Моделирование и анализ информационных систем. Т. 26, № 2 (2019), с. 213–228
Modeling and Analysis of Information Systems. Vol. 26, No 2 (2019), pp. 213–228

Theory of data

DOI: 10.18255/1818-1015-2019-2-213-228

УДК 517.9

Об обнаружении атак типа повторного использования исполнимого кода

Косолапов Ю. В.

Поступила в редакцию 17 декабря 2018
После доработки 13 мая 2019
Принята к публикации 15 мая 2019

Аннотация. При эксплуатации уязвимостей программного обеспечения типа переполнения буфера в настоящее время часто используется техника повторного использования кода. Такие атаки позволяют обходить защиту от исполнения кода в стеке, реализуемую на программно-аппаратном уровне в современных информационных системах. В основе атак лежит нахождение в уязвимой программе подходящих участков исполнимого кода – гаджетов – и сцепление этих гаджетов в цепочки. В статье предлагается способ защиты приложений от атак, использующих повторное использование кода. Способ основан на выделении свойств, которые позволяют отличить цепочки гаджетов от типичных цепочек легальных базовых блоков программы. Появление во время выполнения программы нетипичной цепочки базовых блоков может свидетельствовать о выполнении вредоносного кода. Одним из свойств цепочки гаджетов является исполнение в конце цепочки специальной инструкции процессора, используемой для вызова функции операционной системы. Для операционной системы Linux на базе архитектуры x86/64 проведены эксперименты, показывающие важность этого свойства при выявлении исполнения вредоносного кода. Разработан алгоритм выявления нетипичных цепочек, который позволяет выявлять все известные на настоящий момент техники повторного использования кода.

Ключевые слова: повторное использование исполнимого кода, уязвимости программного обеспечения

Для цитирования: Косолапов Ю. В., "Об обнаружении атак атак типа повторного использования исполнимого кода", *Моделирование и анализ информационных систем*, **26**:2 (2019), 213–228.

Об авторах:
Косолапов Юрий Владимирович, orcid.org/0000-0002-1491-524X, канд. техн. наук
Южный Федеральный Университет,
ул. Мильчакова, 8а, г. Ростов-на-Дону, 344090 Россия, e-mail: itaim@mail.ru

Введение

В информационных системах, построенных на базе архитектуры x86/64, с целью защиты от вредоносного кода, эксплуатирующего уязвимости типа переполнения буфера (для защиты от эксплоитов), используется техника защиты от выполнения программного кода в стеке и защита от записи в страницы с кодом. Эта техника получила название защита типа «запись, либо исполнение» (write xor execute

$W \oplus X$). Однако этот защитный механизм может быть обойден при использовании техники возвратно ориентированного программирования (ROP – return oriented programming) [1,2]. В основе техники ROP лежит логика инструкции `retq`, которой обычно завершаются подпрограммы (функции) машинного кода для архитектуры x86/64: из ячейки памяти, на которую указывает регистр `rsp` (указатель стека), в регистр-указатель инструкций `rip` загружается адрес следующей выполняемой инструкции, при этом указатель `rsp` увеличивается на 8 байтов (для 64-разрядного процессора). При использовании техники ROP в стек во время эксплуатации уязвимости типа переполнения буфера помещается не сам исполнимый код эксплоита, а адреса и аргументы для *гаджетов* — небольших участков кода, уже находящихся в области памяти уязвимой программы с разрешенным исполнением программного кода и заканчивающихся инструкцией `retq`. В этом случае эксплоит — это последовательность гаджетов (ROP-гаджетов), также называемая ROP-цепочкой. На рис. 1. приведен пример использования возвратно ориентированного программирования. В примере предполагается, что в ходе переполнения буфера удается перезаписать значения стека таким образом, что перед возвратом из уязвимой функции (перед выполнением инструкции `retq` этой функции) стек имеет вид, показанный на рис. 1 слева. Тогда при выполнении инструкции `retq` в регистр `rip` загрузится значение ячейки, на которую указывает регистр `rsp`. Таким образом передается управление по адресу `address1`. В результате выполнится код первого гаджета: в регистр `rbp` загрузится значение `val` из стека (инструкция `popq %rbx`) и выполнится инструкция `retq` этого гаджета. При выполнении этой инструкции регистр `rsp` указывает на ячейку со значением `address2`, поэтому, следуя логике исполнения инструкии `retq`, передается управление на второй гаджет по адресу `address2`. Таким образом, указанное в левой части рисунка содержимое стека приводит к тому, что к текущему содержимому регистра `rax` прибавляется значение `val` и результат помещается в регистр `rcx`.

	Содержимое стека
	...
rsp⇒	address1
	val
	address2
	...

Адрес гаджета	**Код гаджета**
address1	popq %rbx retq

address2	addq %rax,%rbx
	movq %rbx,%rcx
	retq

Рис. 1: Пример содержимого стека (слева) и области исполнимой памяти (справа) при использовании возвратно ориентированного программирования
Fig. 1: An example of stack contents (at the left) and executable memory area (at the right) when using return oriented programming

Задача разработчика эксплоита с использованием техники ROP состоит в нахождении необходимых гаджетов в адресном пространстве программы, содержащей уязвимость. Так как большинство программ используют функции стандартных библиотек, загружаемых в область памяти создаваемых процессов, то часто поиск ROP-гаджетов осуществляется в машинном коде этих библиотек [1], [2]. Например, в стандартной библиотеке GNU libc.so.6 для архитектуры x86/64 насчитывается до

17 550 гаджетов [3], с помощью которых может быть реализован любой алгоритм (в этом случае говорят, что набор гаджетов обладает полнотой по Тьюрингу). Так как до появления технологии ASLR (address space layout randomization) стандартные библиотеки загружались в память процесса по фиксированному адресу, то использование возвратно ориентированного программирования позволяло достаточно просто обходить защиту типа $W \oplus X$. Технология ASLR препятствует включению в эксплоит конкретных значений адресов гаджетов. Однако уязвимости, приводящие к утечке информации об адресах, в совокупности с наличием уязвимостей типа переполнения буфера позволяют в ряде случаев обходить защиту ASLR. Также для обхода такой защиты используется техника распыления кучи (heap spray) [4].

Для защиты от ROP-цепочек предложены различные механизмы: теневой стек [5] (реализуется путем внесения дополнительных инструкций процессора), контроль графа потока управления [6], [7] (реализуется на аппаратном уровне, либо на уровне компиляции программ), марковские цепи [8] (выявление необфусцированных ROP-цепочек в документах), шифрование адреса возврата перед помещением его в стек [9] (однако не все компиляторы следуют правилу, согласно которому инструкции `callq` и `retq` должны находиться в строгом соответствии), подсчет числа неверно угаданных предсказаний [10] (реализуется путем внесения дополнительных инструкций процессора), подсчет числа подряд идущих небольших наборов команд, заканчивающихся инструкцией `retq` [11] (реализуется путем внесения изменений в функциональность процессора). Отметим, что компанией Intel разработана технология CET (Control-flow Enforcement Technology) [12] для защиты от атак, изменяющих поток управления инструкций. В частности, в CET реализован теневой стек и добавлены инструкции для работы с ним. Заметим, что перечисленные способы защиты не используют сигнатуры атак, поэтому позволяют обнаруживать неизвестные атаки.

Разновидностями техники ROP являются такие техники, как JOP (jump oriented programming) [13] и COP/PCOP (call oriented programming/ pure call oriented programming) [14]. Одним из отличий этих техник от ROP является то, что гаджеты заканчиваются не инструкций `retq`, а, соответственно, инструкциями косвенного перехода `jmpq` и `callq`, при этом в JOP и COP используются как гаджеты, заканчивающиеся инструкцией `retq`, так и гаджеты с завершающими инструкциями `jmpq` и `callq`. В PCOP используются только гаджеты, заканчивающиеся инструкцией `callq`. В техниках JOP/COP/PCOP необходимо наличие управляющего гаджета (dispatcher), который загружает адреса выполняемых гаджетов и на который передается управление после выполнения очередного гаджета, либо необходимо наличие наборов связующих инструкций (например, в JOP таким набором может быть последовательность инструкций `popq reg; jmpq reg`, которая реализует логику инструкции `retq`). Новые техники позволяют обходить методы защиты, разработанные для техники ROP. Например, защита на основе теневого стека или на основе подсчета числа подряд идущих инструкций `retq` не позволяет выявлять эксплоиты, построенные на основе JOP/COP/PCOP. В [15] для защиты от техники JOP используется подсчет расстояния (количество инструкций) между адресом инструкции `jmpq/callq` и адресом целевого перехода. Однако этот способ защиты не выявляет короткие цепочки гаджетов. Обобщенный метод защиты из [11], основанный на подсчете числа подряд идущих небольших наборов команд, заканчивающихся инструкцией косвенного перехода, также может быть нивелирован, если в цепочку

гаджетов вставлять специальные гаджеты, которые расцениваются системой защиты как нормальные гаджеты [16], [17] (например, гаджеты большого размера).

В настоящей работе ставится задача на основе подсчета числа подряд встречающихся инструкций косвенного перехода среди инструкций передачи управления разработать способ выявления эксплоитов, основанных на использовании техник ROP/JOP/COP/PCOP, устойчивый к включению маскирующих гаджетов. Задача решается на примере операционной системы семейства Linux для процессора с архитектурой x86/64, однако он может быть применим и для других операционных систем для архитектур x86/32 и x86/64.

Статья состоит из введения, двух разделов и заключения. В первом разделе выделяются отличительные свойства цепочек гаджетов от цепочек базовых блоков программы и строится алгоритм выявления цепочек гаджетов. Во втором разделе приводятся результаты эксперимента по оценке порогового значения одного параметра разработанного алгоритма. В заключении рассматриваются ограничения разработанного способа и некоторые возможные варианты его дальнейшего усовершенствования.

1. Алгоритм выявления атак повторного использования кода

Рассмотрим информационно-аналитическую модель $\mathcal{M} = (S, O, P)$, состоящую из программы S, входными данными которой является документ O. Программа P является системой защиты, целью которой служит блокирование выполнения программы S, когда на вход поступает документ O, содержащий вредоносный код, эксплуатирующий уязвимость в программе S. Такой вредоносный код далее будем называть эксплоитом.

1.1. Цепочки базовых блоков

Обозначим символом $\mathcal{J}$ множество возможных инструкций передачи управления. К таким инструкциям относятся инструкции условного перехода (с префиксом `jaq`, `jeq` и т.п.), безусловного перехода (с префиксом `jmpq`), инструкции вызова подпрограмм (с префиксом `callq`) и инструкции возврата из подпрограмм (`retq` и т.п.). В частности, для архитектуры x86/64 в J входит инструкция вызова функции ядра операционной системы. Эту инструкцию обозначим `syscall`.

Последовательность машинных инструкций, выполняемых процессором при работе программы S с входными данными O обозначим $\texttt{Path}(S, O)$. Для $O \neq O'$ в общем случае $\texttt{Path}(S, O) \neq \texttt{Path}(S, O')$. Последовательность $\texttt{Path}(S, O)$ можно представить в виде связанной цепи *базовых блоков*:

$$B_1, B_2, ..., B_n, \tag{1}$$

где базовый блок B_i представляет собой последовательность машинных инструкций:

$$B_i = (\texttt{op}_{i,1}, ..., \texttt{op}_{i,k_i}), \tag{2}$$

в которых среди первых $k_i - 1$ нет инструкций из $\mathcal{J}$, а последняя инструкция принадлежит $\mathcal{J}$. Отметим, что группировка инструкций в базовые блоки вида (2) используется, например, при разработке компиляторов [18]. Далее, если инструкция `op` содержится среди инструкций базового блока B, будем писать $\texttt{op} \in B$. Под объединением блоков B_i и B_j будем понимать блок $B_i \parallel B_j$ вида:

$$B_i \parallel B_j = (\texttt{op}_{i,1}, ..., \texttt{op}_{i,k_i}, \texttt{op}_{j,1}, ..., \texttt{op}_{j,k_j}). \tag{3}$$

Цепь (1) преобразуем в соответствии со следующим правилом: блок, заканчивающийся инструкцией `syscall`, объединяется в соответствии с (3) со следующим блоком в цепи (если таковой имеется). Процедура преобразования выполняется, пока в цепи имеется хотя бы один, не являющийся последним, блок, который завершается инструкцией `syscall`. В результате получим цепь вида:

$$B'_1, B'_2, ..., B'_{n'}, \tag{4}$$

где $n' \leq n$. Множество всех возможных базовых блоков, из которых состоят цепочки вида (4), обозначим $\mathcal{B}$.

1.2. Свойства цепочек гаджетов

Выделим свойства цепочек гаджетов эксплоитов, которые позволят отличить их от цепочек базовых блоков (4). С одной стороны, совокупность этих свойств должна позволять снизить вероятность распознавания легальной цепочки как цепочки гаджетов, а с другой стороны, вероятность написания эксплоита, не характеризующегося этими свойствами, должна быть минимизирована. Выделяются следующие свойства:

G1) каждый гаджет завершается инструкцией с префиксом из множества

$$\mathcal{E} = \{\texttt{retq}, \texttt{jmpq}, \texttt{callq}, \texttt{syscall}\}; \tag{5}$$

G2) если гаджет в цепочке завершается инструкцией с префиксом `jmpq` или `callq`, то эти инструкции для архитектуры x86/64 в качестве операнда содержат регистр из множества

$$\mathcal{R} = \{\texttt{rax}, \texttt{rbx}, \texttt{rcx}, \texttt{rdx}, \texttt{rbp}, \texttt{rsi}, \texttt{rdi}, \texttt{r8} - \texttt{r15}\};$$

G3) инструкцией `syscall` может завершаться только последний гаджет в цепочке гаджетов;

G4) хотя бы один из гаджетов цепочки содержит инструкцию `syscall`;

G5) гаджет может содержать произвольное количество инструкций.

Поясним эти свойства. Первое свойство G1 выделено в связи с тем, что в настоящее время известны способы составления ROP-, JOP-, COP- и PCOP-цепочек, т.е. цепочек гаджетов, в которых в качестве завершающих используются инструкции из $\mathcal{E}$. Свойство G2 также характерно для гаджетов, так как в этом случае имеется возможность контролировать адрес перехода. Такие инструкции называются инструкциями косвенного перехода. Отметим, что несложно отличить инструкции косвенного перехода от инструкций прямого перехода, так как в последнем случае операндом является адрес, а в первом случае операндом является регистр из множества

$\mathcal{R}$ или ячейка памяти, адресуемая таким регистром. Свойство G3 выделено в связи с тем, что инструкция `syscall` в настоящее время не используется как инструкция для связи гаджетов. Целью эксплоита обычно является запуск программы или запись/чтение файла. Эти операции связаны с обращением к системным функциям, поэтому выделяется свойство G4. Например, по данным сайта `shell-storm.org`, все представленные на сайте эксплоиты для архитектуры x86/64 содержат инструкцию `syscall`, при этом 35 из 37 эксплоитов в качестве последней инструкции содержат инструкцию `syscall` (в основном, это системный вызов `execve`). Согласно [16] и [17], защита, основанная на подсчете длины цепочки гаджетов с ограничением на максимальный размер гаджета и минимальную длину цепочки, может быть нивелирована вставкой маскирующих гаджетов (путем включения гаджетов размера, превышающего заданный в системе защиты порог), поэтому введено свойство G5.

Отметим, что при необходимости ограничение G5 можно ослабить, потребовав, чтобы гаджет содержал ограниченное количество инструкций, так как чем больше инструкций в гаджете, тем вероятнее возникновение побочного эффекта, нежелательного для эксплоита, например, перезапись содержимого используемого регистра. Например, в способе защиты из [10] сигнал о выполнении цепочки гаджетов генерируется в случае нахождения цепочки из 10-ти и более гаджетов, каждый из которых содержит не более шести инструкций. В [19] признаком выполнения цепочки гаджетов является цепочка из не менее чем 4-х гаджетов, каждый из которых имеет длину не более семи. Символами C_l и G_l обозначим соответственно пороговое значение на длину цепочки гаджетов (минимальное значение) и пороговое значение на число инструкций в гаджете (максимальное значение).

Пусть $SysC_l(E)$ — номер последнего гаджета в эксплоите E, содержащий инструкцию `syscall`, $SysC_l = \min_E\{SysC_l(E)\}$. Порядковый номер гаджета, в котором встречается инструкция `syscall`, также будем называть расстоянием от начала цепочки до *этой* инструкции `syscall`. В [14] удалось написать эксплоит, состоящий из двух гаджетов, однако пример приведен для случая, когда не используется технология ASLR. Заметим, что в [20] для обхода ASLR потребовалось написать эксплоит, состоящий из 6 гаджетов, а в [21] эксплоит содержит не менее 4-х гаджетов. Отсюда можно предположить, что значение $SysC_l$ может быть не менее четырех. Во втором разделе это предположение подтверждается экспериментально.

Таким образом, отличительными признаками цепочки гаджетов, удовлетворяющей свойствам G1–G5, от цепочки базовых блоков, также удовлетворяющей этим же свойствам, могут быть максимальная длина цепочки и порядковый номер последнего блока (гаджета) в цепочке, содержащего инструкцию `syscall`.

1.3. Модель защиты

Заметим, что все гаджеты, используемые в техниках ROP/JOP/COP/PCOP могут рассматриваться как базовые блоки, в то же время не каждый базовый блок является гаджетом: базовый блок может заканчиваться инструкцией прямого перехода типа `callq address` или инструкцией условного перехода вида `jaq address`, тогда как гаджеты с такими инструкциями в настоящее время не используются. Таким образом, если $\mathcal{G}$ — множество гаджетов, то $\mathcal{G} \subset \mathcal{B}$. Рассмотрим множество $\mathcal{B}_0 \subset \mathcal{B}$, состоящее из базовых блоков, удовлетворяющих свойствам G1,G2,G5. Оче-

видно, что $\mathcal{G} \subset \mathcal{B}_0$. Также рассмотрим множество $\mathcal{B}_0^V (\subseteq \mathcal{B}_0)$, состоящее из базовых блоков, в каждом из которых не более V инструкций. Следующее соотношение очевидно: ... $\subseteq \mathcal{B}_0^{V-1} \subseteq \mathcal{B}_0^V \subseteq \mathcal{B}_0^{V+1} \subseteq$ .... Будем полагать, что $\mathcal{B}_0^\infty = \mathcal{B}_0$. Соотношение введенных множеств изображено на рис. 2. Если вместо множества $\mathcal{B}_0$ рассматривать его подмножество $\mathcal{B}_0^V$ для некоторого $V \in \mathbb{N}$, то возможна ситуация, когда $\mathcal{G} \not\subset \mathcal{B}_0^V$. В этом случае возможна ситуация пропуска системой защиты P эксплоита при рассматриваемом методе обнаружения: в цепочку гаджетов произвольной длины достаточно вставлять маскирующие гаджеты из более чем V инструкций.

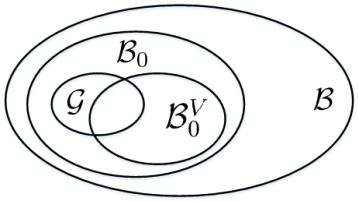

Рис. 2: Соотношение множества гаджетов и множеств базовых блоков
Fig. 2: The correspondence of the set of gadgets and sets of basic blocks

В основе предлагаемого способа защиты лежит предположение о том, что при нормальном выполнении программы S на входных данных O в последовательности (4) не появляются длинные цепочки, состоящие из базовых блоков, принадлежащих исключительно $\mathcal{B}_0$, и одновременно удовлетворяющие свойствам G3 и G4. Поэтому при справедливости этого предположения для выявления факта выполнения эксплоита предлагается в последовательности (4) выявлять подпоследовательности подряд идущих базовых блоков, каждый из которых принадлежит $\mathcal{B}_0$, и одновременно удовлетворяющие свойствам G3 и G4 с длиной более чем C_l и/или для которых последний выявленный базовый блок, содержащий инструкцию `syscall`, находится на расстоянии более чем $SysC_l$ от начала цепочки.

Для выявления цепочек гаджетов вместо цепи (4) рассмотрим последовательность, состоящую из завершающих инструкций:

$$\mathtt{op}_{1,k'_1}, ..., \mathtt{op}_{n',k'_{n'}}, ..., . \tag{6}$$

Тогда для выявления цепочек достаточно искать последовательности подряд идущих инструкций косвенного перехода (ПИИКП) в последовательности (6); префикс которых принадлежит $\mathcal{E}$, а операнд, если есть, содержит регистр из $\mathcal{R}$, причем хотя бы одна инструкция завершает базовый блок, содержащий инструкцию `syscall`. Заметим, что вычисление количества ПИИКП необходимо выполнять на основе $\mathrm{Path}(S, O)$, а не путем простого дизассемблирования соответствующего исполнимого файла программы S. Это связано с тем, что эксплоит может активизироваться только во время исполнения программы S и часто не может быть выявлен в статичном состоянии в документе O. К тому же дизассемблирование исполнимого файла может быть затруднено, если уязвимая программа защищена, например, с помощью методов обфускации.

Пусть $MChain(S, O)$ — максимальное число подряд идущих легальных базовых блоков в цепи (4), удовлетворяющих свойствам G1–G5, а C_l — как отмечалось ранее

— минимальная длина цепочки гаджетов. Если $MChain(S,O) < C_l$, то выявление исполнения цепочки гаджетов может осуществляться по правилу

$$result = \begin{cases} \texttt{exploit}, & counter > C_l \\ \texttt{typical}, & counter <= C_l \end{cases}, \tag{7}$$

где $counter$ — текущая длина цепочки базовых блоков, удовлетворяющих свойствам G1–G5. Если $MChain(S,O) > C_l$, то при использовании правила (7) могут возникать ложные обнаружения цепочек гаджетов: длинная цепочка базовых блоков будет восприниматься как цепочка гаджетов. Для снижения вероятности таких событий предлагается формировать профиль программы. Под профилем программы будем понимать набор хэш-значений, полученных вычислением некоторой хэш-функции $h : \{0,1\}^* \to \{0,1\}^m (m \in \mathbb{N})$ от цепочки базовых блоков, которая встречается в Path(S, O) при выполнении программы S на легальных входных данных O, но длина которой превышает C_l. Здесь предполагается, что цепочка базовых блоков может быть записана последовательностью нулей и единиц. Если $B'_i, ..., B'_{i+r}$ — нетипичная цепочка базовых блоков длины $r + 1 (> C_l)$, в которой встречается инструкция `syscall`, то в профиль должны добавляться следующие хэш-значения $h(B'_i \parallel ... \parallel B'_{i+r-j})$, $j = 0, ..., r - C_l$. Такой профиль программы S обозначим Π_S.

Свойство G4 позволяет существенно снизить вероятность ложного срабатывания. Так, например, эксперименты, проведенные для программ FoxitReader и LibreOffice, показывают на сколько сокращается количество цепочек базовых блоков, похожих на цепочки гаджетов, а также на сколько сокращается длина цепочек. Результаты приведены в таблицах 1 и 2, где L — длины цепочки базовых блоков.

Пусть $Sys(S,O)$ — максимальное расстояние от начала цепочки базовых блоков до последнего базового блока этой цепочки, содержащего инструкцию `syscall`, при выполнении программы S на входных данных O. Если $Sys(S,O) < SysC_l$, то выявление исполнения цепочки гаджетов может осуществляться по правилу

$$result = \begin{cases} \texttt{exploit}, & last\_sys > SysC_l \\ \texttt{typical}, & last\_sys <= SysC_l \end{cases}, \tag{8}$$

где $last\_sys$ — номер последнего проверенного базового блока в текущей цепочке, удовлетворяющей свойствам G1–G5, содержащий инструкцию `syscall`. Если выполняется неравенство $Sys(S,O) > SysC_l$, то при использовании правила (8) также могут возникать ложные обнаружения цепочек гаджетов. Для снижения вероятности таких событий предлагается добавлять в профиль нетипичные легитимные цепочки, для которых $Sys(S,O) > SysC_l$. Если $B'_i, ..., B'_{i+r}$ — нетипичная цепочка базовых блоков длины $r + 1 (> SysC_l)$, в которой инструкция `syscall` последний раз встречается в блоке с номером $i + SysC_l \le t \le i + r$, то в профиль должны добавляться следующие хэш-значения $h(B'_i \parallel ... \parallel B'_{i+r-j})$, $j = r - t, ..., r - SysC_l$. Этот профиль программы обозначим Π_S^{Sys}.

На основе правил (7) и (8) построим алгоритм FindChainOfGadgets (см. алгоритм 1) выявления атак повторно использования кода. В алгоритме пороговое значение C_l используется для выявления нетипично длинных цепочек базовых блоков, удовлетворяющих свойствам G1–G5, а пороговое значение $SysC_l$ используется для выявления цепочек базовых блоков, в которых блок с инструкцией `syscall` имеет

Таблица 1: Статистика длин цепочек базовых блоков, FoxitReader
Table 1: Statistics of chain lengths of basic blocks, FoxitReader

(a) Число цепочек базовых блоков, удовлетворяющих свойствам G1–G3,G5
(a) The number of chains of basic blocks that satisfy the conditions G1–G3,G5

V	L													
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
∞	4822198	425885	75179	12336	2195	1365	461	210	7	367	818	10	19	24
20	4753031	425181	74106	12101	2195	1364	462	209	7	367	818	22	9	22
15	4712871	420099	70493	12023	2241	1273	458	209	7	367	818	22	9	22
10	4506233	391030	66782	10745	2188	924	428	206	6	368	817	21	9	22
9	4392241	391112	61192	10390	2182	1058	292	205	5	366	815	21	9	22
8	4279041	290249	60446	9815	2086	816	478	10	5	366	815	21	9	22
7	3963898	247945	53062	1964	453	470	199	10	4	374	807	21	9	22
6	3818647	181893	44739	1505	631	98	198	382	3	3	807	51	0	0
5	3354795	160279	34152	982	174	95	195	2	0	0	0	12	0	0
4	2575331	100882	26002	664	27	3	0	2	0	12	0	0	0	0
3	1833687	72377	1808	433	0	1	0	0	0	0	0	0	0	0
2	965234	18570	254	1	0	0	0	0	0	0	0	0	0	0

(b) Число цепочек базовых блоков,
удовлетворяющих свойствам G1–G5
(b) The number of chains of basic blocks that
satisfy the conditions G1–G5

V	L										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
∞	167810	2550	236	47	9	8	1	2	1	3	1
20	167509	2536	227	45	9	8	1	2	1	3	1
15	167117	2476	213	41	8	8	1	2	1	3	1
10	164637	1983	153	9	6	5	1	1	1	2	0
9	163587	1847	142	7	6	5	1	1	1	2	0
8	162157	361	83	4	4	4	1	0	1	2	0
7	160648	105	21	3	3	0	0	0	1	2	0
6	159270	14	1	3	0	0	0	0	0	0	0
5	159175	9	0	0	0	0	0	0	0	0	0
4	159175	5	0	0	0	0	0	0	0	0	0
3	159174	5	0	0	0	0	0	0	0	0	0
2	145901	0	0	0	0	0	0	0	0	0	0

нетипично большой для цепочек базовых блоков порядковый номер. Параметр D_S используется для раннего выявления эксплоитов, состоящих из длинных цепочек, в которых инструкция `syscall` размещена на большом расстоянии от начала цепочки. Этот параметр индивидуален для каждой программы и может быть частью общего профиля программы S. Например, как видно из таблиц 1a и 2a, для S=FoxitReader параметр D_S может быть выбран равным 14, а для S=LibreOffice может быть выбрано значение $D_S = 7$. Параметр C_l может определяться как индивидуально для программ (в этом случае он будет частью общего профиля каждой программы), так и задаваться общим для всех программ. Например, согласно таблицам 1a и 2a, параметр C_l может быть выбран равным 6. В этом случае профиль Π_S программы LibreOffice будет пустым, а профиль программы ForitReader будет содержать

Таблица 2: Статистика длин цепочек базовых блоков, LibreOffice
Table 2: Statistics of chain lengths of basic blocks, LibreOffice

(a) Число цепочек базовых блоков, удовлетворяющих свойствам G1–G3,G5
(a) The number of chains of basic blocks that satisfy the conditions G1–G3, G5

V	L						
	1	**2**	**3**	**4**	**5**	**6**	**7**
∞	9011478	220831	189884	5503	38	5	2
20	9009423	221029	189620	5495	38	3	2
15	8964219	219859	189526	5469	37	3	2
10	8856074	214940	189370	5438	41	0	1
9	9013092	209267	78654	5425	41	0	1
8	8877798	201130	78406	5381	40	0	1
7	8846433	184202	78410	5239	20	0	1
6	4428607	152668	73059	5225	20	0	1
5	3722576	156543	70674	269	1	0	1
4	3429475	107576	66760	84	0	0	0
3	3145993	32480	60444	0	0	0	0
2	656869	177	0	0	0	0	0

(b) Число цепочек базовых блоков, удовлетворяющих свойствам G1–G5
(b) The number of chains of basic blocks that satisfy the conditions G1–G5

V	L					
	1	**2**	**3**	**4**	**5**	**6**
∞	35179	94	24	0	0	2
20	35178	94	24	0	0	2
15	33752	75	24	0	0	2
10	32989	18	0	0	0	0
9	32985	10	0	0	0	0
8	32931	10	0	0	0	0
7	32932	9	0	0	0	0
6	32932	9	0	0	0	0
5	32932	9	0	0	0	0
4	32934	5	0	0	0	0
3	32934	5	0	0	0	0
2	32376	0	0	0	0	0

только 8 нетипичных цепочек базовых блоков. Параметр V введен в алгоритм для того, чтобы имелась возможность уменьшить вероятность ложного обнаружения эксплоита, если заранее известен максимальный размер G_l гаджета, используемого в эксплоите. Если положить $G_l = 6$ (в соответствии с [10]), то как видно из таблиц 1b и 2b, возможно сокращение размера профиля (см. строки таблиц для $V \leq 6$). Если G_l заранее неизвестно, то следует использовать значение $V = \infty$.

Параметры C_l и D_S используются для раннего выявления эксплоитов, а также для выявления эксплоитов, состоящих из большого количества гаджетов, в которых не используются инструкция `syscall`. Однако в случае использования общего параметра C_l необходимо для каждой программы S хранить профиль Π_S и выполнять соответствующие проверки хэш-значений. Для исключения хранения профиля Π_S и использования правила (7) алгоритм FindChainOfGadgets может быть модифицирован так, чтобы использовалась только проверка (8). В следующем разделе проводятся эксперименты по определению параметра $SysC_l$. Профиль, состоящий из Π_S, Π_S^{Sys}, D_S и опционально из C_l может формироваться заранее, например, следующим образом: выделяется множество программ, для которых необходимо выявление эксплоитов (редакторы/просмотрщики текстовых файлов, графических файлов и т.п.), и каждая из выбранных программ запускается с входными файлами, полученными из надежных источников (без эксплоитов).

Отметим, что схожий способ защиты разработан в [22], за тем исключением, что он разработан для операционной системы Windows, в качестве гаджетов рассматриваются только те, которые завершаются инструкцией `retq` (защита только от ROP-цепочек), а также не используются профили для программ. В [22] подсчет гаджетов выполняется на основе проверки регистров LBR (Last Branch Recording): в этих регистрах сохраняются адреса возврата из функций. Если для какого-то из адресов возврата нет соответствующей инструкции `callq`, то соответствующая

команда `retq` расценивается как нетипичная. В случае появления подряд идущих нетипичных возвратов в количестве, превышающем порог, и приводящих к вызову системной API-функции, фиксируется выполнение эксплоита. Авторы [22] указывают главное ограничение своего способа: он не позволяет выявлять эксплоиты, написанные с помощью техники JOP. Также не будут выявляться цепочки, построенные с помощью техник COP/PCOP. В то же время предлагаемый в настоящей работе способ защиты от повторного использования кода позволяет выявлять цепочки гаджетов, в которых гаджеты завершаются любой инструкцией с префиксом из набора (5). Стоит отметить важное преимущество подхода из [22] – высокую скорость обнаружения эксплоитов, так как не используется эмуляция исполнения программного кода.

2. Экспериментальная оценка $SysC_l$

Для реализации разработанного способа защиты выбрана система виртуализации с открытым исходным кодом QEMU. Эта система позволяет запускать операционные системы, скомпилированные для одной архитектуры процессора (для целевой архитектуры) на процессорах другой архитектуры (на архитектуре хоста). Это достигается за счет группировки машинных инструкций для целевого процессора в базовые блоки вида (2), трансляции этих базовых блоков в промежуточный код, а затем трансляции промежуточного кода в базовые блоки, состоящие из инструкций процессора хоста. Особенностью системы QEMU является то, что описанная схема трансляции в операционных системах Linux/BSD для архитектуры x86/64 (архитектура хоста и целевая архитектура) может применяться не только для запуска операционных систем, но и для запуска приложений.

В исходные коды QEMU внесены необходимые изменения, позволяющие сохранять в отдельный файл последовательность вида (6). Для каждой инструкции сохраняется следующая информация: название инструкции, аргумент, количество машинных инструкций в базовом блоке.

Для проведения эксперимента выбраны наиболее популярные программы для просмотра pdf-файлов в операционной системе Linux Ubuntu 16.04: FoxitReader, LibreOffice, Xpdf, Okular, Evince, GNU Gv. Выбор обусловлен тем, что эксплоиты могут быть внедрены в pdf-файлы. Например, уязвимость в просмотрщике Evince [23], при открытии специально сформированного файла провести атаку типа отказа в обслуживании. Для исследования были выбраны 15 файлов формата pdf размером от 979 байтов до 1,4 МБ как с шифрованным содержимым, так и с незашифрованным. Также для сравнения полученных результатов дополнительно выбраны программы Mahjogg, Mines, Sol, Sudoku, FireFox, которые запускались без параметров. В таблице 3 для $V = \infty$ содержатся результаты нахождения $Sys(S, O)$: для всех программ в ячейках приведены максимальное и минимальное количество выявленных цепочек с заданным значением $Sys(S, O)$. Результаты для просмотрщиков pdf-файлов отделены от результатов других программ двойной линией.

Выше отмечалось, что написание коротких цепочек гаджетов, позволяющих обходить защиту ASLR, является трудной задачей. Например, в работах [20] и [21] соответствующие цепочки имеют длину не менее 4-х гаджетов. Результаты таб-

Исходные параметры: 1) Последовательность $\mathrm{Path}(S, O)$, представленная цепью базовых блоков вида (4) длины n',
2) $V \in \mathbb{N}$, 3) профиль Π_S, 4) профиль Π_S^{Sys}, 5) D_S,
6) C_l, 7) $SysC_l$.

Результат: Сообщение о наличии эксплоита (`exploit`) или отсутствии нетипичных подцепочек (`typical`)

$chain = \varnothing$, $counter = 0$, $result =$ `typical`, $syscall\_present = false$
цикл $i = 1, ..., n'$ **выполнять**
  если $B'_i \in \mathcal{B}_0^V$ **тогда**
    если `syscall` $\in B'_i$ **тогда**
      $syscall\_present = true$
    конец условия
    $counter = counter + 1$
    // Проверка длины цепочки
    если $counter > D_S$ **тогда**
      $result =$ `exploit`
      break
    конец условия
    иначе
      // Проверка (7)
      если $counter > C_l$ ***and*** $syscall\_present = true$ ***and*** $h(chain) \notin \Pi_S$ **тогда**
        $result =$ `exploit`
        break
      конец условия
      иначе
        // Проверка (8)
        если `syscall` $\in B'_i$ ***and*** $counter > SysC_l$ ***and*** $h(chain) \notin \Pi_S^{Sys}$ **тогда**
          $message =$ `exploit`
          break
        конец условия
        иначе
          $chain = chain \parallel B'_i$
        конец условия
      конец условия
    конец условия
  конец условия
  иначе
    $counter = 0$, $chain = \varnothing$, $syscall\_present = false$
  конец условия
конец цикла
возвратить $result$

Алгоритм 1: FindChainOfGadgets

лицы 3 показывают, что при $SysC_l = 4$ для большинства программ выполняется условие $Sys(S, O) < SysC_l$, и поэтому для этих программ возможно использование

только правила (8) и без составления профиля Π_S^{Sys}. В этом случае для программ FoxitReader и FireFox потребуется хранить лишь небольшие профили: из одной и восьми цепочек соответственно. Отметим, что возможно усиление защиты за счет выбора значения $SysC_l = 3$ (в открытых источниках не найдено информации об эксплоитах, состоящих из трех гаджетов и позволяющих обходить защиту ASLR). Однако в этом случае возрастают размеры профилей: для программ FoxitReader и Evince необходимо хранить соответственно три и одну цепочки, а для FireFox – не менее 30 цепочек.

Результаты показывают, что в алгоритме FindChainOfGadgets может использоваться только проверка вида (8) без использования правила (7) и без сравнения длины цепочки с D_S. Уменьшение числа проверок позволит ускорить процесс проверки файлов. Заметим, что результаты в таблице 3 согласуются с результатами работы [16], где аналогичные вычисления проводились для различных программ для операционной системы Windows. Отметим, однако, что в таблице 3 результаты получены в случае, когда системный вызов может быть любым, в то время как в [16] вычисления проводились не для всех системных вызовов, а только для защищенных, в частности для `VirtualProtect`.

Таблица 3: Распределение значений $Sys(S, O)$
Table 3: Distribution of $Sys(S, O)$

S	$Sys(S,O)$							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
FoxitReader	85063 – 170578	90 – 113	0 – 3	0 – 2	0 – 1	0	0	0
LibreOffice	16950 – 68093	57 – 73	0	0	0	0	0	0
Xpdf	123 – 425	12 – 19	0	0	0	0	0	0
Okular	6604 – 44671	56 – 105	0 – 10	0	0	0	0	0
Evince	8046 – 24716	26 – 31	0 – 1	0 – 1	0	0	0	0
GNU Gv	97 – 112	11	0	0	0	0	0	0
Mahjogg	4247	24	0	0	0	0	0	0
Mines	6807	26	0	0	0	0	0	0
Sol	4201	30	0	0	0	0	0	0
Sudoku	4824	30	0	0	0	0	0	0
FireFox	8091 – 834896	87 – 560	0 – 145	0 – 22	0 – 4	0 – 1	0 – 2	0 – 1

Заключение

Разработанный способ защиты не способен распознавать атаки повторного использования кода, в которых применяются гаджеты, завершающиеся инструкциями условного перехода с префиксами `jeq`, `jaq` и т.п. На настоящий момент в открытых источниках не найдено информации о наличии подобных эксплоитов. Также не будут распознаваться эксплоиты, в которых изменяются данные [25], так как в этих атаках могут не использоваться системные вызовы. В целом предлагаемый способ защиты не исключает написания эксплоита, не подпадающего под выделенные свойства, однако этот способ направлен на существенное затруднение написания такого кода.

Одним из существенных ограничений предлагаемой системы защиты является скорость работы, так как фактически происходит эмуляция работы процессора: за-

пуск программы с входными данными может осуществляться до нескольких минут. Однако необходимость эмуляции подтверждается исследованием, проведенным в работе [24], где показано, что статический анализ имеет ограничения в применении и не всегда позволяет обнаруживать вредоносный код. Способ может быть применим для отложенной проверки или для глубокого исследования открываемых файлов.

Отметим, что техника повторного использования программного кода применяется не только для обхода систем защиты. В последнее время эта техника рассматривается как способ создания систем защиты информации: включение цифровых водяных знаков [26], обфускация алгоритмов для защиты от исследования [27], программная стеганография [28]. Поэтому необходимо дальнейшее усовершенствование предложенного способа защиты для снижения вероятности ложного срабатывания (например, выполнение обфусцированной с помощью техники повторного использования кода программы может быть расценено как исполнение эксплоита). Разработанный способ может быть усовершенствован, например, добавлением эвристического анализа последовательности системных вызовов, использованием марковских моделей по аналогии с [8]. В частности, в предлагаемом способе в требовании G4 не накладывается ограничений на множество системных вызовов. В то же время при необходимости может быть составлен список системных вызовов, обращение к которым может свидетельствовать о наличии эксплоита. Например, появление системного вызова `clone`, `execve` после обнаруженной длинной цепочки гаджетов свидетельствуют о запуске приложения, что с большой вероятностью может свидетельствовать о выполнении эксплоита. Сократить число рассматриваемых цепочек возможно также за счет исключения из рассмотрения цепочек, представляющих собой выполнение цикла (многократное выполнение одно и того же базового блока), а также за счет исключения цепочек, в которых инструкция `syscall` встречается в первом базовом блоке.

Список литературы / References

[1] Shacham H., “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”, *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, 552–561.

[2] Buchanan E., Roemer R., Shacham H., Savage S., “When good instructions go bad: generalizing return-oriented programming to risc”, *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, 27–38.

[3] http://ropshell.com. Last access 26.11.2018..

[4] Binlin C., Jianming F., Zhiyi Y., “Heap Spraying Attack Detection Based on Sled Distance”, *International Journal of Digital Content Technology and its Applications(JDCTA)*, **6**:14 (2012), 379–386.

[5] Davi L., Sadeghi A., Winandy M., “ROPdefender: a detection tool to defend against return-oriented programming attacks”, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, 40–51.

[6] Davi L., Koeberl P., Sadeghi A., “Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation”, *Proceedings of the 51st Annual Design Automation Conference, San Francisco, CA, USA*, 2014, 1–6.

[7] Ge X., Talele N., Payer M., Jaeger T., "Fine-grained control-flow integrity for kernel software", *In IEEE European Symposium on Security and Privacy*, 2016, 179–194.

[8] Usui T., Ikuse T., Iwamura M., Yada T., "POSTER: Static ROP Chain Detection Based on Hidden Markov Model Considering ROP Chain Integrity", *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, 1808–1810.

[9] Cawan S. C. Arnold S. R., Beattie S. M., Wagle P. M., "Pointguard: method and system for protecting programs against pointer corruption attacks", *Patent US7752459B2*, 2010.

[10] Cheng Y., Zhou Z., Miao Y., Ding X., Deng H. R., "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack", *In Symposium on Network and Distributed System Security (NDSS)*, 2014, 1–14.

[11] Chen P., Xiao H., Shen X., Yin X., Mao B., Xie L., "DROP: Detecting Return-Oriented Programming Malicious Code", *Lecture Notes in Computer Science*, **5905** (2009), 163–177.

[12] "Control-flow Enforcement Technology Preview", 2017, software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. Last access 26.11.2018.

[13] Checkoway S., Davi L., Dmitrienko A., Sadeghi A. R., Shacham H., Winandy M., "Return-oriented programming without returns", *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, 559–572.

[14] Sadeghi A., Niksefat S., Rostamipour M., "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions", *Journal of Computer Virology and Hacking Techniques*, **14**:2 (2018), 139–156.

[15] Yao F., Chen J., Venkataramani G., "Jop-alarm: Detecting jump-oriented programming-based anomalies in applications", *IEEE 31st International Conference on Computer Design (ICCD)*, 2013, 467–470.

[16] Goktas E., Athanasopoulos E., Polychronakis M., Bos H., Portokalidis G., "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard", *Proceedings of the 23rd USENIX Security Symposium*, 2014, 417–432.

[17] Carlini N., Wagner D., "ROP is still dangerous: breaking modern defenses", *SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium*, 2014, 385–399.

[18] Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д., *Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ.*, М.:ООО «И.Д. Вильямс», 2008; [Aho A. V., Sethi R., Ullman J. D., *Compilers: Principles, Techniques, and Tools*, Pearson Education, Inc, 1986, (in Russian).]

[19] Kayaalp M., Schmitt T., Nomani J., Ponomarev D., Abu-Ghazaleh N., "Scrap: architecture for signature-based protection from code reuse attacks", *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, 2013, 258–269.

[20] https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-iii/. Last access 06.12.2018.

[21] Katoch V., "Bypassing ASLR/DEP.", https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf. Last access 06.12.2018..

[22] Pappas V., Polychronakis M., Keromytis A. D., "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing", *Proc. of the 22nd USENIX Security Symposium*, 2013, 447–462.

[23] https://www.securityfocus.com/bid/62780/info. Last access 03.12.2018..

[24] Moser A., Kruegel C., Kirda E., "Limits of Static Analysis for Malware Detection", *Proceedings of Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2008, 421–430.

[25] Hu H., Shinde S., Adrian S., Chua Z. L., Saxena P., Liang Z., "Data-oriented programming: On the expressiveness of non-control data attacks", *In Security and Privacy (SP) Symposium*, 2016, 969–986.

[26] Ma H., Lu K., Ma X., Zhang H., Jia C., Gao D., "Software watermarking using return-oriented programming", *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, 369–380.

[27] Gao D., "Method for obfuscation of code using return oriented programming", *Patent WO2016126206A1*, 2015.

[28] Lu K., Xiong S., Gao D., "Ropsteg: program steganography with return oriented programming", *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014, 265–272.

Kosolapov Y. V., "About Detection of Code Reuse Attacks", *Modeling and Analysis of Information Systems*, **26**:2 (2019), 213–228.

DOI: 10.18255/1818-1015-2019-2-213-228

Abstract. When exploiting software vulnerabilities such as buffer overflows, code reuse techniques are often used today. Such attacks allow you to bypass the protection against the execution of code in the stack, which is implemented at the software and hardware level in modern information systems. At the heart of these attacks lies the detection, in the vulnerable program of suitable areas, of executable code — gadgets — and chaining these gadgets into chains. The article proposes a way to protect applications from attacks that use code reuse. For this purpose, features that distinguish the chains of gadgets from typical chains of legal basic blocks of the program are highlighted. The appearance of an atypical chain of the base block during program execution may indicate the execution of a malicious code. An algorithm for identifying atypical chains has been developed. A feature of the algorithm is that it is focused on identifying all currently known techniques of re-execution of the code. The developed algorithm is based on a modified QEMU virtualization system. One of the hallmarks of the chain of gadgets is the execution at the end of the chain of instructions of the processor used to call the function of the operating system. For the Linux operating system based on the x86/64 architecture, experiments have been conducted showing the importance of this feature in detecting the execution of the malicious code.

Keywords: code reuse, software vulnerability

On the authors:
Yury V. Kosolapov, orcid.org/0000-0002-1491-524X, PhD,
Southern Federal University,
8a Milchakova str., Rostov-on-Don 344090, Russia, e-mail: itaim@mail.ru