Оценка критичности программных дефектов в условиях работы современных защитных механизмов*

[1]А.Н. Федотов <fedotoff@ispras.ru>
[1,2]В.А. Падарян <vartan@ispras.ru>
[1]В.В. Каушан <korpse@ispras.ru>
[1]Ш.Ф. Курмангалеев <kursh@ispras.ru>
[1]А.В. Вишняков <vishnya@ispras.ru>
[1]А.Р. Нурмухаметов <oleshka@ispras.ru>
[1]Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2]Московский государственный университет имени М.В. Ломоносова,
119991 ГСП-1, Москва, Ленинские горы

Аннотация. В данной работе предложен уточненный метод автоматизированной оценки степени опасности найденных программных дефектов. На этапе тестирования промышленного программного обеспечения выявляется значительное число дефектов, приводящих к аварийному завершению. Из-за ограниченности ресурсов исправление дефектов растягивается во времени и требует расстановки приоритетов. Основным критерием для назначения высокого приоритета становится возможность использования ошибки в злонамеренных целях. На практике эта задача решается путем автоматизированного построения входных данных, подтверждающего наличие опасной уязвимости. Но в известных публикациях по данной теме не учитывают современные защитные механизмы, препятствующие действиям атакующего, что снижает качество вырабатываемых оценок. В данной статье рассматриваются современные защитные механизмы и дается оценка их распространенности и эффективности. Метод применим к бинарным файлам и не требует какой-либо отладочной информации. В основе метода лежит символьная интерпретация трасс выполнения, полученных при помощи полносистемного эмулятора. Даже в условиях одновременного функционирования DEP, ASLR и защиты стека («канарейка») метод способен демонстрировать возможность использования ошибок, сочетающих условия «write-what-where» и переполнение буфера на стеке. Возможности реализованного метода были продемонстрированы на модельных примерах и реальных программах.

* Работа поддержана грантом РФФИ № 16-29-09632

A.N. Fedotov, V.A. Padaryan, V.V. Kaushan, Sh.F. Kurmangaleev, A.V. Vishnyakov, A.R. Nurmukhametov. Severity software defects estimation in presence of modern defense mechanisms. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 73-92

Ключевые слова: уязвимость; критический дефект; бинарный код; символьное выполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-4

Для цитирования: Федотов А.Н., Падарян В.А., Каушан В.В., Курмангалеев Ш.Ф., Вишняков А.В., Нурмухаметов А.Р. Оценка критичности программных дефектов в рамках работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, стр. 73-92, 2016 г. DOI: 10.15514/ISPRAS-2016-28(5)-4

1. Введение

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из важных задач. Программные продукты применяются в повседневно окружающих нас вещах: настольных компьютерах, смартфонах, автомобилях и технологиях «интернета вещей». Кроме того программное обеспечение используется на объектах критической инфраструктуры, сбои в работе которых могут привести к серьезным последствиям, а использование ошибок в злонамеренных целях может причинить колоссальный ущерб. В безопасности программных продуктов заинтересованы не только конечные пользователи, но и разработчики. Различные научные институты и крупные корпорации уделяют особое внимание разработке методов и технических средств для поиска и оценки возможности эксплуатации уязвимостей.

Одной из первых работ, в которой описывался процесс эксплуатации уязвимости переполнения буфера на стеке была статья [1], опубликованная в 1996 году. На тот момент промышленно выпускаемые программы не обладали защитными механизмами и использование такого рода уязвимостей проходило не вызывая затруднений. С появлением защитных механизмов уровня операционной системы, таких как рандомизация адресного пространства (ASLR) и защита от исполнения данных (DEP), методы использования уязвимостей, направленные на выполнение произвольного кода, заметно усложнились, а в некоторых случаях стали просто неприменимыми. В современных дистрибутивах операционных систем ASLR и DEP, как правило, включены по умолчанию. Вместе с тем, существуют ситуации, когда использование уязвимостей возможно, несмотря на работу этих защитных механизмов [2,3]. Метод защиты, предполагающий размещение в автоматической памяти служебных переменных, «канареек», был призван защищать программу от ошибок переполнения буфера на стеке, но так же, как и в предыдущем случае, существуют возможности обхода данного защитного механизма [4].

В настоящее время активно развивается технология автоматизированной генерации эксплойтов [6-10]. В основе этой технологии лежит динамическое символьное выполнение. Символьное выполнение представляет собой процесс

выполнения программы, где вместо конкретных значений переменных используются символьные значения. В качестве символьных значений обычно выступают входные данные программы, которые могут быть получены из различных источников (сеть, файлы, аргументы командной строки, переменные окружения). Процесс преобразования входных данных в результате работы программы описывается в виде формул над символьными переменными. Условные ветвления, результат выполнения которых зависит от входных данных, порождают уравнения, отвечающие прохождению по определенной ветке в программе. Совокупность всех таких уравнений называется предикатом пути. Решением системы уравнений является конкретный набор входных данных, обеспечивающий прохождение потока управления по пути, в рамках которого была построена данная система уравнений. Для решения систем уравнений используются SMT-решатели [20].

Под эксплуатацией уязвимости понимается процесс перехода программы в состояние, в котором данная уязвимость проявляется. Это состояние описывается при помощи дополнительных уравнений, которые в совокупности с предикатом пути обеспечивают процесс эксплуатации уязвимости. Решением такой системы уравнений будет являться эксплойт – набор входных данных, активирующий уязвимость. Формальное описание срабатывания уязвимостей является сложной задачей, но для некоторых классов уязвимостей, например, переполнение буфера на стеке эта задача уже решена и описана в работах [6-10]. Представленные в этих работах инструменты позволяют эксплуатировать уязвимости при отключенных современных защитных механизмах, что накладывает ограничения на применимость данных инструментов в настоящее время. Данная статья предлагает подход к преодолению этих ограничений.

Статья организована следующим образом. Во втором разделе приводится краткое описание современных защитных механизмов, а также способов их обхода. В третьем разделе представлены методы генерации эксплойтов с учетом работы современных защитных механизмов. Реализация и практическое применение рассматриваются в четвертом разделе. В заключении, обсуждаются полученные результаты и дальнейшие направления исследований.

2. Защитные механизмы и способы их обхода

Среди современных механизмов защиты от эксплуатации уязвимостей можно выделить две категории, повсеместно применяющиеся и во многом определяющие состояние защищенности настольных компьютеров и серверов: защитные механизмы операционной системы и защитные механизмы, предоставляемые компилятором. К первой категории относятся рандомизация адресного пространства (ASLR) и предотвращение выполнения данных (DEP). Во вторую категорию входят упомянутый выше механизм размещения «канареек» и технология Fortify source, совмещающая легковесные проверки

времени компиляции и автоматическую замену потенциально уязвимых функций на защищенные аналоги. На стыке двух категорий оказываются методы защиты, работающие во время загрузки программы и динамической компоновки.

Рандомизация адресного пространства (ASLR). Рандомизация адресного пространства предполагает, что программа будет загружаться на различные адреса. В Windows рандомизация поддерживается, начиная с Windows Vista. В ОС Linux данный механизм защиты доступен достаточно давно, начиная с ядра версии 2.6.25 (17 апреля 2008 г.), но полноценная реализация требует решения ряда вопросов.

Рандомизировать необходимо размещение стека, кучи, динамически загружаемых библиотек, адреса загрузки секций ELF-файлов (в том числе секции .text). Рандомизация загрузки динамических библиотек, в свою очередь, требует обязательной рандомизации в функциях отображения файлов в память (mmap). Размещение на произвольных адресах секции .text требует от разработчиков компилировать все исполняемые файлы приложений в позиционно-независимый код, для чего необходимы дополнительные усилия. Отказ от компиляции в позиционно-независимом коде приводит к тому, что образ программы остается нерандомизированным.

Следует отметить, что большинство реализаций ASLR вырабатывают случайную карту памяти для программы один раз на все время работы системы до следующей загрузки. Это означает, что перезапускаемые сервисы, падение которых не приводит к падению всей системы, будут достаточно долго работать на одной и той же карте памяти. Утечка информации об адресах работающей программы позволит настроить эксплойт таким образом, что он сможет преодолеть ASLR.

Таким образом, имеются два основных подхода к противодействию ASLR: (1) передача управления на нерандомизированные области памяти; (2) выяснение и последующее использование знаний об адресах переменных, кучи, стека, а также базовых адресов исполняемых модулей. В первом случае используется передача управления на инструкции «трамплины» вида *call/jump $reg*, выполнение которых в свою очередь передает управление на код полезной нагрузки. Стоит отметить, что для использования этой технологии необходимо наличие нерандомизированной и исполняемой области памяти, а в ней существование «трамплина», и кроме того значение регистра должно указывать в область памяти, находящуюся под контролем атакующего. Второй способ, утечка чувствительных данных, реализуется посредством эксплуатации все тех же уязвимостей форматной строки или переполнения буфера.

Предотвращение выполнения данных DEP. Предотвращение выполнения данных – механизм защиты, встроенный в различные операционные системы Windows, Linux и т.д., который запрещает программе выполнять код из области памяти, помеченной как «данные». Таким образом, стек и куча становятся

недоступными для выполнения. Для обхода этой защиты используется атака возврата в библиотеку, в результате которой происходит подмена адреса возврата из функции адресом библиотечной функции, например, функции system из библиотеки libc.

ASLR и DEP. Одновременная работа этих защитных механизмов значительно усложняет процесс эксплуатации уязвимостей. В условиях работающей технологии DEP бессмысленно использовать «трамплины», передающие управление на стек, так как эта часть памяти выполняться не может. Также нельзя использовать атаку возврата в библиотеку, если адрес загрузки библиотеки рандомизируется. Для обхода этих защит можно использовать технологию, которая называется возвратно-ориентированное программирование (ROP) [8]. Код полезной нагрузки формируется из исполняемых участков кода, заканчивающихся инструкцией передачи управления. Такие участки кода называются гаджетами. Гаджеты выстраиваются в цепочки: первый гаджет передает управление второму, второй – третьему и т.д. Поиск гаджетов происходит в нерандомизированных областях памяти. В Linux исполняемые файлы, как правило, не собираются в виде позиционно-независимого кода, поэтому адрес загрузки образа программы не изменяется от запуска к запуску. В таком случае, исполняемый модуль подходит для поиска гаджетов.

«Канарейка». «Канарейка» – это специальное значение, которое размещено на стеке и разделяет собой пространство автоматических локальных переменных и служебные данные – адрес возврата и сохраненные регистры. Размещение «канарейки» выполняется в прологе функции, в эпилоге ее значение сравнивается с эталоном. Их различие интерпретируется как срабатывание ошибки переполнения буфера с угрозой порчи адреса возврата и других служебных данных. Программа в таком случае аварийно завершается, не доходя до более серьезных нарушений безопасности.

Значение «канарейки» либо заранее определено (состоит из терминальных символов), либо генерируется случайным образом. Кроме добавления «канарейки» происходит перестановка локальных переменных. Все указатели располагаются перед массивами, что препятствует их перезаписи. Стоит отметить, что перестановок полей в структурах не происходит.

Применение «канарейки» сразу же показало эффективность данного метода защиты. Схожим образом защищаются буферы динамической памяти, соответствующая технология была реализована в glibc 2.3.4 [13]. Помимо того, защитные механизмы контролируют порчу служебных данных кучи, и производят дополнительные проверки, например, на двойное освобождение. Защита кучи развивалась вплоть до версии glibc 2.10, учитывая более изощренные способы эксплуатации [14].

Для обхода «канарейки» может быть использована уязвимость CWE-123 [21], когда имеется возможность записать одно произвольное значение по

произвольному адресу. Например, адрес указывает на ячейку памяти, содержащую адрес возврата из функции. Записываемым значением выступает адрес, по которому расположен код полезной нагрузки. Уязвимость реализуется путем переполнения буфера в структуре, с последующей перезаписью поля-указателя.

ASLR, DEP и «канарейка». Одновременная работа трех защит делает эксплуатацию уязвимостей еще менее осуществимой, но полностью такую возможность не исключает. При выполнении ряда условий угроза эксплуатации остается актуальной.

- Исполняемый модуль программы не рандомизируется.
- Наличие гаджета-трамплина, который сдвигает указатель стека на определенное значение.
- ELF-файл использует «ленивое» связывание.
- В программе должна присутствовать уязвимость CWE-123. В качестве адреса записи используется ячейка .got секции, в которой располагается адрес вызываемой в дальнейшем библиотечной функции. Записываемое значение – адрес гаджета-трамплина, который передвинет указатель стека на оставшуюся часть ROP-цепочки.
- Наличие данных, зависящих от входных данных на стеке в момент передачи управления на гаджет-трамплин.

Fortify Source. Компилятор gcc, начиная с версии 4.0, поддерживает макрос FORTIFY_SOURCE, активирующий легковесные проверки на переполнение буфера в библиотечных функциях копирования: memcpy, strcpy, sprintf, gets и др. Некоторые проверки осуществляются во время компиляции, выдавая результат в виде предупреждений, остальные проверки происходят во время выполнения и в случае срабатывания аварийно завершают программу. Реализуются проверки времени выполнения путем автоматической замены вызываемых функций на их аналоги, имеющие дополнительный параметр – размер буфера-приемника, в тех случаях, когда этот размер известен.

Помимо того, Fortify Source отслеживает в компилируемом коде наличие проверок возвращаемых значений определенных функций (system, write, open, gets и др.). Функции, работающие с форматной строкой, заменяются аналогами, которые проверяют использование формата %n, отслеживают использование не литеральных форматов и др. Наличие таких проверок нацелено на предотвращение целенаправленной перезаписи адреса возврата, не вызывающей порчи «канарейки».

Компиляторные преобразования для повышения сложности эксплуатации уязвимостей. В программном обеспечении содержатся уязвимости, для каждой найденной уязвимости требуется выпустить обновление безопасности и распространить на все изделия. Если добиться того, чтобы эксплойт разработанный для одного из изделий оказался не работоспособен на других, либо приводил не к исполнению произвольного кода, а к отказу в

обслуживании, то это позволит повысить безопасность. Для достижения этой цели на базе обфусцирующего компилятора, разрабатываемого в ИСП РАН, были реализованы преобразования [15]: перестановки функций, добавления и перемешивания локальных переменных. Совместное применение данных преобразований позволяет создать популяцию диверсифицированных бинарных файлов. В качестве базы для реализации преобразований использовались компиляторы GCC и CLANG/LLVM. С помощью компилятора GCC производилась полная сборка системы CentOS, включая ядро Linux. Влияние на производительность оценивалось на приложении SQLite, пиковое замедление составило 15% при увеличении размера кода на 5%.

Добавление и перемешивание локальных переменных преследовало цель затруднить активацию известных уязвимостей, поскольку изменяется размер и расположение перезаписываемых локальных переменных (буферов). Перестановка функций местами направленно на дополнение системного механизма ASLR и повышение его разрешения до уровня функций в исполняемом модуле для затруднения построения цепочки гаджетов, работающих на других сборках ПО. Для обеспечения максимального уровня защиты рекомендуется использовать предлагаемый подход совместно с ASLR и DEP.

Рандомизация адресного пространства с гранулярностью до функций средствами навесной защиты (технология Selfrando). Дальнейшим развитием идеи рандомизации адресного пространства является уменьшение размера участков памяти, для которых производится рандомизация. Одна из реализаций такого подхода внедрена в Tor браузер с июня 2016 года [16]. Суть идеи состоит в размещении функций в отдельных секциях для получения информации об их границах с использованием специальной опции компилятора. В дальнейшем информация о границах функций, настраиваемых адресах и специальный код начального загрузчика во время компоновки добавляются к образу исполняемого файла, точка входа которого изменяется на адрес начального загрузчика. Далее используется идея, схожая с используемой в упаковщиках исполняемых файлов. После загрузки программы в память система передает управление на код начального загрузчика, который, используя информацию о функциях, производит их размещение в памяти в случайном порядке, настраивает адреса и передает управление на оригинальную точку входа. Для оценки эффективности метода авторы провели эксперимент по оценке энтропии, обеспечиваемой ASLR и их подходом. Для системы Debian 8.4 и компиляторов GCC 6.1.0 и Clang 3.5.0 на 32-х битной версии ОС ASLR обеспечивает изменение 9 бит адреса, на 64-х битной версии – 29 бит. Энтропия, обеспечиваемая Selfrando, зависит от размера модуля программы и количества функций (N): для 10KB кода энтропия составляет 13 x N бит, для 6.6 MB – 22 x N бит, для 92 MB – 26 x N бит. Влияние на производительность оценивалось с помощью SPEC 2006, геометрическое среднее значение замедления по всем тестам составило 0.71% для GCC.

Тестирование производилось в режиме “Identity transformation”, при котором отрабатывает код загрузчика, но все функции остаются в оригинальном порядке.

Защита времени загрузки. Некоторые приемы эксплуатации уязвимостей используют перезапись элементов Global Offset Table (GOT). Для защиты от таких приемов глобальную таблицу смещений размещают в памяти, доступной только на чтение. Чтобы полностью защитить ее от перезаписи, необходимо отказаться от ленивого связывания, использующегося в динамической компоновке. Для этого ELF файл подготавливается, как требующий незамедлительное связывание (BIND_NOW).

2.1 Анализ современных дистрибутивов ОС Linux на предмет защищенности исполняемых файлов

Анализ на предмет защищенности проводился для исполняемых файлов из директории */usr/bin/* в распространенных дистрибутивах ОС Linux. Для анализа использовалась утилита hardening-check из пакета hardening-includes. Данная утилита позволяет проверить, собран ли исполняемый файл в позиционно-независимом коде (ПНК), используются ли безопасные функции Fortify Source и «канарейка», защищена ли секция .got от записи, происходит ли связывание непосредственно в момент загрузки. В табл. 1 приведены результаты анализа некоторых современных дистрибутивов Linux, позволяющие сделать несколько выводов.

Табл. 1. Результаты анализа содержимого /usr/bin некоторых дистрибутивов Linux

Table 1. Analysis results for binaries from /usr/bin for some Linux distributions

Дистрибутив	Дата выпуска	Количество исп. файлов	Без ПНК	Без «канарейки»	Без Fortify Source	Ленивое связывание
Debian 6.0.10 32 разряда	19.07.14	4705	4613 / 98%	4617 / 98%	3899 / 83%	4639 / 99%
Debian 8.3.0 32 разряда	23.01.16	387	311 / 80%	81 / 23%	70 / 20%	311 / 80%
Arch 32 разряда	25.05.16	2846	2671 / 94%	383 / 13%	493 / 17%	2717 / 95%
Ubuntu 14.10 32 разряда	23.10.14	1162	1016 / 87%	242 / 21%	96 / 8%	1036 / 89%
Ubuntu 14.04.1 64 разряда	24.07.14	851	712 / 84%	67 / 8%	48 / 6%	709 / 83%
Ubuntu 16.04.1 64 разряда	21.07.16	1053	891 / 85%	211 / 20%	81 / 8%	881 / 84%

Подавляющее большинство (порядка 85%) исполняемых файлов собирается не в виде позиционно-независимого кода, что позволяет их использовать для поиска гаджетов. При этом следует признать, что не изучалось, какие именно

программы остались позиционно зависимыми и насколько критична их компрометация с точки зрения безопасности всей системы. С другой стороны доля программ с позиционно зависимым кодом не меняется в течение последних двух лет, их одинаково много как в 32-х разрядных, так и в 64-х разрядных системах.

Во многих программах используется ленивое связывание, что оставляет возможность перезаписи .got-слотов с целью передачи управления на код полезной нагрузки.

Анализ дистрибутивов Linux показал, что в большинстве современных дистрибутивах присутствуют такие защитные механизмы, как: ASLR, DEP, «канарейка» и FORTIFY_SOURCE. Защитный компиляторный механизм FORTIFY_SOURCE фактически исправляет некоторые ошибки, совершаемые разработчиком, тем самым устраняя потенциальные возможности эксплуатации уязвимостей, основанных на этих дефектах. Таким образом, предлагаемые методы эксплуатации сосредоточены на попытках преодоления таких защитных механизмов, как: ASLR, DEP и «канарейка».

3. Методы использования уязвимостей

Предложенные методы являются развитием подхода к генерации эксплойтов для уязвимости переполнения буфера на стеке, описанного в предыдущей статье [10]. Методы основываются на анализе трасс выполнения, полученных при помощи полносистемного эмулятора. Процесс генерации эксплойта состоит из двух этапов: построения предиката пути и описания предиката безопасности.

3.1 Построение предиката пути

Предикат пути представляет собой набор символьных уравнений и неравенств, описывающий процесс преобразования входных данных программы с момента получения и до аварийного завершения. Для поиска аварийных завершений используется подход на основе анализа прерываний процессора, более подробно рассмотренный в работе [10]. Указание момента получения входных данных, а именно: шага трассы, на котором буфер с входными данными оказывается заполненным, может задаваться аналитиком или определяться автоматически. Для этого используется анализ функций, которые получают данные из различных источников: сеть, файлы, аргументы командной строки. Как правило, такие функции располагаются в известных библиотеках, например, функция *recv* из библиотеки *libc*. Аналитик заранее может подготовить описания известных ему функций получения входных данных и их параметров в различных библиотеках. Эта информация будет использоваться для автоматического поиска точек получения входных данных.

Построение предиката пути над символьными переменными основывается на отслеживании помеченных данных. Отобранные в результате отслеживания

помеченных данных машинные инструкции переводятся в архитектурно-независимое промежуточное представление [11], уже по которому строятся символьные уравнения и неравенства. Как известно, анализу помеченных данных свойственны такие проблемы, как недостаточная помеченность и избыточная помеченность [12].

Недостаточная помеченность возникает, как правило, из-за того, что во время анализа не учитываются некоторые зависимости. В ходе работы программы символьные данные могут оказаться в адресном коде, определяя значение адреса памяти. Дальнейшая интерпретация кода либо предполагает обращение к любой ячейке адресуемой памяти, либо требует ограничения числа возможных адресов, вплоть до конкретизации значения адреса. Такая проблема известна в публикациях, как проблема «символьных адресов» [6].

В рамках предлагаемого подхода символьные пометки не распространяются через адреса, что может приводить к недостаточной помеченности. В свою очередь, недостаточная помеченность может привести к тому, что набор входных данных, полученный в результате решения предиката пути, не проведет программу по тому же самому пути выполнения.

В некоторых случаях последствий недостаточной помеченности можно избежать, добавляя дополнительные ограничения на символьные переменные. Обладая некоторыми сведениями об устройстве программы, аналитик может добавить ограничения на входные данные, параметры функций, а также на произвольные ячейки памяти и регистры. В качестве примера, можно привести ситуацию, когда входные данные считываются при помощи функции *scanf*. Аналитик может добавить такие ограничения, что входные данные не содержат терминальных символов и пробелов.

Избыточная помеченность может привести к росту количества отобранных инструкций, не все из которых оказывают существенное влияние на ход анализа. В основном, такие инструкции содержатся в коде библиотек. Прекращение отслеживания помеченных данных при выполнении кода библиотек может привести к потере нужных зависимостей, например, если копирование помеченных данных происходит с использованием функций копирования. Поэтому предлагается следующий подход. Аналитик указывает список неинтересных для анализа функций. Помимо этого, для каждой функции указывается набор параметров, с которых необходимо снять пометки. Таким образом, в предикат пути не попадут те инструкции внутри функции, на результат выполнения которых могли повлиять выбранные параметры. Примером такой функции может послужить функция *malloc* из библиотеки *libc.* Использование данного подхода позволяет значительно снизить количество отобранных инструкций.

3.2 Построение предиката безопасности

Набор символьных уравнений и неравенств, который описывает факт проявления уязвимости, называется предикатом безопасности. Как правило, построение предиката безопасности происходит в момент проявления дефекта, т.е. сразу после построения предиката пути. В совокупности с предикатом пути, предикат безопасности составляет полный набор уравнений и неравенств, описывающий условия эксплуатации уязвимости.

Противодействие DEP и ASLR посредством использования ROP. Эксплуатация переполнения буфера на стеке с использованием ROP-цепочек не сильно отличается от классического подхода с передачей управления на шелл-код или трамплин [10]. Для успешной эксплуатации переполнения буфера на стеке необходимо, чтобы адрес возврата из функции указывал на первый гаджет, а остальная часть цепочки располагалась выше по стеку. На рис.1 показано состояние стека в момент эксплуатации переполнения буфера при помощи ROP.

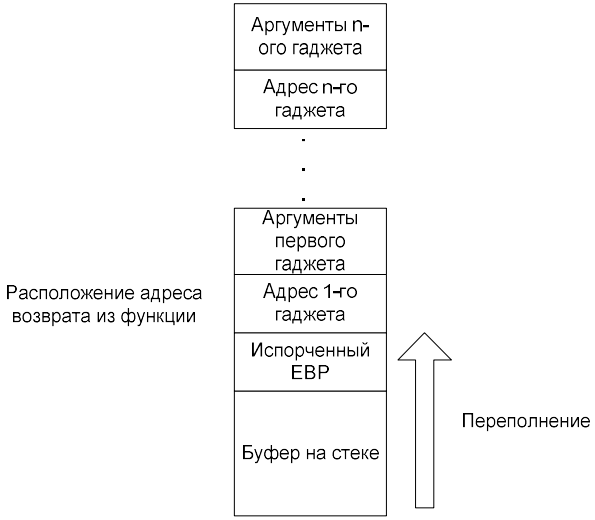

Рис.1. Состояние стека в момент эксплуатации при помощи ROP

Fig. 1. Stack frame in time of explotation using ROP

Таким образом, предикат безопасности будет состоять из набора уравнений, который описывает размещение ROP-цепочки, начиная от ячейки с адресом возврата из функции и далее выше по стеку.

Противодействие «канарейке» посредством эксплуатации уязвимости CWE-123. Как и при классическом переполнении буфера на стеке необходимо добавить набор уравнений, который описывает размещение шелл-кода в выбранной области памяти. Для успешной передачи управления на нужный

адрес необходимо добавить два уравнения. Первое уравнение указывает на то, что адрес записи равен адресу ячейки, в которой хранится адрес возврата из функции. Второе уравнение указывает на то, что записываемое значение равно адресу, по которому располагается шелл-код. Объединение эти уравнения с предикатом пути позволит получить набор уравнений, решением которых является эксплойт, способный работать при наличии «канарейки» на стеке.

Противодействие DEP, ASLR и «канарейке». Для того чтобы успешно осуществить эксплуатацию в рамках одновременной работы трех механизмов защит, необходимо выполнение условий, описанных во втором разделе. На рис. 2 показано состояние стека и GOT в момент перехвата потока управления.

Рис.2. Состояние стека и GOT в момент перехвата потока управления

Fig 2. Stack frame and GOT in time of control flow hijack

Уязвимость CWE-123 используется для передачи управления на гаджет-трамплин, который сдвинет указатель стека на нужное смещение. Для этого требуется добавить два уравнения. Первое уравнение указывает на то, что адрес записи равен ячейки из GOT, в которой хранится адрес вызываемой в дальнейшем библиотечной функции. Второе уравнение указывает на то, что записываемое значение равно адресу гаджета-трамплина. После выполнения гаджета-трамплина указатель стека сместится вверх в область локальных переменных, находящуюся под контролем атакующего. В этой области

необходимо разместить основную часть ROP-цепочки. Для этого следует добавить соответствующие уравнения.

4. Реализация методов и результаты практического применения

Предложенные методы реализованы в виде программного инструмента. Метод использует ранее разработанные и реализованные средства: повышение уровня представления трассы машинных команд, а также модель процессора общего назначения. Для решения системы символьных уравнений и неравенств, полученной в результате построения предиката пути и предиката безопасности, используется SMT-решатель Z3 [17].

Разработанное программное средство было протестировано на нескольких примерах.

Недостаточная помеченность. Механизм добавления дополнительных ограничений на символьные данные применялся для противодействия возникающей недостаточной помеченности при генерации эксплойта для уязвимости переполнения буфера на стеке в программе *faad* из одноименного пакета. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были отключены. Недостаточная помеченность возникала при копировании функцией *vsscanf* буфера с контролируемыми данными на стек. Во время копирования функция *vsscanf* проверяет наличие в буфере-источнике непечатаемых символов (whitespace). При обнаружении таких символов копирование заканчивается. В трассе выполнения такие проверки реализованы с использованием адресных зависимостей. Для того чтобы компенсировать отсутствие таких проверок, аналитик добавил необходимые ограничения (отсутствие пробелов) на параметр буфера-источника функции *vsscanf*. В результате чего был сгенерирован работоспособный эксплойт.

Избыточная помеченность. Механизм прекращения отслеживания параметров функций применялся для противодействия избыточной помеченности во время генерации эксплойта для уязвимости переполнения буфера на стеке в программе *blast2* из одноименного пакета. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 6.0.10. Защитные механизмы, такие как DEP и ASLR, были отключены. Во время выполнения программа многократно вызывает функцию *malloc*, параметр которой зависит от входных данных. Таким образом, в обработку попадают инструкции, относящиеся к работе функции *malloc*, что увеличивает размер полученной системы символьных уравнений. Во избежание такой ситуации аналитик составил описание функции *malloc* с указанием параметра, с которого необходимо снять пометку. Это позволило сократить время генерации эксплойта с 6 часов до 21 минуты.

Противодействие DEP и ASLR с помощью ROP. При генерации эксплойта для уязвимости переполнения буфера на стеке в программе *zsnes* из одноименного пакета использовался метод генерации эксплойта с помощью ROP. Код полезной нагрузки, вызывающий интерпретатор командной оболочки, был составлен в виде ROP-цепочки. Гаджеты были получены из исполняемого файла программы, который не является позиционно-независимым кодом. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были включены. В результате был получен эксплойт, способный функционировать в условиях работы DEP и ASLR.

Эксплуатация уязвимости переполнения буфера на стеке в программе уровня ядра ОС. Для проведения эксперимента по эксплуатации уязвимости в программе, которая работает на уровне ядра операционной системы, был подготовлен модельный пример модуля ядра операционной системы Linux. В этом примере присутствовала уязвимость переполнения буфера на стеке. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0, в котором работает ASLR. Для эксплуатации использовался метод генерации эксплойта для переполнения буфера на стеке с использованием трамплинов. Поиск трамплина происходил в образе ядра операционной системы. Таким образом, был сгенерирован работоспособный эксплойт.

Эксплуатация ошибки типа CWE-123. Эксплуатация ошибки типа CWE-123 проводилась на примере из работы [18]. В примере происходит переполнение буфера на стеке с последующей перезаписью вышележащего по стеку указателя, по которому в дальнейшем происходит запись контролируемых значений. Данный пример был модифицирован следующим образом: переполняемый буфер и указатель были объединены в структуру, которую расположили на стеке, и программа была скомпилирована компилятором *gcc* с добавлением опции *-fstack-protector-all*, что привело к добавлению канарейки в уязвимую функцию. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были отключены. Применение метода генерации эксплойта для ошибки типа CWE-123 с передачей управления на код полезной нагрузки позволило получить рабочий эксплойт.

Эксплуатация ошибки типа CWE-123 при наличии DEP, ASLR и «канарейки». Для проведения экспериментов по эксплуатации ошибки типа CWE-123 при наличии DEP, ASLR и «канарейки» был разработан модельный пример программы, фрагмент листинга которой приведен ниже.

```
1  typedef struct
2  {
3   char name[64];
4   char *bio;
5  } Person;
6
7  int
8  main(int argc, char *argv[])
9  {
10  int cnt = argc / 2;
11  Person persons[cnt];
12
13  for (int i = 0; i != cnt; ++i)
14  {
15          persons[i].bio = malloc(strlen(argv[2 * i + 2]) + 1);
16          strcpy(persons[i].name, argv[2 * i + 1]);
17          strcpy(persons[i].bio, argv[2 * i + 2]);
18  }
```

Листинг 1. Пример программы с ошибкой CWE-123

Listing 1. Program example with CWE-123 defect

Программа была скомпилирована компилятором *gcc* с опцией *-fstack-protector-all.* Из-за того, что буфер и указатель являются полями структуры, их переразмещение компилятором не происходит, и указатель располагается выше на стеке. Благодаря отсутствию проверки на размер при копировании в строке 16 происходит переполнение буфера, которое приводит к перезаписи указателя. Затем в строке 17 уже по перезаписанному указателю выполняется копирование контролируемых данных, что порождает возникновение ошибки типа CWE-123. Для обхода защит DEP и ASLR стоит воспользоваться технологией ROP, но при наличии «канарейки» нельзя перезаписать адрес возврата из функции *main* путем непосредственного переполнения буфера. Для этого стоит воспользоваться эксплуатацией ошибки типа CWE-123, но из-за работы ASLR неизвестен адрес ячейки, в которой храниться адрес возврата из функции *main*. Запись выполнится в .got-слот функции *strlen*, поскольку размещение GOT не рандомизировано, а сама функция *strlen* будет вызвана на очередной итерации цикла. Записываемым значением служит адрес специального гаджета-трамплина, который сдвигает указатель стека вверх на фиксированное значение. После вызова функции *strlen* управление будет передано на гаджет-трамплин. Гаджет-трамплин подбирается таким образом, что в результате его выполнения указатель стека будет ссылаться на контролируемую область памяти. В этой области размещается основная часть

ROP-цепочки, выполняющая код полезной нагрузки. Используя описанный выше метод, удалось автоматически сгенерировать работоспособный эксплойт.

5. Заключение

В настоящее время в мире активно исследуются возможности автоматической генерации эксплойтов. Инструмент AEG [5] производит поиск уязвимости переполнения буфера и уязвимости форматной строки, используя исходные тексты программы. Эксплуатация найденных уязвимостей происходит уже посредством анализа бинарного кода. Инструмент MAYHEM [6] является развитием AEG, он осуществляет поиск и эксплуатацию, анализируя только бинарный файл программы. Эти инструменты находятся в закрытом доступе, а в публикациях нет информации о том, как происходит противодействие современным защитным механизмам. Система CRAX [7], основанная на полносистемном символьном выполнении, представленном в системе S2E [19], позволяет генерировать эксплойты при наличии входных данных, приводящих к аварийному завершению программы. В публикации по системе CRAX предлагаются подходы к генерации эксплойтов для уязвимостей переполнения буфера на стеке и на куче, а также уязвимости форматной строки. Описываются способы обхода защитных механизмов DEP и ASLR. Реализация системы, представленная в открытых источниках, позволяет генерировать эксплойты только для переполнения буфера на стеке без учета работы защитных механизмов. Система REX [9] дает возможность генерировать эксплойты для уязвимостей переполнения буфера на стеке в программах для операционной системы Linux при работе DEP и ASLR.

В статье представлен метод генерации входных данных, подтверждающих наличие в программе критических ошибок переполнения буфера на стеке и ошибки типа CWE-123. Данный метод основан на символьном выполнении бинарного кода и учитывает работу современных защитных механизмов, таких как DEP, ASLR и «канарейка». Применение предложенного метода позволит выявлять среди зафиксированных аварийных завершений крайне важные ситуации, когда использование дефекта приводит к выполнению произвольного кода, несмотря на работу современных защитных механизмов.

Для рассмотренных в данной статье примеров составлялись ROP-цепочки, что потребовало минимальной автоматизации, а именно классификации найденных в программе ROP-гаджетов. Предложенный метод реализован в виде программного инструмента.

Перспективными направлениями дальнейших исследований являются исследование вопросов, связанных с уязвимостями переполнения буфера на куче, а также исследование возможности перехвата потока данных.

Список литературы

[1]. One A. Smashing the stack for fun and profit. Phrack magazine, v. 7, №. 49, 1996, pp. 14-16.

[2]. Durden T. Bypassing pax aslr protection. Phrack Magazine, v. 59, №. 9, 2002, pp. 9.

[3]. Nergal. The advanced return-into-lib (c) exploits: PaX case study. Phrack Magazine, Volume 58, Issue 4, 2001.

[4]. Bulba K. Bypassing stackguard and stackshield. 2000.

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

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

[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on., IEEE, 2012, pp. 78-87.

[8]. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007 ACM Conference on Computer and Communications Security (CCS), Proceedings of CCS 2007, pp. 552-561.

[9]. Shoshitaishvili Y. et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138-157.

[10]. Padaryan V. A., Kaushan V. V., Fedotov A. N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, v. 41, №. 6, 2015, pp. 373-380. DOI: 10.1134/S0361768815060055.

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

[12]. J. Kim, T. Kim and E. G. Im. Survey of dynamic taint analysis. 2014 4th IEEE International Conference on Network Infrastructure and Digital Content, Beijing, 2014, pp. 269-272.

[13]. Ubuntu security features. https://wiki.ubuntu.com/Security/Features

[14]. Malloc Des-Maleficarum. Phrack magazine, v. 13, issue 66, 2009.

[15]. Nurmukhametov, AR; Kurmangaleev, Sh F; Kaushan, VV; Gaissaryan, SS. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, v. 41, № 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052.

[16]. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In Proceedings of the 16th Privacy Enhancing Technologies Symposium (PETS 2016), in press, Darmstadt, Germany, July 19-22, 2016.

[17]. Nikolaj Bjorner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions. Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[18]. Heelan, S. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. M.Sc. thesis. University of Oxford, Oxford, U.K., Sept. 3, 2009.

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

[20]. Vanegue J., Heelan S., Rolles R. SMT Solvers in Software Security. WOOT, 2012, pp. 85-96.

[21]. CWE-123, https://cwe.mitre.org/data/definitions/123.html

Software defect severity estimation in presence of modern defense mechanisms★

[1]A.N. Fedotov <fedotoff@ispras.ru>
[1,2]V.A. Padaryan <vartan@ispras.ru>
[1]V.V. Kaushan <korpse@ispras.ru>
[1]Sh.F. Kurmangaleev <kursh@ispras.ru>
[1]A.V. Vishnyakov <vishnya@ispras.ru>
[1]A.R. Nurmukhametov <oleshka@ispras.ru>
[1]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

Abstract. This paper introduces a refined method for automated exploitability evaluation of found program bugs. During security development lifecycle a significant number of crashes is detected in programs. Because of limited resources, bug fixing is time consuming and needs prioritization. It should be the matter of highest priority to fix exploitable bugs. Automated exploit generation technique is used to solve this problem in practice. Generated exploit confirms the presence of a critical vulnerability. However, state-of-the-art publications omit modern defense mechanisms preventing exploitation. It results in lowering of an evaluation quality. This paper considers modern vulnerability exploitation prevention mechanisms. An evaluation of their prevalence and efficiency is also presented. The method can be applied to program binaries and doesn't require any debug information. Proposed method is based on symbolic interpretation of traces obtained by a full-system emulator. Our method can demonstrate a real exploitability for stack buffer overflow vulnerability with write-what-where condition even when DEP, ASLR, and "canary" operate together. The implemented method capabilities were shown on model examples and real programs.

Keywords: critical vulnerability; binary code; symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(5)-4

For citation: Fedotov A.N., Padaryan V.A., Kaushan V.V., Kurmangaleev Sh.F., Vishnyakov A.V., Nurmukhametov A.R. Software defect severity estimation in presence 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

★ The paper was supported by RFBR grant # 16-29-09632

References

[1]. One A. Smashing the stack for fun and profit. Phrack magazine, v. 7, №. 49, 1996, pp. 14-16.
[2]. Durden T. Bypassing pax aslr protection. Phrack Magazine, v. 59, №. 9, 2002, pp. 9.
[3]. Nergal. The advanced return-into-lib (c) exploits: PaX case study. Phrack Magazine, Volume 58, Issue 4, 2001.
[4]. Bulba K. Bypassing stackguard and stackshield. 2000.
[5]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D.Brumley. AEG: Automatic exploit generation. Commun. ACM, №2, 2014.
[6]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. IEEE Symposium on Security and Privacy, 2012.
[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on., IEEE, 2012, pp. 78-87.
[8]. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007 ACM Conference on Computer and Communications Security (CCS), Proceedings of CCS 2007, pp. 552-561.
[9]. Shoshitaishvili Y. et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138-157.
[10]. Padaryan V. A., Kaushan V. V., Fedotov A. N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, v. 41, №. 6, 2015, pp. 373-380. DOI: 10.1134/S0361768815060055.
[11]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Simulation of operational semantics of machine instructions. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161-170, DOI 10.1134/S0361768811030030.
[12]. J. Kim, T. Kim and E. G. Im. Survey of dynamic taint analysis. 2014 4th IEEE International Conference on Network Infrastructure and Digital Content, Beijing, 2014, pp. 269-272.
[13]. Ubuntu security features. https://wiki.ubuntu.com/Security/Features
[14]. Malloc Des-Maleficarum. Phrack magazine, v. 13, issue 66, 2009.
[15]. Nurmukhametov A.R.; Kurmangaleev Sh.F.; Kaushan V.V.; Gaissaryan S.S. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, v. 41, № 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052
[16]. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In Proceedings of the 16th Privacy Enhancing Technologies Symposium (PETS 2016), in press, Darmstadt, Germany, July 19-22, 2016.
[17]. Nikolaj Bjorner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions. Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.
[18]. Heelan, S. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. M.Sc. thesis. University of Oxford, Oxford, U.K., Sept. 3, 2009.
[19]. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, v. 46, №. 3, 2011, pp. 265-278.

[20]. Vanegue J., Heelan S., Rolles R. SMT Solvers in Software Security. WOOT, 2012, pp. 85-96.
[21]. CWE-123, https://cwe.mitre.org/data/definitions/123.html