовании механизмов безопасности мобильных телефонов и снятии искусственных ограничений, накладываемых на пользователя производителями (jailbreak).

6. Используемые источники

[1] https://s2e.epfl.ch/

[2] http://dslab.epfl.ch/pubs/selsymbex

[3] http://code.google.com/p/avalanche/

[4] http://klee.llvm.org/

[5] «Использование статического и динамического анализа для повышения качества продукции и эффективности разработки»: http://www.swd.ru/index.php3?pid=828

[6] http://security.ece.cmu.edu/aeg/

[7] http://bitblaze.cs.berkeley.edu/lese.html

[8] «The S2E Platform: Design, Implementation, and Applications»:

http://dslab.epfl.ch/pubs/s2e-tocs.pdf

[9] http://en.wikipedia.org/wiki/Return-oriented_programming