Построение предикатов безопасности для некоторых типов программных дефектов*

[1] *А.Н. Федотов <fedotoff@ispras.ru>*
[1] *В.В. Каушан <korpse@ispras.ru>*
[1,2,3,4] *С.С. Гайсарян <ssg@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *2119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК*
[3] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. В статье рассматриваются подходы и способы выполнения кода с использованием уязвимостей в программах. В частности, рассмотрены способы выполнения кода для переполнения буфера на стеке и в динамической памяти, для уязвимости использования памяти после освобождения и для уязвимости форматной строки. Описываются методы и подходы, позволяющие автоматически получать наборы входных данных, которые приводят к выполнению произвольного кода. В основе этих подходов лежит использование символьной интерпретации. Динамическое символьное выполнение предоставляет набор входных данных, который направляет программу по пути активации уязвимости. Предикат безопасности представляет собой дополнительные символьные уравнения и неравенства, описывающие требуемое состояние программы, наступающее при обработке пакета данных, например, передача управления на требуемый адрес. Объединив предикаты пути и безопасности, а затем решив полученную систему уравнений, можно получить набор входных данных, приводящий программу к выполнению кода. В работе представлены предикаты безопасности для перезаписи указателя, перезаписи указателя на функцию и уязвимости форматной строки, которая приводит к переполнению буфера на стеке. Описанные предикаты безопасности использовались в методе оценки критичности программных дефектов. Проверка работоспособности предикатов безопасности оценивалась на наборе тестов, который использовался в конкурсе *Darpa Cyber Grand Challenge.* Тестирование предиката безопасности для уязвимости форматной строки, приводящей к переполнению буфера, проводилось на программе Ollydbg, содержащей эту уязвимость. Для некоторых примеров удалось получить входные данные, приводящие к выполнению кода, что подтверждает работоспособность предикатов безопасности.

Ключевые слова: ошибки; символьное выполнение; предикат безопасности; анализ бинарного кода; динамический анализ.

DOI: 10.15514/ISPRAS-2017-29(6)-8

Для цитирования: Федотов А.Н., Каушан В.В., Гайсарян С.С., Курмангалеев Ш.Ф. Построение предикатов безопасности для некоторых типов программных дефектов. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 151-162. DOI: 10.15514/ISPRAS-2017-29(6)-8

* Работа поддержана грантом РФФИ № 17-01-00600 А

1. Введение

В современном мире безопасности программного обеспечения уделяют большое внимание. Важным аспектом обеспечения безопасности является поиск ошибок и уязвимостей. Поиск ошибок и уязвимостей может осуществляться на разных стадиях жизненного цикла программного обеспечения: в виде стороннего аудита во время эксплуатации ПО или на стадии разработки. В настоящие время применение безопасного цикла разработки (SDLC) становится неотъемлемой частью при создании программ промышленного уровня. ГОСТ Р 56939-2016 «Разработка безопасного программного обеспечения. Общие требования», регламентирует требования к разработке такого ПО. В соответствии с ГОСТ разработчик должен производить: моделирование угроз информационной безопасности, статический анализ кода, экспертизу исходного кода, функциональное тестирование программы, тестирование на проникновение, динамический анализ кода программы, фаззинг-тестирование программы.

Среди множества уязвимостей наиболее опасными являются уязвимости, позволяющие атакующему выполнить произвольный код [1]. Такие уязвимости достаточно сложно обнаружить автоматическими средствами, и поэтому довольно часто приходится привлекать аналитика. Кроме этого, возникает дополнительная сложность в оценке степени критичности проявления уязвимости.

Как правило, для активации уязвимости требуется специально сформированный набор входных данных, при обработке которого, программа перейдёт в состояние проявления этой уязвимости. Для автоматического формирования таких наборов входных данных, прежде всего, необходимо изучить и формализовать действия аналитика. На сегодняшний день существуют подходы и программные средства, позволяющие для некоторых типов уязвимостей получать такие входные данные [2, 4]. Эти подходы основываются на использовании динамического символьного выполнения. Применение этой техники обусловлено несколькими аспектами. Для успешной передачи управления и выполнения произвольного кода, необходимо обладать

сведениями о том, какие ячейки памяти и какие регистры процессора находятся под влиянием входных данных в момент активации уязвимости. Кроме того, в результате динамического выполнения формируются ограничения, описывающие путь в программе, на котором может произойти активация уязвимости (предикат пути). Добавив необходимые условия, требуемые для выполнения кода (предикат безопасности), и затем, решив полученный набор ограничений, можно получить искомый набор входных данных. Существующее системы имеют в своём арсенале набор предикатов безопасности для некоторых способов эксплуатации уязвимостей переполнения буфера на стеке и куче, а также уязвимости форматной строки [3-5]. В данной работе представлен расширенный набор предикатов безопасности для таких уязвимостей, как: переполнение буфера на стеке и куче, уязвимость форматной строки и использование памяти после освобождения. Разработанные предикаты применялись для оценки критичности программных дефектов [6,7].

Статья организована следующим образом. Во втором разделе приводится обзор некоторых типов дефектов программного обеспечения. В третьем разделе описаны методы и инструменты используемые для оценки критичности программных дефектов. В четвёртом разделе предложены разработанные предикаты безопасности. В пятом разделе рассматриваются результаты применения разработанных предикатов безопасности. В заключении обсуждаются полученные результаты и дальнейшие направления исследований.

2. Обзор критических программных дефектов

Перехват потока управления программы или выполнение произвольного кода в контексте программы является одной из наиболее опасных атак, а дефекты, срабатывания которых могут привести к реализации подобной атаки будем называть критическими. Для выполнения кода необходимо решить следующие задачи:

- разместить целевой код в одном из контролируемых буферов памяти;
- передать управление на адрес буфера с кодом.

Выполнение этих задач зависит от типа уязвимостей, а также состояния программы при котором она проявляется.

Переполнение буфера на стеке. Возможность выполнения произвольного кода при возникновении переполнения буфера на стеке известна довольно давно и описана в статье [8]. В результате переполнения буфера под контролем атакующего могут оказаться данные, манипуляция с которыми может позволить ему передать управление на свой код. Эти данные можно разделить на два типа: служебные данные и данные программы. К служебным данным относятся, например, адрес возврата из функции и указатель на кадр стека. К данным программы можно отнести указатели, в том числе указатели на функции. На рис. 1 представлена организация стека и способы атаки, результатом которой является выполнение кода.

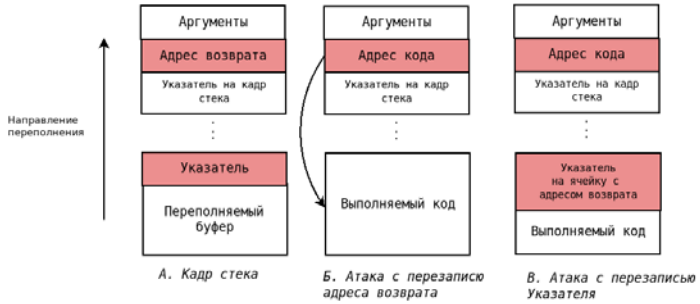

Рис. 1. Организация стека и способы атак для уязвимости переполнения буфера на стеке

Fig. 1. The organization of the stack and methods of attack for the buffer overflow vulnerability on the stack

Наиболее распространённой является атака, приведённая на Рис. 1Б. Атакующий размещает в переполняемом буфере свой код, а адрес возврата подменяет адресом этого буфера. Для данной атаки уже описаны предикаты безопасности, и они успешно реализованы в инструментах [9,10]. В случае, когда на стеке находится указатель на какую-либо переменную, то его можно перезаписать значением адреса ячейки, содержащей адрес возврата из функции. Для успешной реализации атаки необходимо, чтобы в дальнейшем по этому указателю произошла запись контролируемого атакующим значения. В этом случае возможно подменить записываемое значение на адрес буфера, в котором находится его код, и таким образом произойдёт перезапись адреса возврата из функции. Возврат из функции приведёт к передаче управления на код атакующего. Частный случай − перезапись указателя на функцию. В этом случае нарушителю достаточно перезаписать значение указателя адресом буфера с кодом и после вызова функции по указателю произойдёт передача управления на код полезной нагрузки.

Переполнение буфера на куче. В некоторых менеджерах динамической памяти в ОС Linux, основанных на менеджере памяти Doug Lea's Malloc (dlmalloc) существуют возможности для проведения атак, приводящих к выполнению кода. Одной из первых техник эксплуатация была техника *unlink***,** описанная в работе [11]. В результате переполнения буфера портятся служебные данные выделенных объектов в динамической памяти и после вызова функции освобождения может возникнуть ситуация, при которой атакующий может писать произвольное значение по произвольному адресу (CWE-123) [12], передача управления и выполнение кода здесь осуществляется подобно ситуации с перезаписью указателя. Таким образом, подавляющее большинство уязвимостей переполнения буфера на куче приводит к выполнению произвольного кода, но в 2003 году было разработано исправление, препятствующее возникновению ситуации CWE-123 путем аварийного завершения программы. В работе [13] описаны различные способы,

позволяющее эксплуатировать уязвимость переполнения буфера на куче, некоторые из них работают в современных версиях менеджеров памяти. Эти способы требуют соблюдения большого числа условий, и в открытом доступе находятся примеры эксплуатации модельных программ. Как и для переполнения буфера на стеке, остаются доступными способы эксплуатации, основанные на перезаписи указателя.

Использование памяти после освобождения. Один из способов проведения атаки с использованием памяти после освобождения применяется к программам, которые написаны на языке Си++ и содержат вызовов виртуального метода. Для проведения успешной атаки должны выполниться следующие условия:

- на куче выделен объект, содержащий виртуальный метод;
- объект удаляется (освобождается память, выделенная под этот объект);
- выделяется новый блок памяти, таким образом, что он пересекается с ранее выделенным объектом;
- в новый блок записываются данные, контролируемые атакующим;
- вызов виртуального метода, ранее выделенного объекта (факт использования памяти после освобождения).

В самом начале области памяти объекта, первые 4(8) байт, содержат указатель на таблицу виртуальных функций. Таким образом, у атакующего появляется возможность перезаписать указатель на свою, специально сформированную таблицу виртуальных функций. В результате вызова метода из этой таблицы произойдёт передача управления на код атакующего. На Рис. 2 представлен фрагмент кода на языке ассемблера, осуществляющего вызов виртуального метода.

```
MOV EAX, DWORD PTR[EBP - 0x1C]
    MOV EAX, DWORD PTR[EAX]
     CALL DWORD PTR[EAX+8]
```

Рис. 2. Вызов виртуального метода
Fig. 2. Invoking the virtual method

Первая инструкция загружает указатель на объект в регистр *EAX*. Вторая инструкция загружает на регистр указатель на таблицу виртуальных функций. Третья инструкция производит вызов виртуальной функции из таблицы.

Уязвимость форматной строки. Способы выполнения кода, используя уязвимость форматной строки известны и описаны в работах [1, 14]. Кроме того, есть методы, позволяющие это делать автоматически [2-4, 15]. Уязвимость существует в случае, если атакующий контролирует строку формата. Манипулируя спецификаторами в строке формата атакующий может передать управление на свой код. Спецификатор *%n* используется для записи количества

выведенных символов по заданному в строке формата адресу. Таким образом, атакующей может перезаписать, например, адрес возврата из функции адресом буфера со своим кодом. В некоторых функциях работы с форматной строкой количество выводимых символов ограничено максимальным значением 32-битного беззнакового целого числа. В связи с этим возникает проблема при эксплуатации уязвимостей в программах, работающих под управлением 64-разрядных операционных систем. В этом случае, возможна только частичная перезапись ячейки памяти, содержащий адрес. Существует ещё один способ выполнения произвольного кода, используя уязвимость форматной строки. Для этого способа подходят функции, которые печатают строку не на стандартный поток вывода, а в буфер, располагающийся, например, на стеке. Манипулируя спецификаторами строки формата, атакующий может переполнить буфер и перезаписать адрес возврата, что в последствии приведёт к выполнению кода.

3 Методы генерации критических входных данных

Под критическими входными данными будем понимать входные данные, которые приводят программу либо к аварийному завершению, либо к выполнению заданного кода нарушителя. Наиболее результативным методом генерации критических входных данных является динамический анализ, который можно рассматривать как совокупное применение метода фаззинга для автоматической генерации тестовых наборов и поиска входных данных приводящих к аварийному завершению программы, технологий динамической бинарной трансляции и/или инструментации для получения трасс выполнения и информации о покрытии кода, конкретного и символьного выполнения для анализа предикатов безопасности, генерации входных данных обеспечивающих отказ в обслуживании или перехват потока управления.

Традиционно упомянутые средства применяются в следующей последовательности: фаззинг для получения входных данных приводящих к воспроизводимому аварийному завершению, снятие трассы выполнения на найденных данных и трансляции ее в символьную форму. Затем совместно решая полученную систему уравнений и предикат безопасности описывающий необходимые условия для эксплуатации ошибки, получаем данные, которые необходимо подать на вход программе. Полученные данные, реализуют условия, описываемые в предикате безопасности, например, перехват потока управления.

Различают два основных подхода к такому анализу: полносистемный и уровня приложения.

Полносистемный подход к анализу. В этом случае для получения трассы выполнения программы используется полносистемный эмулятор, например, Qemu [16] данный подход применятся в таких платформах как S2E [17], PANDA [18], Avatar [19] и других. Положительная сторона такого подхода заключается в обеспечении возможности детерминированного воспроизведения в рамках системы и полноте трассы выполнения, включающей в себя как код

анализируемой программы, так и системный код. Минусом является скорость работы и требования к системным ресурсам.

Анализ на уровне приложения. При анализе на уровне приложения, для трассировки используется динамическая бинарная трансляция или инструментация кода программы, при этом в трассу выполнения попадают только инструкции, относящиеся к коду исполняющемся в пользовательском пространстве целевого процесса. К подобным системам относятся Angr[5], Mayhem [3], Manticore [20] и д.р.

4 Построение предикатов безопасности

При построении предикатов безопасности для перехвата потока управления требуется описать размещение кода полезной нагрузки и передачу управления на этот код.

Построение предиката безопасности для перезаписи указателя (CWE-123):

- в контролируемом буфере памяти располагается код полезной нагрузки;
- адрес указателя равен адресу ячейки памяти, содержащий адрес возврата из функции;
- значение указателя равно адресу буфера с кодом полезной нагрузки.

Построение предиката безопасности для перезаписи указателя на функцию:

- в контролируемом буфере памяти располагается код полезной нагрузки;
- значение указателя на функцию равно адресу буфера с кодом полезной нагрузки.

Построение предиката безопасности для уязвимости форматной строки, которая приводит к переполнению буфера на стеке:

- в контролируемом буфере памяти располагается код полезной нагрузки;
- форматная строка составлена таким образом, что её обработка приводит к переполнению буфера на стеке и перезаписывает адрес возврата.

При составлении форматной строки для вывода значений используется форматный спецификатор *%x*. Как правило, этот спецификатор используется с параметром ширины и имеет вид *%8x*. Присутствие в форматной строке необходимого количества таких спецификаторов вызовет переполнение буфера.

5. Применение предикатов безопасности

Для многих областей программирования, таких как компиляторные технологии, машинное обучение и др. существуют тестовые наборы, на которых можно оценить эффективность подходов. В качестве примера можно привести наборы

SPEC для тестирования компиляторов и «MNIST база изображений рукописных цифр», наборы Juliet от NIST для тестирования статических анализаторов. Однако для области компьютерной безопасности, в частности комплексного тестирования средств, входящих в процесс безопасной разработки ПО такого набора не было. Более того, все средства, как правило, тестировались на различных наборах программы и платформ, что затрудняло качественную оценку систем.

В августе 2016 года прошел финал DARPA Cyber Grand Challenge[21] – соревнования по построению автономной системы поиска дефекта в программе и генерации критических входных данных для найденных уязвимостей и генерации исправлений для найденных дефектов. Победителем, в котором стала система Mayhem. Организаторами соревнования был сформирован набор программ, содержащих уязвимости. Программы различаются по сложности, функционалу и типам уязвимости. В качестве единой платформы используется основанный на Linux дистрибутив, названный DECREE OS. После окончания соревнований данный тестовый набор был перенесен на другие ОС [22].

Разработанные предикаты безопасности применялись в методе оценки критичности программных дефектов [6]. Тестирование предикатов безопасности для переполнения буфера на стеке и уязвимости использования памяти после освобождения проводилось на нескольких программах из тестового набора [22]. Этот тестовый набор состоит из 241 исполняемого файла. Данные, приводящие к аварийному завершению, были получены аналитиком для 11 программ. Используя метод предварительной фильтрации аварийных завершений из работы [7], были выбраны 8 аварийных завершений, как наиболее опасные. Для 6 из них удалось получить входные данные, приводящие к выполнение заданного кода. Ниже приводится список этих программ:

- Movie_Rental_Service;
- Multi_User_Calendar;
- Palindrome;
- PKK_Steganography;
- Sample_Shipgame;
- ValveChecks.

При анализе программы *Movie_Rental_Service* применялось построение предиката безопасности для уязвимости использования памяти после освобождения. При анализе остальных программ применялся предикат безопасности для перезаписи адреса возврата из функции во время переполнения буфера на стеке. Для программ *Bloomy_Sunday* и *Charter* не удалось получить входные данные, приводящие к выполнению кода. В результате анализа *Bloomy_Sunday* были получены входные данные, при которых выполнение заданного кода не произошло. Такая ситуация может быть связана с тем, что в результате анализа помеченных данных в предикат пути не

попали ограничения, связанные адресными зависимостями. При анализе программы *Charter* выяснилось, что при переполнении буфера переписывается указатель, но в дальнейшем записи контролируемых данных по этому указателю не происходит. Таким образом, применить предикат безопасности для перезаписи указателя не удалось.

Тестирование предиката безопасности для уязвимости форматной строки, приводящей к переполнению буфера на стеке, производилось на программе OllyDbg. Описание уязвимости приводится в [23]. Для этой программы удалось получить входные данные, приводящие к выполнению заданного кода.

6. Заключение

В статье рассмотрены способы выполнения кода, используя уязвимости в программах. Описаны современные методы и средства, позволяющие решать эту задачу автоматически. Представлены новые предикаты безопасности для перезаписи указателя, перезаписи указателя на функцию и уязвимости форматной строки, которая приводит к переполнению буфера на стеке. Разработанные предикаты безопасности применялись в методе оценки критичности программных дефектов [6]. Тестирование предикатов безопасности проводилось на наборе тестов для *Darpa Cyber Grand Challenge,* а также на программе Ollydbg, содержащей уязвимость форматной строки. В результате тестирования удалось получить входные данные, приводящие к выполнению кода.

В качестве дальнейшего направление развития можно выделить разработку предикатов безопасности, которые учитывают современные защитные механизмы, препятствующие перехвату потока выполнения. Эти механизмы можно разделить на две группы: механизмы защиты операционной системы (DEP и ASLR), и защитные механизмы, встраиваемые компилятором ("канарейка" и безопасные функции работы со строками). Эти механизмы повсеместно применяются в современных операционных системах общего назначения, а опции компилятора, обеспечивающие встраивание защит, как правило, включены в опции компиляции по умолчанию. Тем не менее, иногда защиты намеренно отключаются разработчиками, а в некоторых случаях эти защиты можно обойти.

Список литературы

[1]. C. Anley, J. Heasman, F. Lindner, G. Richarte. The shellcoder’s handbook: discovering and exploiting security holes. John Wiley & Sons, 2011, 61 c.

[2]. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. Automatic exploit generation. Communications of the ACM, vol. 57, no. 2, 2014, pp. 74–84.

[3]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley. Unleashing mayhem on binary code. 2012 IEEE Symposium on Security and Privacy (SP), 2012, pp. 380–394.

[4]. Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software 2012 IEEE Sixth International Conference on Security and Reliability (SERE), 2012, pp. 78–87.

[5]. Y Shoshitaishvili, R Wang, C Salls et al. The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy (S&P), 2016, pp. 138-157.

[6]. А.Н. Федотов, В.А. Падарян, В.В. Каушан и др. Оценка критичности программных дефектов в условиях работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 73–92. DOI: 10.15514/ISPRAS-2016-28(5)-4

[7]. А.Н Федотов. Метод оценки эксплуатируемости программных дефектов. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 137-148. 10.15514/ISPRAS-2016-28(4)-8

[8]. One Aleph. Smashing The Stack For Fun And Profit. 1996. URL: http://phrack.org/issues/49/14.html#article (дата обращения: 08.10.2017).

[9]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. Труды ИСП РАН, том 26, вып. 3, 2914 г., стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7

[10]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, vol. 41, № 6, pp. 373–380. DOI: 10.1134/S0361768815060055

[11]. Once upon a free(). 2001. URL: http://phrack.org/issues/57/9.html#article (дата обращения: 08.10.2017).

[12]. CWE-123. URL: https://cwe.mitre.org/data/definitions/123.html (дата обращения:12.05.2017).

[13]. Malloc Des-Maleficarum. 2009. URL: http://phrack.org/issues/66/10.html (дата обращения: 08.10.2017).

[14]. gera. Advances in format string exploitation. 2002. URL: http://phrack.org/issues/59/7.html#article (дата обращения: 08.10.2017).

[15]. И.А. Вахрушев, В.В. Каушан, В.А. Падарян, А.Н. Федотов. Метод поиска уязвимости форматной строки. Труды ИСП РАН, том 27, вып 4, 2015 г., стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(4)-2

[16]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.

[17]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, vol. 46, №. 3, 2011, pp. 265-278.

[18]. Dolan-Gavitt B. et al. Repeatable reverse engineering with PANDA. Proceedings of the 5th Program Protection and Reverse Engineering Workshop, ACM, 2015, p. 4.

[19]. Zaddach J. et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS Symposium 2014.

[20]. Manticore. URL: https://github.com/trailofbits/manticore (дата обращения: 08.10.2017).

[21]. Darpa Cyber Grand Challenge. URL: http://archive.darpa.mil/cybergrandchallenge/ (дата обращения: 08.10.2017).

[22]. Darpa Cyber Grand Challenge tests pack. URL:https://github.com/trailofbits/cb-multios (дата обращения: 08.10.2017).

[23]. Ollydbg bug. URL: https://www.exploit-db.com/exploits/388/ (дата обращения: 08.10.2017).

Building security predicates for some types of vulnerabilities

[1] *A.N. Fedotov <fedotoff@ispras.ru>*
[1] *V.V. Kaushan <korpse@ispras.ru>*
[1,2,3,4] *S.S. Gaissaryan <ssg@ispras.ru>*
[1] *Sh.F. Kurmangaleev <kursh@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics (HSE) 11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. Approaches for code execution using program vulnerabilities are considered in this paper. Particularly, ways of code execution using buffer overflow on stack and on heap, using use-after-free vulnerabilities and format string vulnerabilities are examined in section 2. Methods for automatic generation input data, leading to code execution are described in section 3. This methods are based on dynamic symbolic execution. Dynamic symbolic execution allows to gain input data, which leads program along the path of triggering vulnerability. The security predicate is an extra set of symbolic formulas, describing program's state in which code execution is possible. To get input data, leading to code execution, path and security predicates need to be united, and then the whole system should be solved. Security predicates for pointer overwrite, function pointer overwrite and format string vulnerability, that leads to stack buffer overflow are presented in the paper. Represented security predicates were used in method for software defect severity estimation. The method was applied to several binaries from Darpa Cyber Grand Challenge. Testing security predicate for format string vulnerability, that leads to buffer overflow was conducted on vulnerable version of Ollydbg. As a result of testing it was possible to obtain input data that leads to code execution.

Keywords: software bugs; symbolic execution; security predicates; binary analysis; dynamic analysis.

DOI: 10.15514/ISPRAS-2017-29(6)-8

For citation: Fedotov A.N., Kaushan V.V., Gaissaryan S.S., Kurmangaleev Sh.F. Building security predicates for some types of vulnerabilities. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 151-162 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-8

References

[1]. C. Anley, J. Heasman, F. Lindner, G. Richarte. The shellcoder's handbook: discovering and exploiting security holes. John Wiley & Sons, 2011, 61 pp.
[2]. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. Automatic exploit generation. Communications of the ACM, vol. 57, no. 2, 2014, pp. 74–84.

[3]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley. Unleashing mayhem on binary code. 2012 IEEE Symposium on Security and Privacy (SP), 2012, pp. 380–394.
[4]. Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software 2012 IEEE Sixth International Conference on Security and Reliability (SERE), 2012, pp. 78–87.
[5]. Y Shoshitaishvili, R Wang, C Salls et al. The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy (S&P), 2016, pp. 138-157.
[6]. A.N. Fedotov, V.A. Padarjan, V.V. Kaushan et al. Software defect severity estimation inpresence of modern defense mechanisms. Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 73–92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4
[7]. Fedotov A.N. Method for exploitability estimation of program bugs. Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 137-148 (in Russian). 10.15514/ISPRAS-2016-28(4)-8
[8]. One Aleph. Smashing The Stack For Fun And Profit. 1996. URL: http://phrack.org/issues/49/14.html#article (accessed 08.10.2017).
[9]. Padarjan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation method for stack buffer overflow vulnerabilities. Trudy ISP RAN / Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-7
[10]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, vol. 41, № 6, pp. 373–380. DOI: 10.1134/S0361768815060055
[11]. Once upon a free(). 2001. URL: http://phrack.org/issues/57/9.html#article (accessed 08.10.2017).
[12]. CWE-123. URL: https://cwe.mitre.org/data/definitions/123.html (accessed 12.05.2017).
[13]. Malloc Des-Maleficarum. 2009. URL: http://phrack.org/issues/66/10.html (accessed 08.10.2017).
[14]. gera. Advances in format string exploitation. 2002. URL: http://phrack.org/issues/59/7.html#article (accessed 08.10.2017).
[15]. I.A. Vahrushev, V.V. Kaushan, V.A. Padarjan, A.N. Fedotov. Search method for format string vulnerabilities. Trudy ISP RAN / Proc. ISP RAS, vol. 27, issue 4, 2015, pp. 23-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-2
[16]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.
[17]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, vol. 46, №. 3, 2011, pp. 265-278.
[18]. Dolan-Gavitt B. et al. Repeatable reverse engineering with PANDA. Proceedings of the 5th Program Protection and Reverse Engineering Workshop, ACM, 2015, p. 4.
[19]. Zaddach J. et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS Symposium 2014.
[20]. Manticore. URL: https://github.com/trailofbits/manticore (accessed 08.10.2017).
[21]. Darpa Cyber Grand Challenge. URL: http://archive.darpa.mil/cybergrandchallenge/ (accessed: 08.10.2017).
[22]. Darpa Cyber Grand Challenge tests pack. URL:https://github.com/trailofbits/cb-multios (accessed 08.10.2017).
[23]. Ollydbg bug. URL: https://www.exploit-db.com/exploits/388/ (accessed 08.10.2017).