Поиск ошибок выхода за границы буфера в бинарном коде программ*

В.В. Каушан <korpse@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается метод поиска ошибок выхода за границы буфера в рамках метода комбинированного (статико-динамического) анализа бинарного кода. Для поиска ошибок используется символьная интерпретация последовательности машинных инструкций, выполненных за время работы программы. Рассматриваются способы увеличения точности анализа за счёт анализа циклов работы со строками, а также предварительного расширения покрытия кода. С помощью инструмента, разработанного на основе предложенных методов, были найдены как известные ошибки, так и ошибки, информация о которых ранее не была опубликована.

Ключевые слова: поиск ошибок; бинарный код; динамический анализ; символьное выполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-8

Для цитирования: В.В. Каушан. Поиск ошибок выхода за границы буфера в бинарном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 135-144. DOI: 10.15514/ISPRAS-2016-28(5)-8

1. Введение

В настоящее время особенно остро стоит задача обеспечения безопасности информационных систем. Наиболее частой причиной нарушения безопасности в таких системах являются уязвимости в программном обеспечении этих систем, позволяющие нарушить конфиденциальность, доступность или целостность обрабатываемой информации. В связи с этим актуальной является задача поиска ошибок и уязвимостей в программном обеспечении.

Одним из наиболее распространённых типов уязвимостей является уязвимость переполнения буфера, уступающая по распространённости лишь XSS и SQL-инъекциям, которые присущи веб-приложениям [1]. Эксплуатация этого типа уязвимости во многих случаях позволяет выполнить произвольный код в

* Работа поддержана грантом РФФИ № 16-29-09632

рамках уязвимого приложения и, таким образом, скомпрометировать систему. Кроме того, уязвимость "Heartbleed" [2] в OpenSSL продемонстрировала, что большую опасность может представлять не только переполнение буфера (при записи данных), но и выход за границы буфера (при чтении данных). Таким образом, поиск ошибок работы с буферами в памяти является актуальной задачей.

2. Поиск ошибок с помощью символьной интерпретации

Предложенный метод поиска ошибок основан на анализе результатов наблюдения за выполнением программы. Такой анализ может быть проведен с помощью отладчиков, систем post-mortem анализа трасс выполнения [3-5], а также систем динамического двоичного инструментирования (DBI) таких как Pin [6] и Valgrind [7]. Анализируется последовательность машинных инструкций, которые выполнялись в рамках одного запуска исследуемой программы. Анализ на уровне машинных инструкций позволяет находить ошибки в программах в тех случаях, когда исходный код программы или отладочная информация недоступны.

В качестве основного механизма поиска ошибок используется символьная интерпретация последовательности машинных инструкций. Каждая инструкция транслируется в уравнения для SMT решателя [8] относительно входных данных программы. Полученный набор уравнений для некоторого пути в программе, называемый предикатом пути, отражает прохождение программы по этому пути. К этим уравнениям добавляются уравнения (называемые предикатом безопасности), описывающие ошибочную ситуацию в программе в некоторый момент её выполнения. Если полученная система уравнений оказалась совместной - её решением будет набор входных данных, приводящих к проявлению ошибки.

Для поиска ошибок выхода за границы буфера дополнительно вводится понятие символьной длины буферов - дополнительные переменные, связанные с длинами соответствующих буферов. С помощью оставления уравнений над этими переменными возможен поиск ситуаций выхода за границы буфера как при единичном чтении, так и при последовательном доступе к памяти (например, при копировании или вычислении длины строк). Такой подход позволяет абстрагироваться от реальных размеров данных, обрабатываемых программой, и таким образом находить ошибки работы с памятью, когда размер обрабатываемых данных потенциально может превышать размер буфера, выделенного под эти данные. Кроме того, символьная интерпретация отдельных инструкций дополняется интерпретацией укрупнённых блоков программы, таких как вызовы библиотечных функций и экземпляры циклов программы. Правила интерпретации подробно описаны в работе [9].

Данная работа является продолжением работы [9] и предлагает методы увеличения точности анализа с помощью анализа циклов работы со строками,

а также с помощью предварительного расширения покрытия кода анализируемой программы.

3. Анализ циклов

Одной из важных задач, возникающих во время анализа программ, является анализ циклов. Ошибка в программе может проявиться в результате выполнения большого числа итераций цикла, для обнаружения такой ситуации необходимо иметь возможность вычисления числа итераций цикла, при котором достигается ошибочная ситуация. Кроме того, во многих случаях, число итераций цикла зависит непосредственно от обрабатываемых данных и такие циклы в случае символьной интерпретации должны анализироваться, как если бы число итераций цикла было бы произвольным.

В рамках задачи динамического анализа кода проблема анализа циклов имеет свою специфику: последовательность выполненных инструкций уже содержит выполнение некоторого количества итераций цикла, а символьная интерпретация инструкций этих итераций накладывает ограничения на входные данные, неявно фиксирующие число итераций цикла. Для того чтобы абстрагироваться от числа итераций цикла, необходимо рассматривать этот цикл как единое целое по аналогии с библиотечными функциями с известной семантикой.

В рамках предлагаемого метода выполняется анализ циклов работы со строками, таких как циклы копирования строк и циклы вычисления длины строки. Во время анализа происходит обнаружение таких циклов, извлекаются параметры соответствующих им операций (адреса и длины строк), после чего такие циклы интерпретируются как единое целое по аналогии с вызовами функций strcpy и strlen. Циклы, которые не были обнаружены в процессе анализа, специальным образом не обрабатываются.

Обнаружение циклов работы со строками происходит в несколько этапов. На первом этапе для каждого цикла определяется множество индуктивных переменных и характер их изменения. Под переменной понимается регистр или ячейка памяти с постоянным адресом, связанная с конкретной инструкцией в коде программы. Для таких переменных анализируются значения переменной на последовательных итерациях цикла. На основе анализа нескольких итераций цикла делается вывод о характере изменения значения переменной. Если значение переменной изменяется линейно с одним и тем же шагом на всех итерациях цикла, переменная включается в множество индуктивных переменных цикла. Далее для каждого цикла определяется множество буферов в памяти, соответствующих следующим критериям:

- переменная, используемая в качестве адреса ячейки памяти является индуктивной;
- на каждой итерации цикла происходит обращение к последовательным адресам памяти;

- размеры обрабатываемых ячеек соответствуют размеру символа для одного из типов строк (1 или 2 байта) и равны шагу переменной, используемой в качестве адреса;
- данные буфера представляют собой нуль-терминированную строку.

На этом этапе фиксируется характер доступа к каждому буферу (чтение и/или запись), его размер, а также размер адресуемой ячейки.

На основе полученных данных производится классификация циклов на принадлежность к одному из видов.

Цикл считается циклом копирования строк, если выполнены следующие критерии:

- существует два буфера, к которым осуществляется доступ на каждой итерации цикла;
- к первому буферу обращаются только на чтение;
- ко второму буферу обращаются только на запись;
- на каждой итерации цикла значения, прочитанные из первого буфера, и значения, записанные во второй, совпадают.

Цикл считается циклом вычисления длины строки, если выполнены следующие критерии:

- существует один буфер, к которому осуществляется доступ на каждой итерации цикла;
- к буферу обращаются только на чтение;
- значение одной из индуктивных переменных после последней итерации цикла равно фактической длине строки, находящейся в буфере.

Все остальные циклы классифицируются как циклы с неизвестной семантикой. Если в цикле происходит одновременно копирование строки и вычисление её длины, то циклу присваиваются оба класса. В дальнейшем, различные семантики такого цикла обрабатываются независимо друг от друга.

Циклы копирования и вычисления длины строк обрабатываются аналогично вызовам функций strcpy и strlen. Для каждого такого цикла определяются его границы и описываются значения фактических параметров, как если бы цикл являлся вызовом соответствующей функции. Такой подход позволяет единообразно обрабатывать как вызовы строковых функций, так и эквивалентные им циклы.

4. Расширение покрытия кода

Недостатком анализа последовательности инструкций для одного запуска программы является отсутствие возможности итеративного анализа путей выполнения. Этот недостаток можно компенсировать возможностью анализа нескольких запусков исследуемой программы. Входные данные для таких запусков подбираются так, чтобы максимизировать суммарное покрытие кода

исследуемой программы, что приводит к увеличению вероятности обнаружения ошибки.

Предлагаемый метод получения такого набора входных данных, максимизирующих суммарное покрытие кода, основан на динамическом онлайн символьном выполнении. В процессе символьного выполнения назначаются символьные значения заданным переменным программы и эти символьные значения распространяются по мере выполнения программы, при этом все преобразования в программе, в которых участвуют символьные значения, транслируются в соответствующие уравнения. Для хранения символьных значений переменных программы поддерживается *состояние*, описывающее отображение множества переменных программы на множество соответствующих им символьных значений, а также выполняемую на данный момент операцию программы. Если в процессе выполнения встречается ветвление, зависящее от символьных данных, порождается два состояния, соответствующие двум веткам ветвления, и выполнение продолжается дальше для каждого из состояний.

Обычно символьные значения назначают ячейкам памяти, содержащим входные данные программы. В этом случае, различные состояния соответствуют различным путям выполнения в программе при обработке входных данных. С помощью SMT-решателя для каждого состояния и соответствующего ему пути можно получить подтверждающий набор входных данных. Кроме того, если производится выполнение бинарного кода, часто можно получить покрытие кода в терминах базовых блоках. Анализ покрытия кода для каждого из состояний позволяет получить набор входных данных, для которого, в совокупности, достигается существенный прирост покрытия кода по сравнению с единичным запуском программы.

Пути выполнения, соответствующие порождаемым состояниям, представляют собой древовидную структуру. Это приводит к тому, что покрытие кода, соответствующее двум соседним состояниям отличается незначительно. Кроме того, с каждым ветвлением количество состояний в программе растёт по экспоненциальному закону, что приводит к огромному количеству анализируемых состояний и неэффективности анализа соответствующих запусков. Для решения этой проблемы можно выделить такое подмножество состояний, суммарное покрытие кода для которого будет таким же, как и для всего множества состояний. Это возможно сделать с помощью классической задачи о покрытии множества. Задача о покрытии относится к классу NP-полных, но может быть эффективно решена приближённым алгоритмом, который даёт приемлемый результат. Таким способом на практике удаётся уменьшить количество рассматриваемых состояний на несколько порядков.

Для оценки покрытия используется метрика покрытия кода по базовым блокам, так как использование этой метрики оказывается достаточным для поиска ошибок, привязанных к конкретному месту программы. В большинстве случаев выход за границы буфера происходит из-за отсутствия

проверок на размер буфера непосредственно перед операцией копирования. Для поиска таких ошибок требуется покрыть как можно большее количество мест в программе, в которых происходит копирование данных, что, в свою очередь, можно свести к задаче получения хорошего покрытия в терминах базовых блоков.

Поскольку, в общем случае, процесс перебора путей выполнения может выполняться неопределённо долго, требуется критерий завершения процесса перебора путей. Таким критерием может быть оценка прироста покрытия за определённый период времени. Если за заданный интервал времени прирост покрытия в терминах базовых блоков оказался меньше, чем некоторое пороговое значение, перебор путей завершается.

Для реализации описанного метода использовался инструмент S2E [10], предоставляющий возможности динамического онлайн символьного выполнения в рамках полносистемного эмулятора QEMU. В качестве входных данных для этого инструмента выступает исследуемая программа и начальный набор входных данных. В процессе работы S2E выполняется перебор путей при помощи инвертирования условных переходов, зависящих от символьных данных. В результате перебора путей в программе для каждого завершённого пути формируется набор входных данных, а также достигаемое покрытие кода на этом пути. Далее количество таких путей минимизируется с целью получения минимального набора входных данных, на котором достигалось бы такое же покрытие, как и на полном наборе входных данных. Полученный набор входных данных далее используется непосредственно для анализа каждого из запусков.

5. Апробация

Предлагаемый метод поиска ошибок был реализован и апробирован на программах, работающих под ОС Linux и Windows, а также на программном обеспечении для маршрутизатора, выпускаемого одним из крупных производителей сетевого оборудования. Были найдены ошибки, связанные как с записью, так и с чтением данных за пределами буфера. С помощью метода расширения покрытия удавалось получить прирост покрытия до 25% от такового для начального набора входных данных. Кроме того, с помощью расширения покрытия для приложения mkfs.jfs был автоматически восстановлен список поддерживаемых аргументов командной строки. В таблице 1 приведены наиболее показательные результаты применения предложенного метода поиска ошибок. Время анализа для большинства примеров не превышало нескольких минут. Следует отметить, что анализу предшествует стадия расширения покрытия кода (0.5-4 часа). В столбце "исходный размер данных" указан размер данных, который подавался на вход программе во время анализа и не приводил к ошибке, а в столбце "конечный размер данных" – размер данных, приводящий к ошибке работы с памятью. Для каждого приложения было обработано только первое срабатывание

инструмента, в общем случае возможно получение всех срабатываний ценой незначительного увеличения времени анализа. Также следует отметить, что случаи ложноположительных и ложноотрицательных срабатываний инструмента не встречались. Запуск инструментов проводился на машине с конфигурацией Intel Xeon E5-2650 v2, 32Gb RAM, 500Gb HDD, на всех этапах было задействовано только одно ядро процессора.

Табл. 1. Результаты применения метода.

Table 1. Method approbation results.

ОС	Программа	Размер данных		Размещение буфера	CVE / OSVDB
		исходный	конечный		
Linux	openssl	18	25	куча	CVE: 2014-0160
Linux	alsa_out	14	95	стек	-
Linux	mkfs.jfs	31	436	стек	-
Linux	prepmx	11	813	стек	-
WinXP SP2	httpdx	329	330	куча	OSVDB-ID: 84454
прошивка маршрути-затора	pptp	16	257	стек	-

6. Заключение

В статье представлен метод поиска ошибок выхода за границы буфера в бинарном коде программ. В частности, рассматривается метод анализа циклов, а также метод расширения покрытия кода, позволяющие улучшить точность при поиске ошибок и потенциально уменьшить количество ложноотрицательных срабатываний инструмента. Методы были реализованы в виде программных инструментов, работающих в рамках инструмента динамического анализа бинарного кода, а также среды динамического символьного выполнения S2E.

Список литературы

[1]. Younan Y. 25 Years of Vulnerabilities: 1988-2012. Sourcefire Vulnerability Research Team. – 2013.

[2]. CVE-2014-0160: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[3]. 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 ИСП РАН, том 26, вып. 1, 2014 г., стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8

[4]. 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.

[5]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды ИСП РАН, том 26, вып. 1, стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9

[6]. Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. Dynamic Program Analysis of Microsoft Windows Applications. International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.

[7]. Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge, 2004.

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

[9]. В.В. Каушан, А.Ю. Мамонтов, В.А. Падарян, А.Н. Федотов. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ. Труды ИСП РАН, том 27, вып. 2, 2015, стр. 105-126. DOI: 10.15514/ISPRAS-2015-27(2)-7

[10]. Chipounov V., Kuznetsov V., Candea G. 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.

Buffer overrun detection method in binary code★

V.V. Kaushan <korpse@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Buffer overflows are one of the most common and dangerous software errors. Exploitation of such errors can lead to an arbitrary code execution and system disclosure. This paper considers a method for detecting memory violations. The method is based on combined (static-dynamic) analysis of binary code. Analysis is based on symbolic interpretation of machine instructions executed during a single program run. Proposed method also provides abstraction from buffer sizes and can reveal sizes that cause buffer overflow errors. Analysis can be applied to program binaries and doesn't require a source code. Two techniques are proposed to improve method precision: cycle analysis and code coverage increase. Cycle analysis is one of the cumbersome problems in dynamic analysis. Separate cycle instruction analysis leads to an excess of constraints over input data that causes potential false negatives. The proposed technique is able to analyze cycles entirely and abstract from number of cycle iterations. One of the drawbacks of a single run analysis is an insufficient code coverage which prevents some errors from discovery. The technique proposed to increase code coverage is based on a dynamic symbolic execution. Some minimal path set from discovered code paths is selected and used to achieve better code coverage than from a single run. Inputs corresponding to each path from selected set are used to analyze several program runs. Proposed techniques were implemented and used to discover both known and non-disclosed bugs.

Keywords: bug finding; binary code; dynamic analysis; symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(5)-8

For citation: V.V. Kaushan. Buffer overrun detection method in binary code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 135-144 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-8

References

[1]. Younan Y. 25 Years of Vulnerabilities: 1988-2012 Sourcefire Vulnerability Research Team. – 2013.
[2]. CVE-2014-0160 https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160
[3]. Padaryan V. A., Getman A. I., Solovyev M. A., Bakulin M. G., Borzilov A. I., Kaushan V. V., Ledovskikh I. N., Markin Yu. V., Panasenko S. S. Methods and software

★ The paper is supported by RFBR grant 16-29-09632

tools to support combined binary code analysis. Programming and Computer Software. September 2014, Volume 40, Issue 5, pp 276-287.
[4]. Dovgalyuk P.M., Fursova N.I., Dmitriev D.S. Prospects of using virtual machine deterministic replay insolving computer security problems. The Proceedings RusCrypto'2013, 2014 (In Russian).
[5]. Dovgalyuk P.M., Makarov V.A., Romaneev M.S., Fursova N.I. [Applying program emulators for binary code analysis]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.
[6]. Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. Dynamic Program Analysis of Microsoft Windows Applications. International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.
[7]. Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge, 2004.
[8]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. Proceedings of PDPAR'03, July 2003
[9]. Kaushan V.V., Mamontov A.Yu., Padaryan V.A., Fedotov A.N. [Memory violation detection method in binary code]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 2, 2015, pp. 105-126 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-7
[10]. Chipounov V., Kuznetsov V., Candea G. 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.