Обзор методов автоматизированной генерации эксплойтов повторного использования кода

Вишняков А. В., ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>
Нурмухаметов А. Р., ORCID: 0000-0003-1681-1580 <nurmukhametov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В работе приводится обзор существующих методов и инструментов автоматизированной генерации эксплойтов повторного использования кода. Такие эксплойты используют код, уже содержащийся в уязвимом приложении. Подход повторного использования кода позволяет эксплуатировать уязвимости программного обеспечения при наличии защитного механизма операционной системы, который запрещает исполнение кода страниц памяти, помеченных в качестве данных. В статье приводится описание различных методов повторного использования кода: атаки возврата в библиотеку, возвратно-ориентированного программирования, переходо-ориентированного программирования и других. Дается определение базовых понятий таких, как гаджет, фрейм гаджета, каталог гаджетов. Кроме того, показывается, что гаджет по своей сути является инструкцией, а их набор задает некоторую виртуальную машину. Задача создания эксплойта сводится к задаче генерации кода для такой виртуальной машины. Набор команд виртуальной машины задается исполняемым кодом конкретной программы. В работе приводится обзор методов поиска гаджетов и определения их семантики (формирования каталога гаджетов). Они позволяют получить набор команд виртуальной машины. Если набор гаджетов в каталоге полон по Тьюрингу, то гаджеты из каталога можно использовать в качестве набора команд целевой архитектуры компилятора. Однако в каталоге гаджетов для конкретного приложения могут отсутствовать некоторые инструкции, поэтому в литературе было предложено несколько способов для замены отсутствующих инструкций несколькими гаджетами. Связывание гаджетов в цепочки может происходить как поиском гаджетов по шаблонам, задаваемым регулярными выражениями, так и с учетом семантики гаджета. Более того, существуют подходы конструирования ROP цепочек с использованием генетических алгоритмов, а также методы с использованием SMT-решателей. В статье проводится сравнение инструментов с открытым исходным кодом. Мы предлагаем тестовую систему rop-benchmark, с помощью которой была проведена экспериментальная проверка работоспособности генерируемых инструментами цепочек на специально сформированном наборе тестов.

Ключевые слова: атаки повторного использования кода; возвратно-ориентированное программирование; ROP; переходо-ориентированное программирование; JOP; предотвращение выполнения данных; DEP; рандомизация размещения адресного пространства; ASLR; гаджет; фрейм гаджета; каталог гаджетов; ROP цепочка; поиск гаджетов; классификация гаджетов; ROP компилятор; символьная интерпретация; ROP benchmark.

Благодарности: Работа поддержана грантом РФФИ № 17-01-00600.

Survey of methods for automated code-reuse exploit generation

Vishnyakov A. V., ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>
Nurmukhametov A. R., ORCID: 0000-0003-1681-1580 <nurmukhametov@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia

Abstract. This paper provides a survey of methods and tools for automated code-reuse exploit generation. Such exploits use code that already contains in a vulnerable program. The code-reuse approach allows one to exploit vulnerabilities in the presence of operating system protection that prohibits an execution of code in memory pages marked as data. This paper contains a description of various code-reuse methods: return-to-libc attack, return-oriented programming, jump-oriented programming, and others. We define fundamental terms such as gadget, gadget frame, gadget catalog. Moreover, we show that a gadget is, in fact, an instruction, and a set of gadgets defines a virtual machine. We can reduce an exploit creation problem to code generation for this virtual machine. Each particular executable file defines a virtual machine instruction set. We provide a survey of methods for gadgets searching and determining their semantics (creating a gadget catalog). These methods allow one to get the virtual machine instruction set. If a set of gadgets is Turing-complete, then a compiler can use a gadget catalog as a target architecture. However, some instructions can be absent. Hence we discuss several approaches to replace missing instructions with multiple gadgets. An exploit generation tool can chain gadgets by pattern searching (regular expressions) or taking gadgets semantics into consideration. Furthermore, some chaining methods use genetic algorithms, while others use SMT-solvers. We compare existing open source tools and propose a testing system rop-benchmark that can be used to verify whether a generated chain successfully opens a shell.

Keywords: code-reuse attacks; return-oriented programming; ROP; jump-oriented programming; JOP; data execution prevention; DEP; address space layout randomization; ASLR; gadget; gadget frame; gadget catalog; ROP chain; gadget search; gadget classification; ROP compiler; symbolic execution; ROP benchmark.

Acknowledgments: This work was supported by the Russian Foundation for Basic Research, project no. 17-01-00600.

1. Введение

Современное программное обеспечение содержит ошибки. Их наличие рассматривается некоторыми исследователями как неизбежность. Однако далеко не все ошибки возможно использовать в злонамеренных целях. Эксплуатируемые ошибки называются *уязвимостями*. Эксплуатация уязвимостей приводит к серьезным

последствиям: денежным убыткам, деградации средств коммуникации, компрометации криптографических ключей [1] и др. С развитием технологий интернета вещей окружающие нас повседневно предметы (чайники, холодильники, душевые системы и др.) могут быть подвержены эксплуатации. Особенно критичны вопросы безопасности медицинского оборудования. Halperin и др. [2] показали возможность эксплуатации имлантируемых сердечных дефибрилляторов.

Вместе с развитием безопасного цикла разработки программного обеспечения улучшаются также методы по обнаружению разнообразных программных дефектов. В ответ на усовершенствование методов защиты от эксплуатации уязвимостей разрабатываются новые методы их обхода и эксплуатации. Поэтому необходимо знать и понимать, как устроены методы как защиты, так и атаки на программное обеспечение. Более того, для приоритетного исправления уязвимостей вендоры и разработчики программного обеспечения требуют подтверждающие примеры — эксплойты.

Переполнение буфера на стеке является, пожалуй, одним из самых эксплуатируемых программных дефектов [3]. Это объясняется сравнительной легкостью в использовании этого дефекта для перехвата потока управления под контроль атакующего. В простейшем случае полного отсутствия защит эксплуатация происходит следующим образом. Адрес возврата, расположенный на стеке выше локального буфера, перезаписывается контролируемым значением. Данное значение указывает обратно внутрь буфера, где располагается код, который хочет исполнить атакующий.

Для противодействия исполнению кода, расположенного в буфере на стеке, появилась защита DEP. Она позволила запретить исполнять определенные регионы памяти процесса, такие как стек и куча. И, в свою очередь, запретить писать в регионы памяти, помеченные для исполнения. Данная защита положила конец эпохи инъекции кода в память процесса. Атакующие оказались ограничены в своих возможностях исполнять только код, имеющийся в памяти процесса.

В ответ на повсеместное распространение DEP начали активно развиваться атаки повторного использования кода. Первым таким типом атаки являлась атака возврата в библиотеку [4]. На стек помещается адрес функций для вызова на место адреса возврата, а за ним следом размещаются аргументы функции. Обобщением данного метода атаки стало развитие методов возвратно-ориентированного программирования [5—7]. При возвратно-ориентированном программировании вместо функций выступают короткие последовательности инструкций, заканчивающиеся инструкцией возврата и называемые *гаджетами*. Гаджеты связываются в цепочки так, чтобы они последовательно передавали друг другу управление и осуществляли в совокупности некоторую вредоносную нагрузку. Shacham [5] дал определение понятию гаджет и привел первый каталог гаджетов, для которого была показана полнота по Тьюрингу для архитектуры набора команд x86. В дальнейшем была показана применимость возвратно-ориентированного программирования и для других архитектур набора команд: ARM [8—12], SPARC [13], Atmel AVR [14], PowerPC [15], Z80 [16],

MIPS [15]. В работах [9, 17—19] было показано, что можно использовать гаджеты, которые оканчиваются не только инструкциями возврата.

Вместе с развитием методов повторного использования кода происходило и развитие инструментов, с помощью которых атакующий конструировал атаки данного типа. Вначале это процесс был практически ручным, но со временем он постепенно автоматизировался. В данный момент в литературе представлен набор подходов к автоматизированному построению эксплойтов повторного использования кода [6, 7, 11—13, 18, 20—26]. Кроме того, для некоторых из них даже доступны инструменты [27—38].

Данная работа ставит своей целью детальное изучение имеющихся методов и инструментов автоматизированной генерации эксплойтов повторного использования кода с целью определения сильных и слабых сторон каждого из них, а также выявления перспективных направлений исследования в данной области.

Возвратно-ориентированное программирование может быть использовано как стеганографический метод [39]. Ntantogian и др. [40] предлагают использовать методы повторного использования кода для сокрытия вредоносной функциональности от обнаружения антивирусными средствами, а Mu и др. [41] в целях обфускации кода. Методами повторного использования кода в том числе могут быть заложены недокументированные возможности в программное обеспечение [42, 43].

Кроме практической применимости, рассматриваемые в статье методы и инструменты могут иметь и научный интерес. Задача автоматизированной генерации эксплойтов для атак повторного использования кода является задачей трансляции некоторого описания эксплойта в архитектуру набора команд виртуальной машины, неявно задаваемой состоянием памяти эксплуатируемого процесса. В качестве инструкций набора команд выступают гаджеты, расположенные в памяти процесса. Причем заранее неизвестно, какой набор инструкций предоставляет эксплуатируемый исполняемый файл. Для его определения необходимо найти все гаджеты, а затем произвести процедуру определения их функциональности (семантики). В результате, формируется каталог гаджетов, где описана их семантика. Каталог гаджетов является входными данными для инструмента, который генерирует эксплойт. Генерирующий эксплойт инструмент должен учитывать тот факт, что в наборе гаджетов, в отличии от инструкций процессора, могут отсутствовать некоторые инструкции, а другие — иметь нетривиальные побочные эффекты. Все это усложняет построение инструментов автоматической генерации эксплойтов возвратно-ориентированного программирования.

Данная работа устроена следующим образом. В разделах 2–13 проводится обзор атак и защитных механизмов. В разделе 14 описывается общая схема генерации эксплойтов повторного использования кода. В разделе 15 вводится определение *каталога гаджетов*. В разделе 16 описываются подходы к поиску гаджетов. В разделе 17 приводятся методы определения семантики гаджетов. В разделе 18 проводится обзор методов генерации цепочек гаджетов. В разделе 19 освещается проблема учета запрещенных символов в цепочках. Экспериментальное сравнение инстру-

ментов с открытым исходным кодом проводится с использованием разработанной нами тестовой системы rop-benchmark [44] в разделе 20. В последнем разделе 21 обсуждаются проблемы существующих методов и выделяются дальнейшие направления исследований.

2. Предотвращение выполнения данных

Предотвращение выполнения данных — защитный механизм операционной системы, который запрещает исполнение кода со страниц памяти, помеченных как «данные». Страница памяти не может быть одновременно доступна и на запись, и на исполнение, что точно отражено в названии политики безопасности OpenBSD W^X [45] (**W**rite XOR e**X**ecute). На Windows этот защитный механизм называется DEP (Data Execution Prevention) [46]. Linux [47] и Mac OS X имеют схожие защитные механизмы. Защитный механизм реализуется аппаратно с использованием специального NX-бита (**N**o e**X**ecute), которым помечаются недоступные на исполнение страницы. Если в процессоре отсутствует аппаратная поддержка NX-бита, то этот механизм эмулируется программно.

При классической эксплуатации переполнения буфера на стеке [48], атакующий внедряет в буфер вредоносный код и передает на него управление. Защита не позволит исполнить внедренный код, т.к. он находится на стеке, который помечен как «данные».

3. Атаки повторного использования кода

Атаки повторного использования кода появились для обхода защиты, предотвращающей выполнение данных. Идея заключается в том, чтобы не внедрять вредоносный код, а повторно использовать уже присутствующий в программе и библиотеках код для реализации функциональности вредоносного кода. Уязвимость переполнения буфера на стеке или возможность у атакующего писать произвольное значение в произвольное место памяти (write-what-where [49]) позволяют подменить адрес возврата из функции адресом некоторого кода из адресного пространства программы. Таким образом, после возврата из функции управление передастся на этот код.

4. Атака возврата в библиотеку

Александр Песляк первым показал, что эксплуатация возможна даже при неисполняемом стеке, и предложил атаку возврата в библиотеку (return-to-libc) [4]. Атакующий подменяет адрес возврата адресом некоторой библиотечной функции и размещает ее аргументы выше по стеку. Например, атакующий может вызвать `system("/bin/sh")` из стандартной библиотеки `libc`. Таким образом, атакующий откроет командный интерпретатор операционной системы.

5. Рандомизация размещения адресного пространства

Рандомизация размещения адресного пространства (address space layout randomization, ASLR) [50] — защитный механизм операционной системы, который загружает сегменты памяти по различным базовым адресам для каждого запуска программы. Наличие данной защиты затрудняет проведение атаки возврата в библиотеку (return-to-libc), т.к. базовый адрес загрузки библиотеки `libc` рандомизируется и адрес функции `system` неизвестен до загрузки программы. Однако для совместимости с ASLR программа должна быть скомпилирована в позиционно-независимый код [51], что не всегда выполняется. Например, в Linux рандомизируется адрес загрузки динамических библиотек, стека и кучи, а адрес загрузки образа программы часто остается постоянным [52].

Если адрес загрузки библиотеки рандомизирован, а образ программы нет, то атакующий может вызвать импортированную функцию через таблицу динамического связывания PLT [53], которая содержит код для вызова библиотечных функций. Атака возврата в таблицу связывания (return-to-plt) является модификацией атаки возврата в библиотеку и заключается в подмене адреса возврата адресом кода из PLT, который вызовет функцию из динамической библиотеки.

6. Возвратно-ориентированное программирование

Shacham [5] предложил термин возвратно-ориентированное программирование (return-oriented programming, ROP). ROP является эффективным средством обхода предотвращения выполнения данных (DEP [46], W^X [45]). В некотором смысле данный подход является обобщением атаки возврата в библиотеку. Однако вредоносная нагрузка осуществляется не вызовом одной функции, а формируется из нескольких уже присутствующих в программе кусочков кода, которые называются *гаджетами*. Гаджет — это последовательность инструкций, заканчивающаяся инструкцией передачи управления. Каждый гаджет изменяет состояние регистров и памяти вычислительной машины. Например, складывает значения двух регистров и записывает результат в третий. Атакующий, изучив все имеющиеся в программе гаджеты, связывает их в *цепочки*, в которых гаджеты последовательно передают управление друг другу. Суммарная вредоносная нагрузка реализуется такой цепочкой гаджетов. При достаточном количестве гаджетов атакующим может быть сформирован полный по Тьюрингу набор, который позволит реализовывать произвольные вычисления [5]. Следует отметить, что ROP может также применяться при частичной рандомизации адресного пространства. Тогда используются гаджеты из нерандомизированных областей памяти.

Для наглядности приводится пример нескольких гаджетов для x86 в таблице 1, где представлен ассемблерный код[1] инструкций трех гаджетов. Каждый из гаджетов заканчивается инструкцией передачи управления `ret`, которая позволяет передавать управление следующему гаджету через адрес, размещаемый на стеке.

[1] Здесь и далее мы будем использовать синтаксис Intel для x86 ассемблера.

Табл. 1. Пример гаджетов для x86
Table 1. Example of x86 gadgets

`mov eax, ebx ; ret`	Копирование значения регистра `ebx` в регистр `eax`
`pop ecx ; ret`	Загрузка на регистр `ecx` значения со стека
`add eax, ebx ; ret`	Прибавление к регистру `eax` значения регистра `ebx`

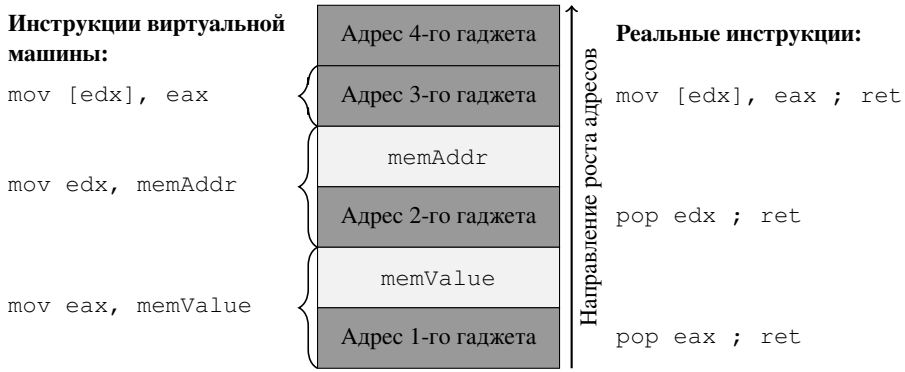


Рис. 1. Цепочка гаджетов, осуществляющая запись значения `memValue` по адресу `memAddr`
Fig. 1. A ROP Chain, storing `memValue` to `memAddr`

Архитектура x86 является CISC. Инструкции x86 имеют нефиксированную длину и каждая инструкция может выполнять несколько других низкоуровневых команд. Количество команд настолько велико и они закодированы так плотно, что практически любая последовательность байтов декодируется в корректную инструкцию. Кроме того, из-за различных длин команд (от 1 байта до 15) архитектура x86 не требует выравнивания инструкций. С точки зрения ROP это означает следующее. Набор гаджетов в программе не ограничивается только инструкциями, которые были сгенерированы компилятором. Этот набор расширяется за счет инструкций, не присутствовавших в исходной программе и полученных при доступе в середину других команд. Пример, иллюстрирующий это, приводится ниже [54]:

```
f7c7070000000f9545c3 → test edi, 0x7 ;
                       setnz BYTE PTR [ebp-0x3d]
  c7070000000f9545c3 → mov DWORD PTR [edi], 0xf000000 ;
                       xchg ebp, eax ; inc ebp ; ret
```

Набор гаджетов, который можно использовать при составлении ROP цепочки по сути задается исполняемым файлом. И для другого исполняемого файла ROP цепочку придется собирать заново. ROP цепочку можно рассматривать как программу для некоторой виртуальной машины, задаваемой исполняемым файлом [55]. Указатель

стека выполняет роль счетчика инструкций этой виртуальной машины. Коды операций (адреса гаджетов) и их операнды размещаются на стеке. Graziano и др. [56] даже предложили инструмент для трансляции ROP цепочек в обычную программу x86. На рисунке 1 приводится пример размещенной на стеке цепочки гаджетов, осуществляющей запись значения `memValue` по адресу `memAddr`. Коды операций (адреса гаджетов) размещаются от адреса возврата на стеке и закрашены темно-серым. Операнды `memValue` и `memAddr` закрашены светло-серым. Фигурными скобками обозначены инструкции виртуальной машины (код команды и ее операнды). Реальные инструкции гаджета на машине x86 приводятся справа. В начало цепочки, где при нормальном исполнении размещается адрес возврата из функции, помещается код команды — адрес первого гаджета. Затем располагается операнд `memValue`, который первый гаджет загрузит на регистр `eax`. Затем следует адрес второго гаджета, которому передаст управление первый гаджет на инструкции `ret`, и так далее.

Инструкции возврата на x86 кодируются как: `c3`, `c2**`, `cb`, `ca**` (где вместо звездочек могут быть любые байты). Кодирование инструкции возврата таким образом приводит к тому, что в коде для x86 очень много гаджетов. Даже бинарные файлы сравнительно небольшого размера содержат практически применимые с точки зрения атакующих наборы гаджетов. Schwartz и др. [6, 57] приводят статистику, что среди программ размером больше 100KB около 80 % содержат наборы гаджетов, которые позволяют вызывать любую функцию из динамически скомпонованной с уязвимым приложением библиотеки.

В дальнейшем применение ROP было успешно продемонстрировано для других архитектур набора команд: SPARC [13], Z80 [16] (машина для голосования Гарвардской архитектуры конца 80-х годов), ARM [8—12, 58]. В перечисленных работах было показано, что на RISC архитектурах возможно сконструировать как применимый для эксплуатации, так и полный по Тьюрингу набор гаджетов. RISC архитектуры часто характеризуются фиксированной длиной команд, требованием к выравниванию инструкций по их размеру и упрощенным доступом к памяти (к памяти как правило обращаются только инструкции сохранения и загрузки). Требование к выравниванию инструкций приводит к тому, что по сравнению с x86 остаются только гаджеты, оканчивающимися инструкциями возврата, которые изначально содержались в программе.

6.1 Фрейм гаджета

Для размещения ROP цепочки на стеке удобно ввести понятие *фрейма гаджета* [59, 60] аналогично стековому кадру x86. Цепочка гаджетов собирается из фреймов. Фрейм гаджета содержит в себе значения параметров гаджета (например, значение, загружаемое на регистр со стека) и адрес следующего гаджета. Начало фрейма определяется значением указателя стека перед выполнением первой инструкции гаджета. На рисунке 2 фигурной скобкой обозначены границы фрейма гаджета `pop eax ; ret 8`. Гаджет загружает значение со стека в `eax` по смещению 0 от начала фрейма. Гаджет имеет размер фрейма `FrameSize = 16`,

Рис. 2. Фрейм гаджета `pop eax ; ret 8`
Fig. 2. `pop eax ; ret 8` *gadget frame*

а адрес следующего гаджета располагается по смещению 4 от начала фрейма (`NextAddr = [esp + 4]`).

6.2 Атака возврата в рандомизированную библиотеку

Roglia и др. [61] показали, как с помощью ROP вызвать функцию из библиотеки, несмотря на то, что базовый адрес загрузки библиотеки рандомизируется (адрес загрузки образа уязвимой программы при этом не рандомизируется). Для выполнения динамического связывания Linux хранит адреса импортированных функций в секции `.plt.got` [53] (в Windows есть аналогичный механизм через таблицу импорта Import Address Table [62]). Эта информация может быть использована атакующим для вычисления адресов оставшихся функций из динамически связанных библиотек. Предположим, что в `.plt.got` содержится адрес импортированной функции `open` из `libc`. Тогда адрес функции `system` может быть вычислен по следующей формуле:

$$system = open + (offset(system) - offset(open)),$$

где функция $offset(s)$ возвращает смещение виртуального адреса функции s относительно базового адреса загрузки библиотеки. Дело в том, что ASLR рандомизирует базовый адрес загрузки библиотеки, а значение смещения функции `system` относительно `open` внутри библиотеки ($offset(system) - offset(open)$) остается постоянным и заранее известно атакующему.

Атакующий составляет ROP цепочку, которая загрузит из `.plt.got` адрес функции `open`, прибавит к загруженному адресу заранее известное смещение `system` относительно `open` и передаст управление на вычисленный адрес, т.е. на функцию `system`. Также атакующий может составить цепочку, которая прибавит к адресу

функции `open` в памяти `.plt.got` необходимое смещение и вызовет функцию по адресу из этой памяти. Если же необходимо вызвать импортированную функцию, то может быть составлена ROP цепочка, которая вызовет функцию по ее адресу из `.plt.got`, или использована атака return-to-plt (разд. 5), где код в PLT вызовет эту функцию.

Следует отметить, что в Linux используется механизм ленивого связывания. Изначально в `.plt.got` вместо адресов импортированных функций записывается адрес функции-заглушки, которая в свою очередь при первом вызове импортированной функции осуществит ее динамическое связывание и запишет ее виртуальный адрес в `.plt.got`. Таким образом, вычисление адреса функции `system` нужно производить на основе адреса уже вызванной на момент эксплуатации функции из libc, т.е. адрес которой уже записан в `.plt.got`.

Для защиты `.plt.got` от перезаписи существует флаг `LD_BIND_NOW`, который отключает ленивое связывание и указывает загрузчику незамедлительно производить связывание всех импортируемых функций [63]. Однако чтение из `.plt.got` по прежнему возможно и можно вычислять адрес `system` на основе адреса любой импортированной функции.

Kirsch и др. [64] показали, что даже при включенных защитах динамический загрузчик (POSIX) оставляет в программе указатели на функции, вызываемые при выходе из программы, которые доступны на запись. Атакующий может перезаписать эти указатели так, чтобы при выходе из программы исполнился вредоносный код.

6.3 Использование гаджетов из рандомизированной библиотеки

Гаджетов в уязвимом исполняемом файле не всегда хватает для осуществления вредоносной нагрузки. Например, могут отсутствовать гаджеты для загрузки аргументов функции, передаваемых через регистры. Ward и др. [65] предложили метод, позволяющий использовать гаджеты из динамически связанных библиотек, чей адрес загрузки рандомизируется. Предполагается, что адрес загрузки исполняемого файла уязвимой программы при этом не рандомизируется.

Суть идеи опирается на способность осуществлять частичную перезапись указателей из таблицы глобальных смещений (GOT [53]). Такая перезапись может осуществляться, например, при write-what-where [49] уязвимости. Данная таблица содержит значения указателей на код в памяти процесса (как правило, находящийся в библиотеках). Изменение последнего байта значения указателя позволяет адресовать код в пределах параграфа памяти вокруг этого указателя. Например, если значение указателя 0xdeadbeef, то для адресации доступен интервал адресов 0xdeadbe00–0xdeadbeff. Рандомизация адресного пространства, работающая на уровне изменения таблиц виртуальной памяти, меняет только старшие байты адресов. Таким образом, расположенный на одной странице код не меняет своих младших байтов. Из чего следует, что переписывание младшего байта указателя

на код позволяет позиционно-независимо адресовать код в пределах параграфа памяти размером $2^8 = 256$ байтов.

После того, как младшие байты указателей в таблице глобальных смещений (GOT) исправлены, необходимо передать на них управление. Этого можно добиться путем расположения на стеке указателей на записи таблицы связывания процедур (PLT [53]), которые косвенно передают управление по адресам, записанным в соответствующих ячейках таблицы GOT. Стоит отметить, что можно использовать только записи тех функций, адреса которых были заполнены динамическим загрузчиком (т.е. тех функций, которые вызывались хотя бы раз до момента эксплуатации в программе). Тем самым реализуется вызов гаджетов, которые лежат в пределах одного параграфа памяти от начала функции. Таким образом, можно вызывать гаджеты из рандомизированной библиотеки.

6.4 Гаджеты-трамплины

Dino Dai Zove [66] ввел понятие *гаджета-трамплина* (*Stack Pivot*), который может быть использован при эксплуатации переполнения буфера на стеке или на куче как промежуточное звено. Гаджет-трамплин перемещает указатель стека на начало ROP цепочки и тем самым передает на нее управление. Гаджеты-трамплины бывают следующие:

- `mov esp, eax ; ret`
- `xchg eax, esp ; ret`
- `add esp, <константа> ; ret`
- `add esp, eax ; ret`

Атакующий может подменить указатель функции адресом гаджета-трамлина, например, с помощью сформированной на куче поддельной таблицы виртуальных функций. Вместо вызова функции гаджет-трамплин переместит указатель стека на начало ROP цепочки.

6.5 Обход «канарейки»

Для защиты от эксплуатации уязвимости переполнения буфера на стеке компилятор использует «канарейки» [67]. При вызове функции компилятор непосредственно перед адресом возврата на стеке вставляет произвольное значение — «канарейку». Перед возвратом из функции вставляется код, который проверяет значение «канарейки». Если значение изменилось, то программа аварийно завершается. Таким образом, разместить от адреса возврата и выполнить ROP цепочку становится невозможным, т.к. это приведет к перезаписи и изменению значения «канарейки».

Федотов и др. [52] показали, как обойти «канарейку» при работающем защитном механизме, предотвращающем выполнение данных. Метод может быть применен, если присутствует уязвимость write-what-where [49]:

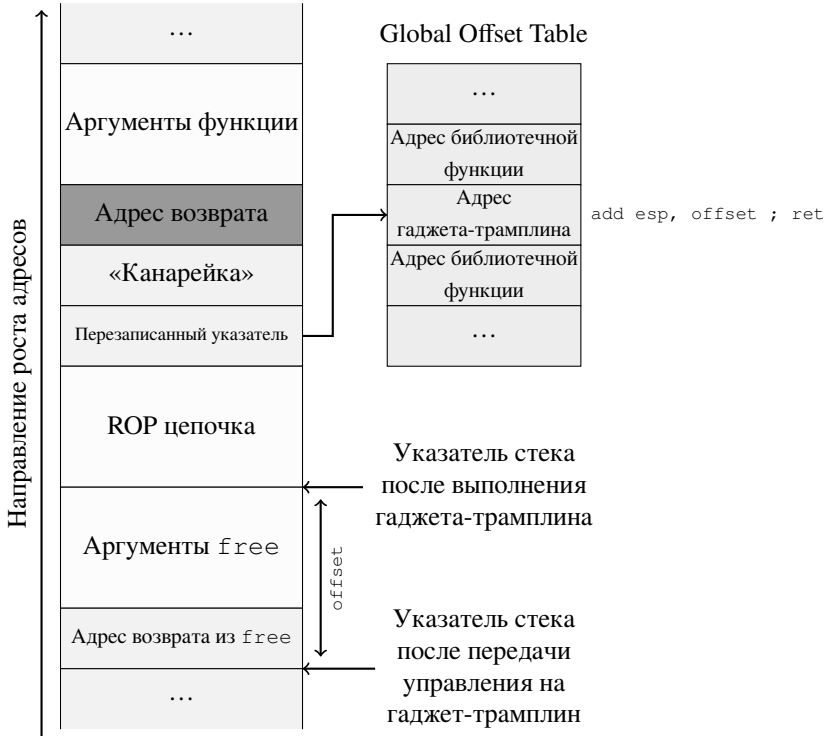


Рис. 3. Обход «канарейки»
Fig. 3. Stack canary bypass

1. Переполнение буфера приводит к перезаписи указателя, размещенного на стеке.
2. Атакующий контролирует значение, которое записывается по этому указателю.

Пусть после переполнения и до проверки значения «канарейки» вызывается функция `free` (рис. 3). Тогда атакующий перезаписывает указатель адресом ячейки из таблицы GOT, в которой хранится адрес функции `free`. В ячейку функции `free` в таблице GOT записывается адрес гаджета-трамплина, который сдвинет указатель стека и передаст управление на ROP цепочку, размещенную выше на стеке, но до «канарейки». Таким образом, вместо вызова функции `free` управление передается на гаджет-трамплин, который в свою очередь передает управление на ROP цепочку. При этом значение «канарейки» не изменяется, и проверка ее значения при возврате из функции пройдет успешно.

6.6 Отключение DEP и передача управления обычному шелл-коду

Распространен двухстадийный способ эксплуатации [66]. Первая стадия осуществляется с использованием ROP. Она отвечает за размещение шелл-кода для второй стадии, отключение защит и передачу управления на размещенный шелл-код. На второй стадии исполняется обычный шелл-код, который содержит основную вредоносную нагрузку. Таким образом, можно легко изменять вредоносную нагрузку эксплойта, подменив только шелл-код из второй стадии. Ниже приводится подробное описание обеих стадий:

1. **ROP стадия.** Атакующий размещает шелл-код на стеке или же записывает его в память с использованием ROP цепочки. Далее атакующий составляет ROP цепочку, которая отключит DEP: вызов функции `mprotect` [68] (`VirtualProtect` [69]) сделает размещенный шелл-код исполняемым. В итоге управление передается на обычный шелл-код.
2. **Шелл-код стадия.** Вредоносная нагрузка содержится в шелл-коде, который теперь является исполняемым. Выполнение шелл-кода завершает эксплуатацию.

Peter Van Eeckhoutte [70] описал способ обхода DEP в 32-битных Windows программах с использованием гаджета `pushad ; ret`. Идея заключается в том, что регистры предварительно инициализируются значениями так, чтобы после выполнения инструкции `pushad` (которая сохраняет регистры общего назначения на стек) на стеке оказалась обыкновенная ROP цепочка. ROP цепочка, в свою очередь, вызовет функцию `VirtualProtect`, чтобы сделать стек исполняемым, и передаст управление на обычный шелл-код, размещенный выше на стеке. Подробное описание этого способа и рисунок можно найти в статье [60].

7. Применение ROP при наличии DEP и ASLR

В определенных условиях возможно построить атаку повторного использования кода на приложение, бинарный код которого отсутствует у атакующего. Bittau и др. [71] приводят пример такой атаки — BROP (blind return-oriented programming). Модель атаки в данной работе полагает, что атакуемый веб-сервис обрабатывает каждый входящий запрос в отдельном процессе, который порождается системным вызовом `fork`. Это означает, что карта памяти процессов обработчиков не меняется. Данный факт предоставляет атакующему возможность динамически изучать атакуемый веб-сервис.

Авторы работы показывают, что в таких условиях возможно реализовать динамический поиск гаджетов в памяти атакуемого процесса. Поиск гаджетов осуществляется путем наблюдения за побочными эффектами выполнения атакуемой программы. Вместо адреса возврата на стек функции, содержащей переполнение буфера, помещается тестовое значение адреса. При выходе из уязвимой функции управление передается на этот адрес. Концептуально в этот момент может произойти два принципиальных события: падение или пауза. Падение происходит при подаче любого адреса, который находится за пределами исполняемых областей памяти. Пауза наступает в результате задержки выполнения программы, например, после вызова функции `sleep`. Оба события легко наблюдаются через состояние соединения с сервером. Соединение закрывается или остается открытым некоторое время соответственно. Адрес инструкции, вызывающей паузу, принципиально важен для описываемого метода атаки, поскольку он позволяет находить и классифицировать ROP гаджеты в динамике.

Атакующий обнаруживает:

1. S — адрес, приводящий к паузе выполнения программы.
2. T — адрес, заведомо приводящий к аварийному завершению программы.

Для поиска гаджетов берется пробный адрес P. Если составленная атакующим цепочка с пробным адресом приводит к паузе с дальнейшим падением, то тем самым атакующий определяет пробный гаджет к некоторому классу. Ниже приводятся примеры таких цепочек:

- $P, S, T, T \ldots$ — найдет гаджеты, которые не снимают со стека значений, такие как `ret` и `xor rax, rax ; ret`;
- $P, T, S, T, T \ldots$ — найдет гаджеты, которые снимают со стека ровно одно слово, такие как `pop rax ; ret` и `pop rdi ; ret`;

Конкретное определение регистров, используемых каждым гаджетом загрузки, производится по побочным эффектам вызовов функций из таблицы связывания (которую также обнаруживают специальной процедурой [71]) и системных вызовов (`syscall`). Конечной целью построения цепочки является вызов системного вызова `write`, который считывает из памяти образ исполняемого файла и отправляет по сети атакующему. Он производит подробный анализ исполняемого файла и конструирует ROP цепочку, выполняющую нужные ему действия.

Snow и др. [72] предложили другой пример построения ROP цепочки для приложения, бинарный код которого отсутствует у атакующего — JIT-ROP. Отличительной особенностью данной работы являются условия модельной атаки. Авторы полагают, что в атакуемой системе присутствует набор современных средств защиты, таких как DEP, ASLR и даже мелкозернистая рандомизация адресного пространства при каждом запуске приложения [73]. Однако авторы полагают также, что в атакуемом приложении присутствуют множественные утечки, раскрывающие адресное пространство процесса. Пример приводимой атаки описывается для браузеров IE. Атака реализуется, например, через вредоносный JavaScript код, загруженный браузером вместе с веб-страницей. В такой постановке у атакующего есть возможность строить ROP цепочку только непосредственно во время атаки. Все методы поиска, классификации гаджетов должны быть достаточно легковесны, чтобы их можно было разместить в коде скрипта и их выполнение не нагрузило критическим обра-

зом атакуемую вычислительную машину. Для достижения этого авторы адаптировали предложенные Schwartz и др. [6] алгоритмы, заменив метод классификации на собственный эвристический алгоритм, достаточно хорошо работающий в условиях присутствия в адресном пространстве браузера огромного количества бинарного кода.

Göktas и др. [74] предложили подход формирования ROP цепочки, который работает в условиях наличия DEP и рандомизации бинарного образа и всех библиотек. Их подход опирается на идею, которую они называют «массаж» стека. Ключевая идея заключается в том, что при выполнении программы на стек записываются указатели на код (как минимум адреса возврата, а иногда и локальные переменные, содержащие указатели на функции). При возврате из функции записанные данные не очищаются и остаются там до последующих вызовов, которые просто перезаписывают их некоторыми новыми значениями. Кроме того, неинициализированные локальные переменные оставляют значения, записанные на стеке предыдущими вызовами. В итоге аккуратно подобранными входными данными атакующий формирует в пространстве, расположенном ниже стека уязвимой функции, последовательность указателей на код, которые перемежаются местом для данных. Затем, используя уязвимость записи отдельного значения за пределы массива, он исправляет младшие байты указателей так, чтобы они указывали на ROP гаджеты. При необходимости таким же изменениям подвергаются параметры гаджетов. При построении цепочек авторы по аналогии с работой Ward и др. [65] ограничены параграфом памяти относительно указателей, расположенных на стеке программы. Это заставляет их использовать в том числе гаджеты, оканчивающиеся на инструкции вызова `call`. Главным недостатком данного метода является сложность и неавтоматизированность процедуры поиска и формирования такого пути исполнения программы, который бы установил значения ниже по стеку таким образом, чтобы там сформировался скелет будущей ROP цепочки.

8. Переходо-ориентированное программирование

Переходо-ориентированное программирование [17] (jump-oriented programming, JOP) использует в качестве гаджетов последовательности инструкций, оканчивающихся инструкциями перехода по контролируемому атакующим адресу. В случае x86 это такие инструкции, как `jmp eax` и `jmp [eax]`. Отличительной особенностью переходо-ориентированного программирования является более сложная по сравнению с возвратно-ориентированным программированием процедура передачи управления от одного гаджета к другому.

Bletsch и др. [17] устраивают передачу управления для JOP следующим образом:

- Вводится специальный тип гаджета, называемый *гаджетом-диспетчером*, который является связующим звеном между функциональными гаджетами. Данный гаджет поддерживает виртуальный счетчик команд и исполняет JOP программу путем перемещения этого счетчика с одного функционального

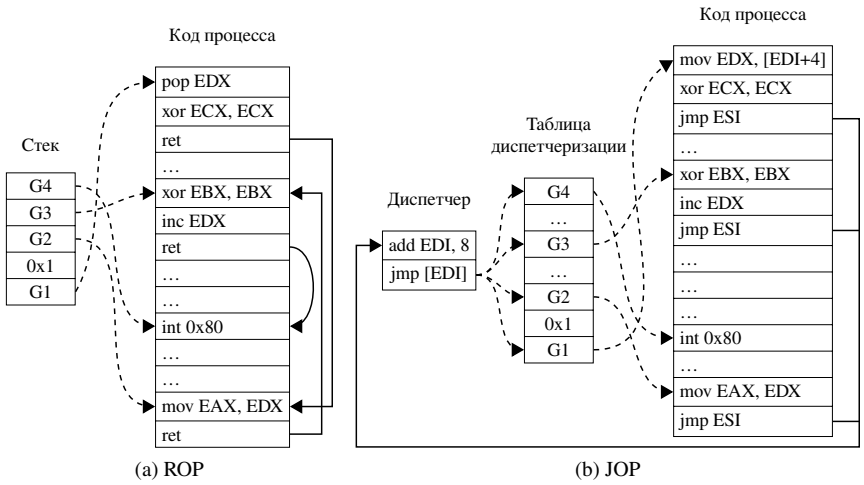


Рис. 4. Сравнение потока управления ROP и JOP на примере цепочки, осуществляющей системный вызов `exit`

Fig. 4. Comparison of control flow for ROP and JOP with example chain invoking `exit` *system call*

гаджета на другой. Адреса функциональных гаджетов хранятся в таблице диспетчеризации. В качестве счетчика команд используется некоторый фиксированный регистр, значение которого указывает на ячейку таблицы диспетчеризации с адресом текущего гаджета. Например, таким гаджетом является `add edx, 4 ; jmp [edx]` (`edx` — виртуальный счетчик команд).

- Гаджеты, выполняющие примитивные вычислительные операции, называются *функциональными*. Каждый функциональный гаджет обязан возвращать управление на гаджет-диспетчер. Например, возвращение управления может быть реализовано переходом по регистру, значение которого всегда равно адресу гаджета-трамплина (`pop eax ; jmp esi` — гаджет загрузки значения в `eax`).

На рисунке 4 схематично представлена модель передачи управления в переходо-ориентированной цепочке 4b по сравнению с возвратно-ориентированной цепочкой 4a на примере нагрузки, осуществляющей системный вызов `exit(0)`. JOP цепочка состоит из набора адресов гаджетов и значений их параметров, записанных в памяти, и представляет собой таблицу диспетчеризации с описанным потоком управления. Адреса гаджетов по сути являются кодами операций в виртуальной машине, задаваемой состоянием памяти атакуемого процесса. ROP цепочка хранится на стеке, и регистр `esp` выступает в качестве счетчика команд. JOP отличается тем,

что место расположения цепочки и регистр, выступающий в качестве счетчика команд, могут быть любыми. В случае JOP на рисунке 4b в роли гаджета-диспетчера выступает последовательность инструкций `add edi, 8 ; jmp [edi]`. В роли счетчика команд выступает регистр `edi`, который увеличивается на 8 при каждом вызове диспетчера. JOP цепочка хранится в памяти и представляет собой таблицу диспетчеризации, с помощью которой гаджет-диспетчер последовательно вызывает функциональные гаджеты (G1, G2, G3, G4). Каждый функциональный гаджет заканчивается на `jmp ESI`. Это позволяет изначально задать значение регистра `ESI` таким образом, чтобы оно указывало на гаджет-трамплин.

Формально описать гаджет-диспетчер можно следующим образом:

$$pc \longleftarrow f(pc); goto *pc;$$

Где в качестве pc может выступать регистр или некоторый адрес в памяти, а $f(pc)$ — любая функция, которая изменяет pc предсказуемым и монотонным образом.

В работах [9, 18] для передачи управления от одного функционального гаджета к другому используется JOP гаджет-трамплин (не путать с разд. 6.4). В качестве гаджета-трамплина могут выступать следующие инструкции:

```
pop eax ; jmp [eax]
```

Данный гаджет поочередно вызывает функциональные гаджеты. Функциональные гаджеты могут представлять собой следующие последовательности инструкций:

```
pop ebx ; jmp [edx]
pop ecx ; jmp [edx]
add ebx, 4 ; jmp [edx]
```

Каждый функциональный гаджет выполняет некоторую базовую вычислительную операцию и обязательно возвращает управление в гаджет-трамплин. Для этого в приведенном примере инструкций в регистре `edx` хранится адрес памяти, по которому хранится адрес гаджета-трамплина. При таком подходе увеличивается регистровое давление, что может затруднить конструирование цепочек для x86, но для некоторых других архитектур набора команд с большим количеством регистров это может не являться проблемой.

На ARM гаджетом-трамплином может выступать следующий гаджет:

```
adds r6, #4; ldr r5, [r6, #124]; blx r5.
```

Chen и др. [18] показали, что из набора JOP гаджетов из реальных библиотек можно построить полный по Тьюрингу набор инструкций.

Кроме инструкций перехода, по крайней мере на архитектуре x86, существуют инструкции вызова `call eax` и `call [eax]`. Гаджеты, которые заканчиваются на такие инструкции, также могут быть использованы в конце ROP и JOP цепочек [17].

Однако задача составления цепочки только из таких гаджетов требует особого подхода, который описан в работе PCOP [19]. Sadeghi и др. [19] показывают, что прямое применение метода конструирования цепочек с гаджетом-диспетчером, как у Bletsch и др. [17], неприменимо в данном случае. При выполнении инструкции вызова происходят две операции:

1. Значение адреса следующей инструкции (адрес возврата) кладется на стек.
2. Происходит передача управления на значение адреса, указанного в инструкции.

Главной проблемой при использовании гаджетов, заканчивающихся вызовом, являются адреса возврата, которые помещаются на стек. Их необходимо убирать со стека последующими гаджетами. Sadeghi и др. [19] предлагают использовать для этого *сильные гаджеты-трамплины*. Такие гаджеты необходимо располагать между функциональными гаджетами для передачи управления и очищения стека от ненужных значений адресов возврата. В PCOP цепочке, состоящей из n функциональных гаджетов, необходимо разместить $n-1$ сильных гаджетов-трамплинов между функциональными гаджетами. Примером сильного гаджета-трамплина может являться `pop x ; pop y ; call y`. Естественным образом Sadeghi и др. [19] демонстрируют полноту по Тьюрингу набора таких гаджетов из библиотеки `libc`.

9. Sigreturn-ориентированное программирование

Bosman и др. [42] предложили способ эксплуатации механизма обработки сигналов в операционных системах семейства Unix. Во время доставки сигнала процессу ядро сохраняет на стеке (в пользовательском пространстве) контекст процесса (регистры, указатель стека, флаги процессора и т.д.) в сигнальном фрейме. А в качестве адреса возврата из обработчика сигнала ядро размещает указатель на код, выполняющий системный вызов `sigreturn`, который восстановит контекст процесса из сигнального фрейма. Атакующий может сформировать сигнальный фрейм и вызвать `sigreturn` для изменения контекста процесса. Таким образом, используя лишь один гаджет, выполняющий системный вызов `sigreturn`, можно записать в регистры произвольные значения. Данный подход называется sigreturn-ориентированное программирование (SROP) и позволяет выполнять полные по Тьюрингу вычисления. Авторы предлагают два вида атаки:

1. **Системный вызов `execve`.** Атакующий размещает на стеке строковые аргументы `execve` и формирует сигнальный фрейм с указателями на эти строки. Таким образом, `sigreturn` гаджет проинициализирует регистры, через которые передаются аргументы системного вызова, указателями на строки. А `syscall` гаджет, который уже содержится в коде `sigreturn` гаджета, выполнит системный вызов `execve`.
2. **Быстрые системные вызовы `vsyscall`.** В операционных системах с версией ядра Linux до 3.3 используется механизм быстрых системных вызовов

`vsyscall`. В коде реализации `vsyscall` находится набор полезных гаджетов по фиксированным адресам. В частности, там находится `syscall` гаджет. Авторы утверждают, что гаджеты остаются на тех же адресах после обновлений безопасности ядра и на разных дистрибутивах. Более того, на той же странице памяти, что и код `vsyscall`, хранится текущее время, младшие биты которого при должном терпении можно использовать в качестве гаджета.

10. Контроль целостности потока управления программы

Одним из способов противодействия атакам повторного использования кода является подход, обеспечивающий контроль целостности потока управления (от англ. — CFI, control flow integrity). В литературе существует множество публикаций на данную тему, предлагающих тот или иной способ реализации данного метода [75]. Некоторые из предложенных реализаций даже включены в состав компиляторов как тестовые расширения, но по причинам производительности не используются по умолчанию. Общая идея данных методов заключается в том, что поток управления во время атаки повторного использования кода обычно значительно отличается от потока управления, наблюдаемого во время нормальной работы программы. Подобные отклонения могут быть обнаружены с помощью построения некоторой модели поведения потока управления программы и проверки соответствия ей при фактических передачах управления во время исполнения. Теоретически обеспечение целостности потока управления позволяет предотвратить эксплуатацию приложения методами повторного использования кода, описанными в предыдущих главах.

11. Использование гаджетов, следующих сразу за инструкциями вызова

В обычных программах после возврата из функции управление в большинстве случаев передается на инструкцию, следующую сразу за инструкцией вызова этой функции. Возвратно-ориентированное программирование в общем случае нарушает это условие, передавая после инструкции возврата управление в произвольные места программы. Некоторые реализации контроля целостности потока управления проверяют соблюдение условия возврата на инструкцию, следующую сразу за инструкцией вызова функции, для противодействия эксплуатации ROP цепочками. Однако Carlini и др. [76] заметили, что в таком случае можно использовать только гаджеты, начинающиеся сразу после инструкции вызова (англ. CPROP — Call-Preceded ROP). Такие гаджеты получаются больше в размере и с более сложными побочными эффектами. Однако авторы показали, что в реальных приложениях они встречаются в достаточном количестве, чтобы при аккуратном учете их побочных эффектов создавать работоспособные ROP цепочки. Поток управления при использовании CPROP гаджетов изображен на рисунке 5a сплошной линией, пунктирной линией изображен оригинальный поток управления.

Табл. 2. Набор POSIX-совместимых виджетов
Table 2. A set of POSIX-compliant widgets

Категория	Виджеты
Ветвление	`lfind()` + `longjmp()`, `lsearch()` + `longjmp()`
Арифметика/Логика	`wordexp()`, `sigandset()`, `sigorset()`
Доступ к памяти	`memcpy()`, `strcpy()`, `sprintf()`, `sscanf()`, др.
Системные вызовы	`open()`, `close()`, `read()`, `write()`, др.

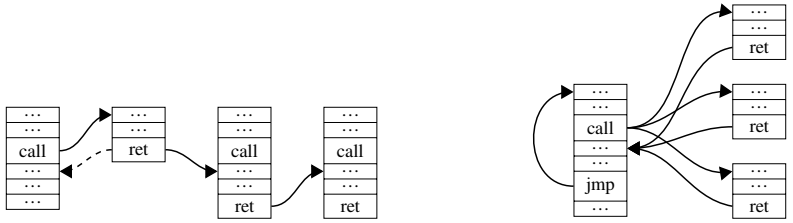
12. Повторное использование функций целиком

Для обхода методов контроля целостности потока управления программы были предложены специфические атаки повторного использования кода. Например, в простейших случаях достаточно использовать в качестве гаджетов целые функции. Tran и др. [77] задаются вопросом, насколько на самом деле выразительны атаки возврата в библиотеку. Они показывают, что на самом деле атаки возврата в библиотеку обладают полнотой по Тьюрингу. Для доказательства этого они строят из POSIX совместимых функций стандартной библиотеки Си полный по Тьюрингу набор *виджетов*. *Виджетом* называется функция с полезными побочными эффектами, это аналог гаджета. Для реализации ветвлений используются `longjmp()` виджеты, которые изменяют указатель стека. На основе набора виджетов строятся два примера эксплойта, которые показывают практическую применимость предложенного подхода. Более того, поскольку данный набор виджетов построен из POSIX-совместимых функций, то обеспечивается переносимость цепочек из виджетов между POSIX-совместимыми операционными системами. Пример POSIX-совместимых виджетов приведен в таблице 2.

Цепочки виджетов сложнее составлять руками, чем ROP цепочки, из-за сложных зависимостей по данными и управлению. Кроме того, цепочки виджетов требуют большего размера стека из-за размера самой цепочки.

Стоит заметить, что построение эксплойтов только из виджетов возможно только на архитектуре x86 с соглашением вызовов, при котором аргументы функций передаются только через стек. В противном случае для загрузки аргументов необходимо прибегать к использованию ROP гаджетов.

Lan и др. [78] предлагают метод цикло-ориентированного программирования (loop-oriented programming, LOP), использующий в качестве гаджетов целые функции. Данный метод разработан с целью обхода метода крупнозернистой проверки целостности потока управления и теневого стека [79]. Для обхода таких средств защиты необходимо передавать управление в начало функции, а возвращать управление — в вызывающую функцию в точку сразу после места вызова. Метод отдаленно напоминает переходо-ориентированное программирование (рис. 4b). Авторы берут в качестве гаджетов целые функции (функциональные гаджеты), а управление между ними передается с помощью гаджета-диспетчера. В качестве



(a) CPROP (b) LOP

Рис. 5. Поток управления для CPROP и LOP. Каждый блок представляет собой целую функцию.
Fig. 5. The control flow of CPROP and LOP. All blocks represent functions.

гаджета-диспетчера выступает функция с циклом, внутри которого содержится косвенный вызов. Адреса функциональных гаджетов хранятся в таблице диспетчеризации. Гаджет-диспетчер на каждой итерации цикла монотонно изменяет виртуальный счетчик команд и вызывает очередной гаджет по адресу из таблицы диспетчеризации, на который указывает счетчик. После возврата из функционального гаджета управление вернется на следующую итерацию гаджета-диспетчера. На рисунке 5b показано, как передается управление между функциональными гаждетами с помощью гаджета-диспетчера.

В объектно-ориентированных языках часто могут встречаться примитивы подобные функции-диспетчеру, например, обходы коллекций однотипных объектов с вызовом для каждого из них определенного виртуального метода. Для таких случаев в работах COOP [80], LOOP [81] предлагается метод эксплуатации, который подменяет таблицу виртуальных вызовов. Правильно созданные виртуальные таблицы объектов из коллекции позволяют организовать цепочку вызовов методов объектов. В таком случае по аналогии с предыдущими подходами текущего раздела в качестве гаджетов выступают целые функции. Данные могут передаваться между разными гаджетами либо через общие поля объектов, либо через неинициализированные переменные методов. В процедурных языках также могут встречаться подобные обходы коллекций структур, содержащих указатели на функции-обработчики. На примере приложения, написанного на Си, в работе FOP [82] показывается пример конструирования эксплойта из функций программы.

13. Эксплуатация потоков данных

Chen и др. [83] в первый раз показали, что возможно эксплуатировать поток данных приложения и выполнять в его контексте полезную нагрузку без нарушения целостности потока управления. Авторы привели несколько примеров, которые были найдены и созданы ими вручную. Позднее Hu и др. [84, 85] дали название таким типам атак (англ. DOP — Data-Oriented Programming) и показали, что DOP атаки

могут быть полными по Тьюрингу, т.е. выполнять произвольный код без нарушения целостности потока управления программы. Тем самым такие атаки не обнаруживаются методами контроля целостности потока управления.

При построении DOP цепочек используются DOP гаджеты, которые могут быть произвольными фрагментами кода, и специальный гаджет-диспетчер, который необходим для передачи управления между DOP гаджетами. Инструкциями виртуальной машины, в которой исполняется DOP цепочка, являются некоторые последовательности инструкций в исходной программе. Значения переменных DOP цепочки хранятся в памяти, поскольку регистры активно используются в программе и, как правило, портятся между выполнениями двух последовательных DOP гаджетов. Примером гаджета-диспетчера может служить цикл с некоторым механизмом выбора DOP гаджета, который позволяет текущему гаджету передавать управление последующему гаджету. Hu и др. [84, 85] создавали DOP цепочки в ручном режиме с элементами автоматизации некоторых шагов. Для построения DOP цепочки вначале находятся все DOP гаджеты и гаждет-диспетчер. После их обнаружения необходимо найти входные данные для целевой программы, которые приведут путь выполнения к месту, содержащему найденные гаджеты. Кроме того, для каждого гаджета описывается, какой регион памяти он менял (глобальные переменные, параметры функции, локальные переменные). Проще всего использовать гаджеты, которые меняют состояние глобальной памяти. Конструирование атаки при наличии в программе ошибки работы с памятью делится на следующие этапы: поиск гаджетов, подбор подходящих гаджетов, сшивание гаджетов. Для сшивания гаджетов Hu и др. [84] строят 2D граф потока данных, который отображает потоки данных в двух измерениях: адресах памяти и времени исполнения. Далее они пытаются открывать новые ребра на этом графе. Авторы показывают несколько примеров сконструированных атак на реальные приложения, которые обходят защиты DEP, ASLR и CFI. Кроме того, они показывают, что в некоторых реальных приложениях содержится достаточное количество гаджетов для создания DOP цепочек, в том числе полных по Тьюрингу.

В работах [86, 87] происходит дальнейшие развитие DOP и методов автоматизированной генерации DOP цепочек. Несмотря на схожесть основной идеи данного типа атак, методы для автоматизированной генерации таких цепочек существенно отличаются, и их детальное обсуждение выходит за рамки данной статьи.

14. Общая схема генерации эксплойтов повторного использования кода

Схематично процесс генерации эксплойтов повторного использования кода делится на четыре этапа:

1. Поиск гаджетов в нерандомизованных исполняемых областях образа памяти процесса (разд. 16).
2. Определение семантики гаджетов (в некоторых методах данный этап может

быть пропущен). На этом этапе определяется полезная нагрузка, которую выполняет каждый гаджет (разд. 17).

3. Комбинация гаджетов и их параметров для получения цепочки гаджетов, выполняющей заданную последовательность действий (разд. 18).
4. Автоматизированная генерация эксплойта [52, 88—92] — входных данных, приводящих к эксплуатации программы путем размещения и выполнения ROP цепочки. На этом этапе в результате символьной интерпретации [93—95] машинных инструкций на трассе выполнения программы от точки получения входных данных до точки проявления уязвимости происходит построение предиката пути. Предикат пути объединяется с предикатом безопасности, описывающим размещение ROP цепочки и передачу на нее управления. Решением полученной системы уравнений будет эксплойт. Предикат пути обеспечит выполнение программы по тому же пути до уязвимости. А предикат безопасности — перехват потока управления.

15. Каталог гаджетов

Прежде чем рассказать о конкретных методах поиска и определения семантики гаджетов, стоит ввести определение понятия *каталог гаджетов*. Определим *каталог гаджетов* как список записей со следующим содержанием:

1. **Семантическое описание** последовательности инструкций машинного кода. Каждое описание соответствует, как правило, некоторой базовой вычислительной операции или операции работы с памятью (сложение, вычитание, запись в память, чтение из памяти, инициализация регистра непосредственным значением, передача управления и т.д.).
2. **Виртуальный адрес** гаджета, обнаруженного в адресном пространстве приложения. Является некоторой аналогией кода операции для архитектуры набора команд, которая задана каталогом гаджетов.
3. **Машинные инструкции** гаджета — конкретная последовательность инструкций, реализующих заданное семантическое описание. Может задаваться вручную при составлении каталога или заполняться при автоматическом анализе двоичного образа приложения.
4. **Параметры гаджета** — параметры семантического описания, а именно: конкретные регистры, константы и т.д.
5. **Побочные эффекты** выполнения гаджета с соответствующей семантикой. Побочным эффектом является любое изменение памяти и регистров, не описываемое семантикой гаджета. Побочные эффекты могут задаваться при построении каталога гаджетов вручную или автоматически вычисляться в процессе классификации гаджетов.

Для пояснения данного определения приведем пример. В таблице 3 представлен каталог гаджетов, состоящий из нескольких семантических описаний. Первое семантическое описание соответствует операции сложения значений двух регистров $r1 \mathrel{+}= r2$. Второе семантическое описание соответствует инструкции загрузки значения со стека в регистр. Последнее семантическое описание определяет гаджет загрузки трех регистров со стека. После проведения поиска гаджетов каталог приобретает вид, показанный в таблице 4. Первые два гаджета, найденные по адресам 0xdeadbeef и 0xcafecafe, обладают побочными эффектами относительно основного семантического описания, потому что они изменяют значения регистров `edx`, `ecx` соответственно.

Табл. 3. Незаполненный каталог гаджетов
Table 3. Incomplete gadget catalogue

Семантическое описание	Виртуальный адрес	Машинные инструкции	Параметры гаджета	Побочные эффекты
r1 += r2				
r = M[ESP + Offset]				
r1 = M[ESP + Off1] r2 = M[ESP + Off2] r3 = M[ESP + Off3]				

15.1 Полный по Тьюрингу каталог гаджетов

Авторы многих работ [5, 7, 18, 19, 77, 96] составляют каталог гаджетов таким образом, чтобы набор семантических описаний являлся полным по Тьюрингу. Такой каталог гаджетов задает некоторую новую вычислительную машину, способную выполнять произвольные вычисления.

В процессе поиска и определения семантики гаджетов происходит заполнение каталога адресами найденных гаджетов. После этого возможны две ситуации:

1. Для каждого семантического описания удалось найти конкретный гаджет.
2. Для некоторых семантических описаний отсутствуют конкретные гаджеты.

В первом случае получается, что найденный набор адресов реализует описываемую каталогом вычислительную машину на конкретном исполняемом файле. Это означает, что с использованием найденных гаджетов возможно произвести произвольные вычисления. Более того, можно использовать этот каталог в качестве описания набора команд целевой архитектуры для компилятора языка Си (llvm [7]).

Во втором случае, когда отсутствуют гаджеты для некоторых семантических описаний полного по Тьюрингу каталога гаджетов, произвольные вычисления уже не выполнить. Поэтому возникает вопрос: чем ограничена вычислительная способность обнаруженного набора гаджетов? Иначе говоря, возможно ли с найденным набором гаджетов выполнить заданную программу? Для ответа на поставленный вопрос нужно анализировать каждую программу, описывающую эксплойт, отдельно. Ос-

Табл. 4. Заполненный каталог гаджетов
Table 4. Complete gadget catalogue

Семантическое описание	Виртуальный адрес	Машинные инструкции	Параметры гаджета	Побочные эффекты
r1 += r2	0xdeadbeef	add eax, ebx pop edx ret	r1 = eax r2 = ebx	edx ✗
	0xcafecafe	add eax, ebx pop ecx ret	r1 = eax r2 = ebx	ecx ✗
	0xcafebabe	add edx, ecx ret	r1 = edx r2 = ecx	—
r = M[ESP + Offset]	0x12345678	pop eax ret	r = eax Offset = 0	—
r1 = M[ESP + Off1] r2 = M[ESP + Off2] r3 = M[ESP + Off3]	0x10203040	pop eax pop ebx pop ecx ret	r1 = eax, Off1 = 0 r2 = ebx, Off2 = 4 r3 = ecx, Off3 = 8	—

новываясь на операциях, которые она совершает, и содержании каталога гаджетов, в некоторых случаях можно до генерации эксплойта сделать вывод, что генерация невозможна. Например, в исходной программе эксплойта присутствуют условные переходы, а гаджетов, реализующих ветвление, в каталоге гаджетов не представлено. Аналогично с записью в память. В остальных случаях нужно пытаться построить эксплойт из имеющихся гаджетов. Если он построен, то ответ на вопрос положителен и подтверждается сконструированным эксплойтом. В противном случае, задача генерации может свестись к перебору всех возможных комбинаций гаджетов, что может быть затратно по времени в случае большого размера каталога гаджетов. Предположительно для реальных исполняемых файлов данная ситуация очень долгого перебора в случае невозможности сгенерировать эксплойт является редкой.

Для составления полных по Тьюрингу ROP цепочек необходимо уметь условно изменять указатель стека, который выступает в роли счетчика команд. Roemer и др. [7] предлагают следующий способ реализации условного ветвления для архитектуры x86:

1. Выполнить некоторую операцию, которая обновит интересующий флаг.
2. Скопировать интересующий флаг из регистра флагов в регистр общего назначения.
3. Использовать этот флаг для условного изменения указателя стека на желаемое смещение (например, путем умножения смещения на значение флага 0 или 1).

16. Поиск гаджетов

Независимо от способа построения эксплойта необходимо в первую очередь найти в двоичном образе приложения все доступные гаджеты. К задаче поиска гаджетов существует два принципиальных подхода. Первый из них предлагает осуществлять поиск гаджетов по списку шаблонов. Шаблоны, как правило, задаются регулярными выражениями над бинарными кодами команд гаджетов. Изначально каталог гаджетов содержит семантические описания гаджетов. Для каждого семантического описания производится поиск гаджетов по некоторому шаблону. В результате в каталог гаджетов для семантических описаний будут добавлены конкретные гаджеты: виртуальные адреса, машинные инструкции и параметры гаджетов. Побочные эффекты (например, испорченные регистры [29, 33]) могут быть получены после анализа машинных инструкций найденных гаджетов.

Второй подход заключается в автоматическом поиске всевозможных последовательностей инструкций, заканчивающихся инструкцией передачи управления. Классическим алгоритмом, реализующим поиск всех гаджетов, является алгоритм Галилео [5]. Алгоритм сначала ищет инструкции передачи управления в исполняемых секциях программы. Для каждой найденной инструкции пробует дизассемблировать несколько байтов, предшествующих инструкции. Все корректно дизассемблированные последовательности инструкций добавляются в каталог гаджетов. Таким образом, каталог будет содержать виртуальные адреса и машинные инструкции гаджетов. Данный алгоритм используется во многих инструментах поиска гаджетов с открытым исходным кодом [27—33, 97—109].

17. Определение семантики гаджетов

Не все найденные гаджеты пригодны для составления ROP цепочек. Для использования гаджета при составлении ROP цепочки необходимо понимать, какую полезную нагрузку выполняет этот гаджет. Определение семантики гаджета может производится вручную [5]. При шаблонном поиске гаджетов семантика содержится в описании шаблона [7, 9, 13, 17—20, 74, 96].

17.1 Типы гаджетов

Schwartz и др. [6] предложили определять функциональность гаджета его принадлежностью к некоторым параметризованным типам, которые задают новую архитектуру набора команд (ISA). Параметрами типов выступают регистры, константы и бинарные операции. Чтобы гаджет можно было использовать при составлении пригодных для эксплуатации ROP цепочек, в работе требуют выполнения следующих свойств гаджета:

- **Функциональность.** У каждого гаджета есть тип, который определяет его функциональность. Тип гаджета описывается семантически с помощью постусловия — булева предиката $\mathcal{B}$, который должен быть всегда истинным по-

сле выполнения гаджета. Следует отметить, что один гаджет может принадлежать сразу нескольким типам. Например, гаджет `push eax ; pop ebx ; pop ecx ; ret` одновременно перемещает `eax` в `ebx` и загружает значение со стека в `ecx`, что соответствует типам `MoveRegG: ebx ← eax` и `LoadConstG: ecx ← [esp + 0]`.

- **Сохранение управления.** Каждый гаджет должен быть способен передать управление другому гаджету.
- **Известные побочные эффекты.** У гаджета не должно быть неизвестных побочных эффектов. Побочные эффекты выполнения гаджета не должны приводить к неконтролируемому поведению программы. Например, запись значения по произвольному адресу памяти может привести к аварийному завершению программы.
- **Константное смещение стека.** Большинство типов гаджетов требуют, чтобы указатель стека увеличивался на постоянное значение после каждого выполнения.

Для того чтобы определить, удовлетворяет ли последовательность инструкций гаджета $\mathcal{I}$ постусловию $\mathcal{B}$, Schwartz и др. [6] используют известную технику из формальной верификации — вычисление слабейшего предусловия [110]. Проще говоря, слабейшее предусловие $wp(\mathcal{I}, \mathcal{B})$ для последовательности инструкций $\mathcal{I}$ и постусловия $\mathcal{B}$ — это булево предусловие, которое описывает, когда $\mathcal{I}$ завершается в состоянии, удовлетворяющем $\mathcal{B}$.

Слабейшие предусловие используется, чтобы убедиться, что определение семантики гаджета всегда выполняется после выполнения последовательности инструкций $\mathcal{I}$. Для этого достаточно проверить:

$$wp(\mathcal{I}, \mathcal{B}) \equiv true \tag{1}$$

Если формула верна, то $\mathcal{B}$ всегда истинно после выполнения $\mathcal{I}$, а значит, $\mathcal{I}$ — гаджет с семантическим типом $\mathcal{B}$.

Однако формальная верификация семантики гаджетов на практике показала себя очень медленной. Для ускорения процесса авторы предложили комбинированный подход. Инструкции гаджета предварительно несколько раз выполняются с использованием случайных входных данных, а затем проверяется истинность $\mathcal{B}$. Если $\mathcal{B}$ окажется ложным хотя бы для одного выполнения, то последовательность инструкций не может быть гаджетом этого типа. Таким образом, более сложное вычисление слабейшего предусловия производится, только если $\mathcal{B}$ истинно для каждого выполнения.

Комбинированный подход можно условно разделить на два этапа: классификацию гаджетов и верификацию гаджетов. На этапе классификации делаются гипотезы о принадлежности гаджетов к некоторым типам и о значениях параметров этих типов. Гипотезы по сути задаются булевыми постусловиями. А на этапе верификации для каждого постусловия формально доказывается его истинность или ложность,

и гипотеза принимается или отвергается соответственно.

17.1.1 Классификация гаджетов

В настоящее время существует множество процессорных архитектур с различными инструкциями. Для того, чтобы абстрагироваться от специфики конкретной архитектуры для написания универсальных алгоритмов, традиционно используется промежуточное представление машинных инструкций (VEX [111], REIL [112], Pivot [113] и др.). В этом случае алгоритмы анализа бинарного кода работают с более простым промежуточным представлением, а не с архитектурой целевого процессора.

В работах [59, 114] классификация гаджета производится на основе интерпретации промежуточного представления инструкций гаджета. Во время интерпретации отслеживаются обращения к регистрам и памяти. Если происходит первое чтение регистра или области памяти, считанное значение генерируется случайным образом. В результате интерпретации будут получены начальные и конечные значения регистров и памяти. На основе этой информации делается вывод о возможной принадлежности гаджета тому или иному типу. Например, для принадлежности типу `MoveRegG` [6] должна существовать такая пара регистров, что начальное значение первого регистра равно конечному значению второго. В результате анализа составляется список всех удовлетворяющих гаджету типов и их параметров (список кандидатов). Затем производится еще несколько запусков процесса интерпретации с отличными входными данными, в результате которых из списка кандидатов удаляются ошибочно определенные типы.

Более того, в результате классификации гаджета могут быть получены [59]:

- Список «испорченных» регистров, значения которых изменились в результате выполнения гаджета.
- Информация о фрейме гаджета (разд. 6.1): размер фрейма и смещение ячейки с адресом следующего гаджета относительно начала фрейма.

Следует отметить, что число неверно классифицированных гаджетов можно уменьшить, если добавить запуски процесса интерпретации с граничными входными данными 0 и -1. Доля неверно классифицированных гаджетов в таком случае незначительна и составляет 0.7 % [115].

17.1.2 Верификация гаджетов

Классификация гаджета предоставляет набор постусловий, описывающих возможную семантику гаджета. Верификация гаджета позволяет формально доказать истинность этих постусловий для произвольных входных данных. Верификация гаджета может производиться как на основе построения слабейшего предусловия [6, 23], как было описано выше, так и с использованием символьной интерпретации

инструкций гаджета [114—116].

Рассмотрим подробнее способ классификации гаджета с использованием символьной интерпретации [93—95]. Во время символьной интерпретации моделируется семантика гаджета с использованием SMT [117] выражений. Изначально всем регистрам присваиваются свободные символьные переменные. Символьная память в начале представляет из себя пустой байтовый массив M битовых векторов:

$$M = (Array\ (\_\ BitVec\ \langle addrSize \rangle)\ (\_\ BitVec\ 8)),$$

где $\langle addrSize \rangle$ — размерность адресного слова архитектуры. Символьное состояние содержит отображение регистров в символьные переменные и текущее состояние символьной памяти. Символьная интерпретация инструкций гаджета порождает SMT формулы над переменными и константами, а также обновляет символьное состояние регистров и памяти в соответствии с операционной семантикой инструкции. Работа с символьной памятью реализуется через операции $select$ и $store$ над $Array$. Функция $(select\ M\ i)$ возвращает i-ый элемент массива M и моделирует чтение байта по адресу i. Функция $(store\ M\ i\ b)$ возвращает массив, полученный из массива M сохранением элемента b по индексу i, что моделирует запись байта b по адресу i.

Heitman и др. [114] сначала транслируют инструкции гаджета в промежуточное представление REIL [112]. А после уже производится символьная интерпретация REIL инструкций.

Постусловие для верификации семантики гаджета представляет из себя булевый предикат над начальными и конечными значениями регистров и памяти. В предикат подставляются регистры и память из соответствующих символьных состояний. Общезначимость формулы постусловия проверяется через невыполнимость ее отрицания с использованием SMT-решателя.

В таблице 5 приводится пример верификации гаджета `ArithmeticLoadG: ebx ← ebx + [eax]`. Изначально регистрам сопоставляются свободные символьные переменные ϕ_i, а память представляется массивом M. Множество формул пусто. Новые формулы добавляются в соответствии с операционной семантикой интерпретируемой инструкции. Для множества формул поддерживается SSA форма — при добавлении формулы создается новая символьная переменная, которой присваивается эта формула. На первом шаге создается новая символьная переменная ϕ_6, которая равна загруженному из памяти значению второго операнда инструкции `[eax]`. В символьном состоянии регистру ecx ставится в соответствие символьная переменная ϕ_6. На втором шаге результат сложения присваивается переменной $\phi_7 = \phi_2 + \phi_6$, которая, в свою очередь, ставится в соответствие результирующему операнду инструкции — регистру ebx в символьном состоянии. На конечном шаге символьное состояние обновляется согласно операционной семантике инструкции возврата, т.е. указатель инструкций загружается со стека, а указатель стека увеличивается на 4. В постусловие, описывающее тип гаджета, подставляются символьные переменные из начального и конечного символьных состояний. При помощи

Табл. 5. Пример верификации гаджета `ArithmeticLoadG: ebx ← ebx + [eax]`
Table 5. Verification of gadget `ArithmeticLoadG: ebx ← ebx + [eax]`

Шаг	**Символьное состояние**	**Инструкция**	**Множество формул**
initial	M, $eax = \phi_1$, $ebx = \phi_2$, $ecx = \phi_3$, $esp = \phi_4$, $eip = \phi_5$	—	$S_0 = \emptyset$
1	$ecx = \phi_6$	`mov ecx, [eax]`	$S_1 = S_0 \cup \{\phi_6 = (concat\ (select\ M\ \phi_1)\ (select\ M\ \phi_1 + 1)\ (select\ M\ \phi_1 + 2)\ (select\ M\ \phi_1 + 3))\}$
2	$ebx = \phi_7$	`add ebx, ecx`	$S_2 = S_1 \cup \{\phi_7 = \phi_2 + \phi_6\}$
final	$eip = \phi_8$, $esp = \phi_9$	`ret`	$S_3 = S_2 \cup \{\phi_8 = (concat\ (select\ M\ \phi_4),\ (select\ M\ \phi_4 + 1),\ (select\ M\ \phi_4 + 2),\ (select\ M\ \phi_4 + 3)),\ \phi_9 = \phi_4 + 4\}$
	Определение семантики		**Верификация**
verify	$final(ebx) = initial(ebx) + initial(M[eax])$		$\phi_7 \neq \phi_2 + (concat\ (select\ M\ \phi_1)\ (select\ M\ \phi_1 + 1)\ (select\ M\ \phi_1 + 2)\ (select\ M\ \phi_1 + 3))$ is **UNSAT**

SMT-решателя проверяется выполнимость отрицания формулы. Отрицание формулы невыполнимо, значит, гаджет удовлетворяет заявленному типу с параметрами.

На листинге 1 приводится пример гаджета, который может быть неверно классифицирован, а верификация позволит устранить эту ошибку. Во время классификации гаджет был отнесен к типу `MoveRegG: EAX ← ECX`. Для отличного от нуля начального значения регистра `eax` гаджет действительно скопирует значение регистра `ecx` в `eax`. Однако, если начальное значение `eax` будет нулевым, то и его конечное значение будет нулевым, что не является копией значения регистра `ecx`, отличного от нуля.

17.1.3 Заполнение каталога гаджетов

Процесс заполнения каталога гаджетов (разд. 15) происходит следующим образом. Каталог гаджетов изначально содержит семантические описания типов гаджетов. Каждый найденный алгоритмом Галилео гаджет классифицируют. В результате классификации будут получены семантические описания (типы гаджетов), которым соответствует гаджет. Записи с виртуальным адресом и машинными инструкциями гаджета добавляются к соответствующим семантическим описаниям. Также в этих записях заполняются параметры гаджета (значения параметров типа) и побочные

```
neg eax ; sbb eax, eax ; and eax, ecx ; pop ebp ; ret
MoveRegG: EAX ← ECX
```

Листинг 1. Пример неверно классифицированного гаджета, который будет отклонен после верификации
Listing 1. Incorrectly classified gadget that will be rejected after verification

эффекты («испорченные» регистры). Далее все гаджеты из каталога верифицируют. В результате верификации из каталога будут удалены неверно классифицированные гаджеты.

17.2 Резюме гаджетов

Резюме гаджета [8, 24, 118] представляет собой описание семантики гаджета в виде компактной спецификации. Резюме гаджета содержит предусловия и постусловия над значениями регистров и памяти. В частности, резюме гаджета может содержать:

- регистры, загруженные со стека (`eax = [esp + 4]`),
- регистры, считанные из памяти (`ecx = [edx + 2]`),
- регистры, значение которых было изменено (`ecx = eax + ebx`),
- диапазоны адресов памяти, по которым производились чтение или запись (`[rsp] <-> [rsp + 0x20]`).

Follner и др. [24] предложили следующий метод составления резюме гаджета. Сначала инструкции гаджета поднимаются до уровня промежуточного представления VEX [111]. Потом продвигаются все присваивания, таким образом, чтобы сформировать в результате одно выражение, называемое постусловием, которое описывает все операции, с помощью которых получилось конечное значение в рассматриваемом регистре. Анализ поддерживает модель памяти, которая позволяет корректно моделировать ситуацию передачи значения через стек. Также этот анализ позволяет получить предусловия, которые описывают диапазоны доступа к памяти по регистру со смещением (`[rax] <-> [rax + 0x20]`). Предусловия указывают на то, что регистры из этих диапазонов памяти должны указывать на память, доступную для чтения/записи.

Процесс **заполнения каталога гаджетов** (разд. 15) происходит следующим образом. Каталог гаджетов изначально содержит виртуальные адреса и машинные инструкции гаджетов. Для каждого гаджета из каталога составляется резюме, которое по сути позволяет заполнить семантическое описание и побочные эффекты гаджета.

17.3 Граф зависимостей гаджета

Milanov [25] предложил представлять гаджет в виде ориентированного графа зависимостей (рис. 6). Вершины соответствуют регистрам, памяти и константам. Вся память представляется единственной вершиной. Регистр же может соответствовать нескольким вершинам: каждая модификация регистра порождает новую вершину (reg_0, reg_1, reg_2 и т.д.). Направленные ребра отражают зависимости по данным (присваивание регистров, доступ к памяти и др.). Инструкции гаджета порождают новые ребра на графе. Ребра, соединенные с памятью, также содержат метки с адресом доступа к памяти и пронумерованы в хронологическом порядке.

Каталог гаджетов (разд. 15) заполняется следующим образом. Изначально каталог содержит виртуальные адреса и инструкции гаджетов. Инструкции каждого гаджета транслируются в промежуточное представление REIL [112], для которого после строится граф зависимостей. В результате обхода графа вычисляется семантическое описание гаджета: конечные значения регистров и памяти выражаются через начальные. Выражение для некоторого конечного значения может иметь условие, при котором это выражение истинно. Далее гаджеты классифицируются по типам (разд. 17.1). Автор мотивировал такой метод определения семантики гаджета меньшим временем работы по сравнению с методами, использующими SMT-решатели.

Для примера на рисунке 6 будет получено следующее семантическое описание:

$$
\begin{aligned}
EIP_1 &= MEM[ESP_0] \\
ESP_3 &= ESP_0 + 4 \\
EAX_1 &= \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases} \\
MEM[ESP_0 - 4] &= \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases} \\
MEM[ESI_0] &= 0
\end{aligned}
$$

18. Генерация цепочек гаджетов

В данном разделе описываются различные методы генерации ROP цепочек. Следует отметить, что комбинирование гаджетов в цепочки является переборной задачей, поэтому для уменьшения числа итераций перебора можно предварительно отфильтровывать ненужные гаджеты и сортировать их по качеству [119]. Процесс генерации ROP цепочек отличается от обычной компиляции следующим:

- Чаще всего у ROP цепочки нет возможности сохранять значения регистров в память для их последующего восстановления из-за нехватки соответствующих гаджетов.
- У ROP гаджетов могут быть побочные эффекты. Например, гаджет может «портить» регистры. Значения «испорченных» регистров не сохранятся по-

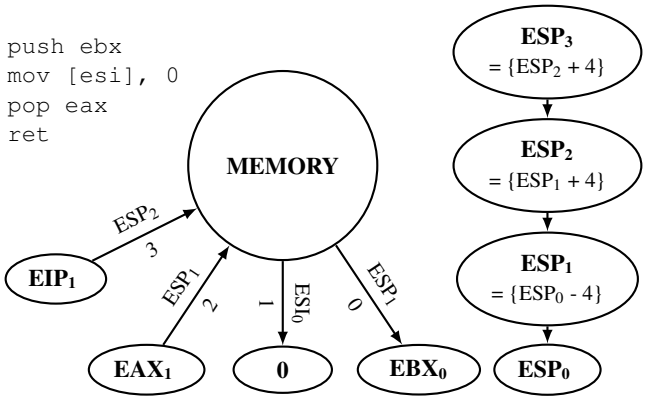


Рис. 6. Граф зависимостей гаджета
Fig. 6. Gadget dependency graph

сле выполнения гаджета. Побочные эффекты необходимо учитывать при составлении расписания гаджетов из цепочки [6].

- Некоторые типы гаджетов (разд. 17.1), которые выступают в качестве инструкций виртуальной машины, могут отсутствовать. В таком случае необходимо заменять недостающие гаджеты последовательностью других [6].

Во время генерации следует учитывать запрещенные символы, которые нельзя использовать в ROP цепочке. Такая потребность возникает, когда, например, переполнение происходит при помощи функции `strcpy`, что не позволяет цепочке содержать нулевые байты. Однако только немногие [22, 31] полноценно решают проблему запрещенных символов. Большинство просто удаляют гаджеты, чьи адреса содержат запрещенные символы, но не следят за значениями параметров гаджетов на стеке.

ROP нагрузка часто может быть разбита на установку значений регистров в заданные значения и выполнение еще одного гаджета [32]. Таким образом, метод генерации ROP цепочек можно базировать на установке регистров, а остальные нагрузки осуществлять путем добавления к полученной цепочке одного гаджета записи в память, вызова функции, системного вызова и др.

18.1 Полная по Тьюрингу компиляция с фиксированным каталогом гаджетов

Рассмотрим построение компилятора на основе фиксированного каталога гаджетов. Buchanan и др. [7, 13] вручную сформировали полный по Тьюрингу каталог гаджетов из машинного кода стандартной библиотеки `libc` ОС Solaris для архитектуры набора команд SPARC. Каждому семантическому описанию сопоставляется

единственная последовательность инструкций из машинного кода библиотеки. Архитектура SPARC допускает только выровненный доступ к инструкциям, поэтому все примеры гаджетов являются легитимными эпилогами функций библиотеки.

Одна из особенностей архитектуры SPARC — использование регистровых окон. Регистровое окно состоит из регистров, предназначенных для входных параметров, возвращаемых значений, временных значений внутри процедуры. При вызове функции происходит сдвиг регистрового окна вперед, а при возврате — в обратную сторону. При большой вложенности стека вызовов в программе происходит нехватка регистровых окон, что приводит к необходимости их сохранения на стеке. В таком случае, при возврате из функции значения регистров восстанавливаются из сохраненных на стеке значений, что приводит к нежелательному изменению значений регистров при передаче управления от одного гаджета к другому. Таким образом, архитектура SPARC и ее соглашение о вызовах накладывает ограничения на способ передачи вычисленных значений между гаджетами — только через память. Каталог гаджетов Buchanan и др. [13] реализует набор гаджетов, использующих модель память-память, которая позволяет использовать регистры только внутри гаджетов, а хранение и передача значений от одного гаджета к другому происходит через память. Каждой переменной в ROP цепочке сопоставляется адрес ячейки памяти, который используется как операнд гаджета.

После полного заполнения каталога гаджетов существует две опции для автоматического создания ROP цепочек. Во-первых, у каталога гаджетов существует программный интерфейс на языке Си. В нем содержатся 13 функций, которые позволяют создавать переменные, присваивать им значения и вызывать функции (или делать системные вызовы). С помощью данного программного интерфейса можно написать программу, которая будет автоматически генерировать ROP цепочку по заполненному каталогу гаджетов. Во-вторых, Buchanan и др. [13] написали транслятор из некоторого псевдо-языка описания эксплойтов (урезанного Си) в последовательность вызовов функций программного интерфейса каталога гаджетов на языке Си. Компилятор реализует большую часть базовой арифметики, логических операций, операций работы с указателями и памятью, и операций условной и безусловной передачи управления. Авторами ряда работ [13, 23] отмечается возможность написания для компиляторной инфраструктуры LLVM расширения, позволяющего генерировать код для виртуальной машины, задаваемой каталогом гаджетов.

Инструмент, представленный Mosier [26, 105], опирается на ROPC-IR, ассемблерный язык описания эксплойтов, который задает полную по Тьюрингу архитектуру набора команд. Он содержит три регистра: `ACC(eax)`, `SP(rbp)`, `PC(rsp)`, и операции для взаимодействия с этими регистрами: базовая арифметика (`ADD`, `SUB`, `NEG`), инструкции ветвления (`CMP`, `JMP`, `JNE`), инструкции взаимодействия регистр-регистр (`MOV`), инструкции взаимодействия регистр-память (`LD`, `STO`), инструкции для работы со стеком (`PUSH`, `POP`, `ALLLOC`, `LEAVE`), инструкции передачи управления (`CALL`, `SYSCALL3`, `LIBCALL3`). В качестве счетчика команд `PC` выступает регистр `rsp`. Кроме того, выделяется отдельный стек для функций ROP цепочки,

указателем на который `SP` выступает `rpb`.

Поддержка второго стека позволяет реализовывать вызовы функций внутри ROP цепочки, которые реализуют полноценные подпрограммы. Кроме того, поддерживается возможность совершать вызовы функций из адресного пространства целевой программы и возможность совершать системные вызовы. В качестве иллюстрации Mosier [105] приводит пример кода на языке ROPC-IR для вычисления чисел Фибоначчи, использующий рекурсивный вызов функции из ROP цепочки, вызов библиотечной функции `printf`, а также системный вызов `exit`.

Каталог гаджетов в данном инструменте представлен описанием языка ROPC-IR. Процесс поиска гаджетов выводит всевозможные гаджеты из целевой программы. Затем необходимо вручную найти и сопоставить каждому семантическому описанию конкретный найденный в целевой программе гаджет и вручную заполнить следующие поля каталога гаджетов: виртуальный адрес, параметры гаджета. По определению языка ROPC-IR гаджеты не имеют побочных эффектов (не принимая во внимание выходные регистры). Теоретически ассемблерный язык ROPC-IR может выступать в качестве целевого языка компилятора Си. Однако практическое применение для построения эксплойтов для реальных программ может быть существенно ограничено наличием необходимых гаджетов в программе и размером генерируемых ROP цепочек. Размер цепочек из-за неоптимальности процесса трансляции языка ROPC-IR значительно больше типичных размеров эксплойтов.

Подход к построению автоматизированного инструмента генерации ROP цепочек, предложенный в работах [7, 13, 26], опирается на фиксированный каталог гаджетов. Он единожды сформирован авторами и не изменяется. Кроме того, семантические описания жестко привязаны к конкретным регистрам, используемым в гаджетах. В случае, если в какой-то версии стандартной библиотеки `libc` отсутствует какой-то из гаджетов, то компиляция ROP цепочки может завершиться неудачно. При этом можно было бы использовать другие гаджеты, имеющие другие операнды, но схожий функционал, и добиться успешной компиляции. Другими словами, данный подход обладает ограниченной практической применимостью, особенно в ситуациях небольшого количества гаджетов в исследуемом коде библиотеки.

18.2 Генерация на основе шаблонов гаджетов

Генерация на основе шаблонов гаджетов заключается в поиске по регулярным выражениям определенной последовательности гаджетов, выполняющей некоторую вредоносную нагрузку: системный вызов `execve` [30, 33], вызов функции `VirtualProtect` с последующим выполнением обычного шелл-кода на стеке (разд. 6.6) [29] и др. Запрещенные символы при таком подходе могут учитываться путем отрицания загруженного со стека значения, многочисленным повторным инкрементом до желаемого значения или же другими арифметическими операциями. Следует отметить, что Ropper [33] поддерживает поиск с использованием SMT-решателей гаджетов, удовлетворяющих семантике, которая задается постусловием

над регистрами, памятью и константами. Однако на момент написания статьи для генерации ROP цепочек инструмент использует только регулярные выражения.

В работе Huang и др. [11] для архитектуры набора команд ARM применяется подход, основанный на использовании специального гаджета, который одновременно устанавливает значения всех регистров со стека. Алгоритм поиска и одновременной проверки гаджета на соответствие заданной семантике производится путем анализа инструкций ассемблерного кода. Генерация цепочки из одного гаджета является тривиальной задачей и требует только правильного расположения значений регистров на стеке.

Другой подход к построению каталога гаджетов и последующей компиляции представлен Hund и др. [20]. На этапе поиска ищутся только гаджеты, состоящие из одной инструкции, не считая саму инструкцию возврата. Скорее всего, так сделано ради упрощения алгоритмов анализа параметров гаджета и побочных эффектов. Такие гаджеты добавляются в каталог гаджетов. На следующем шаге каталог гаджетов дополняется гаджетами, которые можно скомбинировать из имеющихся. Это можно проиллюстрировать следующим примером:

1. `pop eax ; ret` — гаджет загрузки значения со стека в регистр `eax`,
2. `mov ebx, eax ; ret` — гаджет перемещения значения из регистра `eax` в регистр `ebx`.

Эти два гаджета, вызванные последовательно, образуют гаджет загрузки значения со стека в регистр `ebx`. Hund и др. [20] приводят алгоритм поиска всех возможных комбинаций гаджетов, перемещающих значение из одного регистра в другой регистр. Данная задача сводится к поиску пути от одной вершины к другой в специальном графе. В качестве вершин в нем выступают регистры, а в качестве ребер выступают первичные гаджеты, осуществляющие перемещение одного значения регистра в другой. На рисунке 7 представлен пример такого графа.

Данный подход позволяет расширить каталог гаджетов, что особенно полезно для эксплуатируемых программ небольшого размера. Однако использование комбинированных гаджетов требует обязательного учета побочных эффектов на стадии объединения гаджетов в ROP цепочку.

Nguyen Anh Quynh [21] предложил похожую идею объединения нескольких гаджетов в один, выполняющий желаемое поведение. Например, последовательность гаджетов `push 0x1234 ; pop ebp ; ret ; xchg ebp, eax ; ret` может рассматриваться как один гаджет загрузки константы 0x1234 в регистр `eax`.

18.3 Связывание гаджетов в цепочки с использованием семантических запросов

Milanov [25] получает семантическое описание каждого гаджета путем построения его графа зависимостей (разд. 17.3). Связывание гаджетов в цепочки осуществляется последовательными семантическими запросами к каталогу гаджетов. Данный ме-

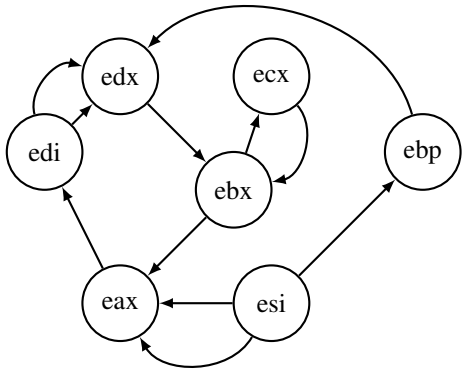


Рис. 7. Граф связи гаджетов перемещения. Комбинация таких гаджетов может быть использована для эмуляции работы недостающих гаджетов.
Fig. 7. MOV connection graph. Chained gadgets can be used to emulate missing gadgets.

тод реализован в виде инструмента с открытым исходным кодом ROPGenerator [31].

Семантический запрос по сути является выражением над константами, конечными/начальными значениями регистров и памяти. Сначала в каталоге гаджетов ищутся гаджеты с семантическим описанием, удовлетворяющим семантическому запросу. Если такие гаджеты отсутствуют, то семантический запрос разбивается на несколько по некоторым стратегиям. Например, первый регистр можно переместить во второй через некоторый промежуточный третий регистр.

Примечательной особенностью инструмента ROPGenerator является поддержка использования при построении ROP цепочки гаджетов, оканчивающихся на инструкции `call Reg` и `jmp Reg`. Для этого перед такими гаджетами добавляется гаджет загрузки регистра `Reg`, который загружает значение адреса гаджета, которому необходимо передать управление после выполнения гаджета с `call` или `jmp`. В случае с `call` может также потребоваться передача управления на специальный гаджет, убирающий со стека адрес возврата, размещаемый `call`.

18.4 Генетический алгоритм

Fraser и др. [12, 120] предлагают иной подход к конструированию ROP цепочки. Авторы предлагают использовать генетические алгоритмы для этого. Fraser и др. представлен инструмент ROPER, доступный на github [28]. Инструмент позволяет генерировать для ARM архитектуры ROP цепочку, устанавливающую значения регистров в заданные значения.

Вначале в исполняемом файле находятся гаджеты. Для каждого из них вычисляется размер фрейма гаджета и смещение адреса следующего гаджета в нем (разд. 6.1). Затем исполняемый целевой файл загружается в адресное пространство виртуальной машины для повторяемого выполнения ROP цепочек-кандидатов. Виртуальная машина предоставляет удобный интерфейс для выполнения инструкций гостевой архитектуры.

В процессе генетических мутаций роль генов выполняют адреса гаджетов и случайные значения, размещаемые на стеке в качестве данных и адреса следующего гаджета. Функцией приспособленности является разность текущего и желаемого вектора значений регистров виртуальной машины. Каждый элемент популяции изменяется методами генетических мутаций. Среди всех кандидатов отбирается набор потенциально лучших, для которых процесс мутации повторяется вновь.

Следует отметить, что сформированные генетическим алгоритмом ROP цепочки сильно отличаются от тех, что создаются людьми. Например, они могут писать значения себе на стек или передавать управление на гаджеты, которые не были изначально обнаружены в процессе поиска. Кроме того, размер цепочки из-за неоптимального выбора гаджетов может быть большим. Описанные недостатки могут быть следствием отсутствия у генетического алгоритма информации о семантике инструкций гаджета. Возможно, если в каком-то виде ее учитывать и использовать более современные методы машинного обучения, то можно развить концепцию данного подхода в практически применимый инструмент.

18.5 Генерация цепочек с использованием SMT-решателей

Follner и др. [24] предложили метод генерации на основе резюме гаджетов (разд. 17.2), который имеет доступный исходный код [27]. Метод позволяет получать последовательность гаджетов, которая запишет в m запрошенных регистров заданные значения. Следует отметить, что метод не вычисляет загружаемые со стека параметры гаджетов, а только предоставляет последовательность адресов гаджетов. Изначально для всех гаджетов составляется резюме. Для каждого запрошенного регистра выбираются n наиболее подходящих гаджетов [119], которые загружают его значение со стека или из памяти, контролируемой атакующим. Далее для всевозможных цепочек гаджетов ($n^m * m!$ комбинаций) вычисляются предусловия и постусловия, но уже для всей цепочки. Если постусловия удовлетворяют ситуации, когда атакующий контролирует значения всех запрошенных регистров, то метод переходит к финальной стадии — разрешению предусловий. К цепочке дополнительно в начало добавляются гаджеты, которые проинициализируют регистры из предусловий, так чтобы они указывали на доступную для чтения и записи память.

Salls [32] развил описанный выше метод. Ниже будет описан метод генерации цепочки, устанавливающей значения регистров в заданные значения. Остальные цепочки, такие как запись в память и вызов функции, могут быть получены добавлением всего лишь одного гаджета к цепочке, инициализирующей регистры. Метод можно разделить на три шага:

1. **Составление резюме гаджетов.** Происходит символьная интерпрета-

Алгоритм 1. Алгоритм поиска кратчайших цепочек инициализации регистров
Algorithm 1. Search algorithm of shortest chain initializing registers

```
regset_to_chain ← empty register set mapping to shortest chains
queue ← empty queue
queue.push(empty chain)
while queue is not empty do
    chain ← queue.pop()
    for all gadget ∈ gadgets do
        new_chain ← chain + gadget
        regset ← controlled_registers(new_chain)
        if regset not in regset_to_chain or new_chain is shorter than
regset_to_chain[regset] then
            regset_to_chain[regset] ← new_chain
            queue.push(new_chain)
        end if
    end for
end while
```

ция [93—95] инструкций каждого гаджета. Резюме гаджетов составляется с использованием статического анализа полученных в результате символьной интерпретации SMT выражений и запросов к SMT-решателю.

2. **Связывание гаджетов в цепочку.** На этом шаге происходит поиск кратчайших цепочек для инициализации произвольных наборов регистров. Предложенный алгоритм 1 был вдохновлен алгоритмом Дейкстры [121] поиска кратчайших путей от одной из вершин графа до всех остальных. Создается пустое отображение из наборов регистров в кратчайшие цепочки, инициализирующие эти регистры. В очередь добавляется пустая цепочка. Алгоритм достает цепочки из очереди. Для каждого гаджета создается новая цепочка, полученная добавлением этого гаджета к взятой из очереди цепочки. Вычисляется набор инициализируемых регистров ($controlled\_registers$) новой цепочкой. Если такого набора нет в отображении, или полученная цепочка короче той, что в отображении, то в отображение для этого набора добавляется новая цепочка. Также эта же цепочка добавляется в очередь. Таким образом, будет получено отображение из наборов регистров в кратчайшие цепочки, которые эти регистры инициализируют.
3. **Размещение ROP цепочки на стеке.** Запускается процесс символьной интерпретации инструкций всей ROP цепочки. Для значений, загружаемых со стека, создаются свободные символьные переменные. В конце процесса интерпретации составляется конъюнкция равенств запрошенных регистров заданным значениям. В результате решения этой конъюнкции SMT-решателем будут получены байты, которые необходимо разместить на

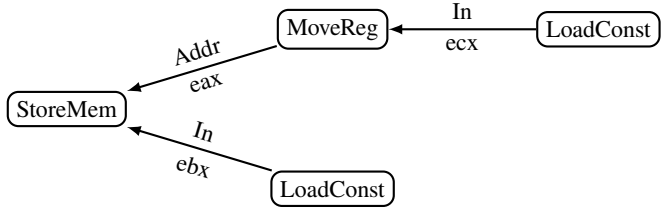


Рис. 8. Дерево гаджетов, которое записывает произвольное значение по произвольному адресу памяти
Fig. 8. Gadget tree that stores arbitrary value at arbitrary memory address

стеке.

Описанный метод, в отличие от предыдущего, позволяет использовать в цепочках гаджеты, которые инициализируют сразу несколько регистров, а также гаджеты, которые выполняют арифметическую операцию над регистрами, загруженными другими гаджетами (правильные значение на стеке при этом вычислит SMT-решатель). Более того, данный метод позволяет выбирать самые короткие цепочки.

18.6 Генерация на основе семантических деревьев

Schwartz и др. [6] предлагают подход к генерации ROP цепочек на основе семантических деревьев. Авторы создали свой язык QooL для написания ROP цепочек, который не обладает полнотой по Тьюрингу, но позволяет выражать применяемые на практике ROP цепочки (вызов библиотечной функции, системный вызов и запись в память). Процесс трансляции программы на языке QooL в ROP цепочку состоит из следующих этапов:

1. Генерация семантических деревьев путем замощения [122] абстрактного синтаксического дерева исходной программы на языке QooL. Семантическое дерево состоит из абстрактных гаджетов (типов гаджетов), которые задают архитектуру набора команд и описаны в разделе 17.1.
2. Присвоение абстрактным гаджетам из семантического дерева реальных гаджетов, найденных в программе. Пример дерева реальных гаджетов приводится на рисунке 8. В вершинах дерева записаны типы гаджетов. На ребрах — имена параметров типов и их значения (конкретные регистры). Дерево гаджетов производит запись произвольного значения по произвольному адресу памяти. Записываемое значение и адрес загружаются со стека в регистры ebx и ecx соответственно. Адрес из регистра ecx перемещается в регистр eax. После чего уже происходит запись значения регистра ebx по адресу eax.
3. Построение расписания по дереву гаджетов и генерация ROP цепочки.

На первом шаге происходит ленивая генерация всех возможных семантических де-

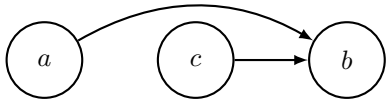


Рис. 9. Построение расписания для дерева гаджетов
Fig. 9. Scheduling gadget tree

ревьев из абстрактных гаджетов. Это необходимо делать поскольку некоторые гаджеты могут отсутствовать в конкретной программе. На втором шаге для каждого семантического дерева применяется присвоение гаджетов. В случае, если не удается присвоить каждому абстрактному гаджету конкретный, то семантическое дерево отбрасывается и берется следующее. В случае успешного присваивания на третий шаг передается дерево реальных гаджетов. Для генерации ROP цепочки его необходимо линеаризовать, т.е. построить расписание. Построение расписания для дерева гаджетов должно учитывать: зависимости между регистрами гаджетов по данным и «испорченные» регистры. Это означает следующее (рис. 9):

1. Расписание должно удовлетворять топологической сортировке дерева.
2. Если выходной регистр гаджета a используется гаджетом b, то этот регистр не должен быть «испорчен» ни одним гаджетом в расписании между a и b.

При генерации семантических деревьев учитывается возможность отсутствия некоторых типов гаджетов и применяются последовательно все имеющиеся правила выражения вершины абстрактного синтаксического дерева через семантические деревья из абстрактных гаджетов. Например, авторы заметили, что успешность генерации ROP цепочки возрастает, если добавить следующее правило выражения вершины сохранения значения в память:

1. `mov [eax], 0 ; ret`
2. `pop ebx ; ret`
3. `add [eax], ebx ; ret`

Ouyang и др. [23] расширили набор инструкций языка QooL до полного по Тьюрингу набора. В целом они повторяют подход Schwartz и др. [6] с построением семантических деревьев, используя при учете побочных эффектов анализ жизни значений.

Следует отметить, что существуют попытки реализации метода Schwartz и др., имеющие открытый исходный код [34, 36, 123].

19. Учет запрещенных символов

Автоматические инструменты генерации ROP цепочек должны учитывать особенности санитизации входных данных для конкретного эксплойта. Например, данные, копируемые через функцию `strcpy`, не могут содержать нулевые байты.

Байты, требующие санитизации, могут содержаться как в адресах гаждетов, так

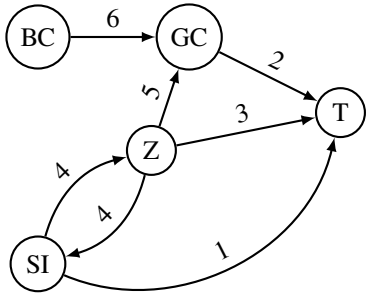


Рис. 10. Конечный автомат, описывающий алгоритм санитизации значений загрузки на регистр
Fig. 10. State machine describing the input sanitizing algorithm

и в данных, загружаемых этими гаджетами на регистры. В простейшем случае санитизация адресов гаджетов производится путем отбрасывания гаджетов, содержащих запрещенные символы в адресе. Так поступают многие инструменты. Однако данный подход неизбежно приходит к уменьшению каталога гаджетов, что приводит к нехватке гаджетов и необходимости их комбинирования для моделирования недостающих гаджетов.

Гораздо сложнее обстоит ситуация, когда запрещенные символы содержатся в данных, предназначенных для загрузки на регистры (значения аргументов функций и необходимые для записи в память значения). Для решения данной проблемы можно использовать всевозможные арифметические операции для получения значений, содержащих запрещенные символы.

Подробное описание способов борьбы с запрещенными символами описано в статье Ding и др. [22]. Стоит отметить, что авторы статьи борются со всеми непечатаемыми символами в цепочках, что может быть избыточно в некоторых случаях. Однако их методы применимы и в более общем случае произвольного заданного множества запрещенных символов. Для каждого найденного гаджета авторы строят семантическое дерево, описывающее функциональность гаджета и содержащее явные зависимости между регистрами и памятью относительно арифметических операций и операций взаимодействия с памятью.

Построенные семантические деревья используются при построении конечного автомата (рис. 10), используемого для поиска инструкций, загружающих значение на регистр. Вершинами в конечном автомате являются следующие состояния, отвечающие разным загружаемым значениям:

- Z — ноль,
- SI — небольшое число,
- GC — число, не содержащее запрещенных символов,

- BC — число, содержащее запрещенные символы,
- T — конечное состояние.

Между этими вершинами проводятся ребра, соответствующие определенным гаджетам, при условии их наличия в каталоге гаджетов. Возможные варианты перехода между состояниями конечного автомата:

1. $SI \longrightarrow T$, из вершины с небольшим числом в конечное состояние ведет ребро, соответствующее гаджету с инструкцией, непосредственно устанавливающей это значение в регистре.
2. $GC \longrightarrow T$, из вершины с числом, не содержащим запрещенных символов, в конечное состояние ведет ребро с гаджетом `pop`.
3. $Z \longrightarrow T$, из вершины с нулем в конечное состояние ведет ребро с инструкцией `xor`.
4. $SI \longleftrightarrow Z$, из вершины с небольшим числом в нуль и обратно ведут ребра с инструкциями `inc, dec`.
5. $Z \longrightarrow GC$, из вершины с нулем в состояние с числом, не содержащим запрещенных символов, ведет ребро с арифметическими инструкциями `and, or, sal, shl, shr, sar`.
6. $BC \longrightarrow GC$, из вершины с числом, содержащим запрещенные символы, в вершину, не содержащую запрещенных символов, ведут ребра, состоящие из комбинации двух арифметических операций, например, $a + b - c$.

Работа алгоритма начинается из состояния, соответствующего значению, которое нужно установить в определенном регистре. Путем обхода состояний данного автомата решается вопрос возможной санитизации данных ROP цепочки. Алгоритм прерывается, если достигнуто конечное состояние, что соответствует успешному нахождению комбинации гаджетов, решающих поставленную задачу, или в случае отсутствия переходов в другие состояния из текущего.

20. Экспериментальное сравнение инструментов

Экспериментальная проверка инструментов с доступным исходным кодом проводилась с помощью тестовой системы rop-benchmark [44]. Данная система позволяет проверять работоспособность ROP цепочек, генерируемых инструментами. Система предоставляет воспроизводимое окружение для проверки факта успешной генерации и работоспособности ROP цепочек, осуществляющих системный вызов `execve("/bin/sh", 0, 0)`. Система тестирования поддерживает платформу Linux x86-64. В качестве тестовых наборов взяты исполняемые файлы и библиотеки минимальных установок нескольких популярных дистрибутивов: CentOS 7, Debian 10, OpenBSD 6.2, OpenBSD 6.4. Дистрибутивы OpenBSD 6.2 и 6.4 взяты по причине того, что авторы этой операционной системы ведут целенаправленную борьбу с ROP гаджетами [124].

Табл. 6. Экспериментальное сравнение инструментов автоматической генерации ROP цепочек
Table 6. The experimental evaluation of automatic ROP chain generating tools

Тестовый набор	Debian 10			CentOS 7			OpenBSD 6.2			OpenBSD 6.4		
Кол-во файлов	689			649			397			410		
Инструмент	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL
ROPgadget [30]	7	0	0	8	0	0	4	0	0	2	0	0
angrop [32]	87	8	4	53	6	1	24	1	5	7	1	5
ROPGenerator [31]	83	4	20	66	5	3	24	3	13	1	0	12
Ropper [33]	53	33	0	31	37	0	14	14	1	1	0	2

Результаты экспериментальной проверки приводятся в таблице 6. Четыре столбца соответствуют четырем наборам тестовых файлов. В первой строке указано общее количество тестовых файлов в каждом из наборов. Ниже находятся строки с инструментами, и напротив каждого инструмента указана следующая информация:

- OK — количество тестовых файлов, для которых созданная ROP цепочка работоспособна, т.е. приводит к открытию оболочки системного интерпретатора.
- F — количество тестовых файлов, для которых созданная ROP цепочка не является работоспособной, т.е. по каким-то причинам не приводит к открытию оболочки системного интерпретатора.
- TL — количество тестовых файлов, на которых время работы инструмента превысило установленный лимит в 300 секунд.

В экспериментальное сравнение попали только находящиеся в открытом доступе инструменты, которые способны в полностью автоматическом режиме генерировать ROP цепочку, осуществляющую системный вызов для архитектуры x86-64 с операционной системой семейства Linux. Из-за операционной системы не попал в рассмотрение инструмент mona.py [29]. Другие могут работать только с архитектурой x86 (32-битной) [37], ARM [28]. Некоторые доступные инструменты не удалось успешно встроить в автоматизированную систему запуска [34].

В набор тестовой системы не вошли тесты с запрещенными символами (например, 0x00 в случае переполнения при копировании с `strcpy`) по причине того, что только ROPGenerator [31] поддерживает их в полном объеме, т.е. проверяет на наличие запрещенных символов не только адреса гаджетов, но и значения параметров гаджетов на стеке.

21. Заключение

В данной статье был проведен подробный обзор атак повторного использования кода и методов автоматизированной генерации эксплойтов для таких атак. Атаки повторного использования кода предполагают использование кусочков кода из адресного пространства программы, называемых *гаджетами*. Гаджеты связываются

в цепочку, выполняющую вредоносную нагрузку. Схематично процесс генерации эксплойтов повторного использования кода делится на четыре этапа: поиск гаджетов в эксплуатируемой программе, определение семантики гаджетов, комбинация гаджетов в цепочки и генерация входных данных, эксплуатирующих уязвимость. Найденные в программе гаджеты добавляются в каталог гаджетов. После этого происходит определение семантики гаджетов: классификация гаджетов по параметризованным семантическим типам, составление резюме гаджетов или построение графов зависимостей гаджетов. Если набор гаджетов в каталоге полон по Тьюрингу, то гаджеты из каталога можно использовать в качестве целевой архитектуры набора команд компилятора. Связывание гаджетов в цепочки может происходить как поиском гаджетов по шаблонам, задаваемых регулярными выражениями, так и с учетом семантики гаджета. Также существуют подходы конструирования ROP цепочек с использованием генетических алгоритмов, а также методы с использованием SMT-решателей. Следует отметить, что рассмотрение методов автоматизированной генерации цепочек, эксплуатирующих потоки данных (DOP), выходит за рамки данной статьи.

Мы предложили набор тестов ROP Benchmark [44] для экспериментального сравнения инструментов генерации ROP цепочек. С помощью него было проведено сравнение инструментов генерации ROP цепочек с открытым исходным кодом для платформы Linux x86-64. В том числе сравнение производилось на дистрибутивах операционной системы OpenBSD, авторы которой целенаправленно ведут борьбу с ROP гаджетами [124].

Важным аспектом при генерации цепочек является учет запрещенных символов. Например, если данные получаются с использованием функции `strcpy`, то они не могут содержать нулевых байтов. Однако немногие авторы учитывают запрещенные символы в методах генерации цепочек.

Существует множество методов атак повторного использования кода (ROP, JOP и др.). Открытым вопросом является, какой набор таких методов является достаточным для осуществления эксплуатации. Например, атакующий мог вручную сформировать JOP цепочку для некоторой уязвимой программы, а продвинутые методы генерации ROP цепочек позволили сгенерировать обычную ROP цепочку для той же программы. Вызывает вопрос о том, возможно ли улучшить методы генерации цепочек и не использовать сложные вариации атак повторного использования кода.

Перспективным направлением является исследование возможности эксплуатации защит, рандомизирующих адресное пространство (ASLR, Fine-grained ASLR и др.), без использования утечек информации о расположении адресного пространства и перебора адресов.

Большинство методов запрещает использование гаджетов, которые, помимо основной нагрузки, разыменовывают произвольный адрес памяти. Учет побочных доступов к памяти [24] у гаджетов потенциально позволил бы расширить каталог гаджетов и улучшить методы генерации цепочек.

Список литературы / References

[1]. The Heartbleed bug. URL: `http://heartbleed.com`.

[2]. D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: software radio attacks and zero-power defenses. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 129–142. IEEE, 2008. DOI: `10.1109/SP.2008.31`.

[3]. CWE - 2019 CWE top 25 most dangerous software errors. URL: `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`.

[4]. A. Peslyak. Getting around non-executable stack (and fix), Aug. 1997. URL: `https://seclists.org/bugtraq/1997/Aug/63`.

[5]. H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, Alexandria, Virginia, USA. ACM, 2007. DOI: `10.1145/1315245.1315313`.

[6]. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, San Francisco, CA. USENIX Association, 2011. URL: `https://www.usenix.org/legacy/event/sec11/tech/full_papers/Schwartz.pdf`.

[7]. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012. DOI: `10.1145/2133375.2133377`.

[8]. T. Kornau. *Return oriented programming for the ARM Architecture*. Master's thesis, Ruhr-University, Bochum, Germany, 2009. URL: `https://bxi.es/Reversing-Exploiting/ROP/Return%20Oriented%20Programming%20for%20ARM.pdf`.

[9]. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, Chicago, Illinois, USA. ACM, 2010. DOI: `10.1145/1866307.1866370`.

[10]. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical report, Technical Report HGI-TR-2010-002, Ruhr-University Bochum, 2010. URL: `https://www.ei.ruhr-uni-bochum.de/media/trust/veroeffentlichungen/2010/07/21/ROP-without-Returns-on-ARM.pdf`.

[11]. Z.-S. Huang and I. G. Harris. Return-oriented vulnerabilities in ARM executables. In *2012 IEEE Conference on Technologies for Homeland Security (HST)*, pages 1–6, Nov. 2012. DOI: 10.1109/THS.2012.6459817.

[12]. O. L. Fraser, N. Zincir-Heywood, M. Heywood, and J. T. Jacobs. Return-oriented programme evolution with ROPER: a proof of concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, New York, NY, USA. ACM, 2017. DOI: 10.1145/3067695.3082508.

[13]. E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, Alexandria, Virginia, USA. ACM, 2008. DOI: 10.1145/1455770.1455776.

[14]. A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, Alexandria, Virginia, USA, 2008. DOI: 10.1145/1455770.1455775.

[15]. F. Lindner. Cisco IOS router exploitation. In *Black Hat*, 2009. URL: https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-PAPER.pdf.

[16]. S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of the 2009 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE'09, Montreal, Canada. USENIX Association, 2009. URL: https://www.usenix.org/legacy/event/evtwote09/tech/full_papers/checkoway.pdf.

[17]. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, Hong Kong, China. ACM, 2011. DOI: 10.1145/1966913.1966919.

[18]. P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 20–29, Hong Kong, China. ACM, 2011. DOI: 10.1145/1966913.1966918.

[19]. A. Sadeghi, S. Niksefat, and M. Rostamipour. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14(2):139–156, May 2018. DOI: 10.1007/s11416-017-0299-1.

[20]. R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 383–398, Montreal, Canada. USENIX Association, 2009. URL: https://www.usenix.org/legacy/events/sec09/tech/full_papers/hund.pdf.

[21]. N. A. Quynh. OptiROP: hunting for ROP gadgets in style, 2013. URL: https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-Slides.pdf.

[22]. W. Ding, X. Xing, P. Chen, Z. Xin, and B. Mao. Automatic construction of printable return-oriented programming payload. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 18–25, Oct. 2014. DOI: 10.1109/MALWARE.2014.6999408.

[23]. Y. Ouyang, Q. Wang, J. Peng, and J. Zeng. An advanced automatic construction method of ROP. *Wuhan University Journal of Natural Sciences*, 20(2):119–128, Apr. 2015. DOI: 10.1007/s11859-015-1069-x.

[24]. A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. PSHAPE: automatically combining gadgets for arbitrary method execution. In *Security and Trust Management*, pages 212–228. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-46598-2_15.

[25]. B. Milanov. ROPGenerator: practical automated ROP-chain generation, 2018. URL: https://youtu.be/rz7Z9fBLVs0.

[26]. N. Mosier and P. Johnson. ROP with a 2nd stack, 2019. URL: http://www.cs.middlebury.edu/~nmosier/portfolio/rsrc/ropc-slides.pdf.

[27]. A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. PSHAPE - practical support for half-automated program exploitation. URL: https://github.com/Alexandre-Bartel/inspector-gadget.

[28]. O. L. Fraser. ROPER: a genetic ROP-chain development tool. URL: https://github.com/oblivia-simplex/roper.

[29]. mona.py, Corelan Consulting BVBA. URL: https://github.com/corelan/mona.

[30]. J. Salwan. ROPgadget tool. URL: https://github.com/JonathanSalwan/ROPgadget.

[31]. B. Milanov. ROPGenerator. URL: https://github.com/Boyan-MILANOV/ropgenerator.

[32]. C. Salls. angrop. URL: https://github.com/salls/angrop.

[33]. S. Schirra. Ropper. URL: https://github.com/sashs/ropper.

[34]. Paul. ROPC. URL: https://github.com/pakt/ropc.

[35]. ropc-llvm, Programa STIC. URL: https://github.com/programa-stic/ropc-llvm.

[36]. J. Stewart. An open source, multi-architecture ROP compiler. URL: https://github.com/jeffball55/rop_compiler.

[37]. SQLab. SQLab ROP payload generation. URL: https://github.com/SQLab/ropchain.

[38]. A. V. Gautham and S. Singh. ROPilicious. URL: https://github.com/ROPilicious/src.

[39]. K. Lu, S. Xiong, and D. Gao. RopSteg: program steganography with return oriented programming. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 265–272. ACM, 2014. DOI: 10.1145/2557547.2557572.

[40]. C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis. Transforming malicious code to ROP gadgets for antivirus evasion. *IET Information Security*, 2019. DOI: 10.1049/iet-ifs.2018.5386.

[41]. D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, and L. Shi. ROPOB: obfuscating binary code via return oriented programming. In *Security and Privacy in Communication Networks*, pages 721–737. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-78813-5_38.

[42]. E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, May 2014. DOI: 10.1109/SP.2014.23.

[43]. P. Borrello, E. Coppa, D. Daniele Cono, and C. Demetrescu. The ROP needle: hiding trigger-based injection vectors via code reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1962–1970, Limassol, Cyprus, 2019. DOI: 10.1145/3297280.3297472.

[44]. A. Nurmukhametov. ROP Benchmark. URL: https://github.com/ispras/rop-benchmark.

[45]. W^X now mandatory in OpenBSD, May 2016. URL: https://undeadly.org/cgi?action=article&sid=20160527203200.

[46]. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. URL: https://support.microsoft.com/kb/875352/EN-US/.

[47]. A. van de Ven. New security enhancements in Red Hat Enterprise Linux v.3, update 3, 2004. URL: https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[48]. A. S. Tanenbaum and H. Bos. *Modern operating systems*. In 4th edition edition. Pearson, 4th edition edition, 2015. Chapter Buffer Overflow Attacks, pages 640–649.

[49]. CWE-123: write-what-where condition. URL: https://cwe.mitre.org/data/definitions/123.html.

[50]. B. Spengler. PaX: the guaranteed end of arbitrary code execution. URL: https://grsecurity.net/PaX-presentation.pdf.

[51]. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12(2), pages 291–301, 2003. URL: https://www.usenix.org/legacy/event/sec03/tech/full_papers/bhatkar/bhatkar.pdf.

[52]. А. Н. Федотов, В. А. Падарян, В. В. Каушан, Ш. Ф. Курмангалеев, А. В. Вишняков и А. Р. Нурмухаметов. Оценка критичности программных дефектов в условиях работы современных защитных механизмов. *Труды института системного программирования РАН*, 28(5):73—92, 2016. DOI: 10.15514/ISPRAS-2016-28(5)-4. / A. N. Fedotov, V. A. Padaryan, V. V. Kaushan, S. F. Kurmangaleev, A. V. Vishnyakov, and A. R. Nurmukhametov. Software defect severity estimation in presence of modern defense mechanisms. *Proceedings of the Institute for System Programming of the RAS*, 28(5):73–92, 2016. DOI: 10.15514/ISPRAS-2016-28(5)-4.

[53]. Procedure Linkage Table (processor-specific). URL: https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-1235.html.

[54]. J. Salwan. An introduction to the return oriented programming and ROP-chain generation, 2014. URL: http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf.

[55]. T. F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2018. DOI: 10.1109/TETC.2017.2785299.

[56]. M. Graziano, D. Balzarotti, and A. Zidouemba. ROPMEMU: a framework for the analysis of complex code-reuse attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 47–58. ACM, 2016. DOI: 10.1145/2897845.2897894.

[57]. E. J. Schwartz, T. Avgerinos, and D. Brumley. Update on q: exploit hardening made easy, 2012. URL: https://edmcman.github.io/papers/usenix11-update.pdf.

[58]. N. R. Weidler, D. Brown, S. A. Mitchell, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes. Return-oriented programming on a resource constrained device. *Sustainable Computing: Informatics and Systems*, 2018. DOI: 10.1016/j.suscom.2018.10.002.

[59]. А. В. Вишняков. Классификация ROP гаджетов. *Труды института системного программирования РАН*, 28(6):27—36, 2016. DOI: 10.15514/ISPRAS-2016-28(6)-2. / A. V. Vishnyakov. Classification of ROP gadgets. *Proceedings of the Institute for System Programming of the RAS*, 28(6):27–36, 2016. DOI: 10.15514/ISPRAS-2016-28(6)-2.

[60]. А. В. Вишняков, А. Р. Нурмухаметов, Ш. Ф. Курмангалеев и С. С. Гайсарян. Метод анализа атак повторного использования кода. *Труды института системного программирования РАН*, 30(5):31—54, 2018. DOI: 10.15514/ISPRAS-2018-30(5)-2. / A. V. Vishnyakov, A. R. Nurmukhametov, S. F. Kurmagaleev, and S. S. Gaisaryan. Method for analysis of code-reuse attacks. *Proceedings of the Institute for System Programming of the RAS*, 30(5):31–54, 2018. DOI: 10.15514/ISPRAS-2018-30(5)-2.

[61]. G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *2009 Annual Computer Security Applications Conference*, pages 60–69, Dec. 2009. DOI: 10.1109/ACSAC.2009.16.

[62]. PE format - Import Address Table. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#import-address-table.

[63]. ld.so(8) - Linux manual page. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html.

[64]. J. Kirsch, B. Bierbaumer, T. Kittel, and C. Eckert. Dynamic loader oriented programming on Linux. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 1–13, Vienna, Austria. ACM, 2017. DOI: 10.1145/3150376.3150381.

[65]. B. C. Ward, R. Skowyra, C. Spensky, J. Martin, and H. Okhravi. The leakage-resilience dilemma. In *Computer Security – ESORICS 2019*, pages 87–106. Springer International Publishing, 2019. DOI: 10.1007/978-3-030-29959-0_5.

[66]. D. Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010. URL: https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[67]. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998. URL: https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.

[68]. mprotect(2) - Linux manual page. URL: http://man7.org/linux/man-pages/man2/mprotect.2.html.

[69]. VirtualProtect function. URL: https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualprotect.

[70]. P. Van Eeckhoutte. Exploit writing tutorial part 10 : chaining DEP with ROP – the Rubik's[TM] cube, 2010. URL: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/.

[71]. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA. IEEE Computer Society, 2014. DOI: 10.1109/SP.2014.22.

[72]. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013. DOI: 10.1109/SP.2013.45.

[73]. A. R. Nurmukhametov, E. A. Zhabotinskiy, S. F. Kurmangaleev, S. S. Gaissaryan, and A. V. Vishnyakov. Fine-grained address space layout randomization on program load. *Programming and Computer Software*, 44(5):363–370, Sept. 2018. DOI: 10.1134/S0361768818050080.

[74]. E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent code reuse: on the effectiveness of ASLR in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 227–242, Apr. 2018. DOI: 10.1109/EuroSP.2018.00024.

[75]. N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, Apr. 2017. DOI: 10.1145/3054924.

[76]. N. Carlini and D. Wagner. ROP is still dangerous: breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, 2014. URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-carlini.pdf.

[77]. M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In R. Sommer, D. Balzarotti, and G. Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-23644-0_7.

[78]. B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 190–197, Aug. 2015. DOI: 10.1109/Trustcom.2015.374.

[79]. L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, 2014. URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-davi.pdf.

[80]. F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015. DOI: 10.1109/SP.2015.51.

[81]. C. Wang, B. Chen, Y. Liu, and H. Wu. Layered object-oriented programming: advanced vtable reuse attacks on binary-level defense. *IEEE Transactions on Information Forensics and Security*, 14(3):693–708, Mar. 2019. DOI: 10.1109/TIFS.2018.2855648.

[82]. Y. Guo, L. Chen, and G. Shi. Function-oriented programming: a new class of code reuse attack in C applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, May 2018. DOI: 10.1109/CNS.2018.8433189.

[83]. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, pages 177–192, 2005. URL: http://static.usenix.org/event/sec05/tech/full_papers/chen/chen.pdf.

[84]. H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 177–192, Washington, D.C. USENIX Association, 2015. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-hu.pdf.

[85]. H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: on the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, May 2016. DOI: 10.1109/SP.2016.62.

[86]. J. Pewny, P. Koppe, and T. Holz. STEROIDS for DOPed applications: a compiler for automated data-oriented programming. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 111–126, 2019. DOI: 10.1109/EuroSP.2019.00018.

[87]. K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, Toronto, Canada. ACM, 2018. DOI: 10.1145/3243734.3243739.

[88]. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, 2011. URL: http://security.ece.cmu.edu/aeg/aeg-current.pdf.

[89]. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA. IEEE Computer Society, 2012. DOI: 10.1109/SP.2012.31.

[90]. В. А. Падарян, В. В. Каушан и А. Н. Федотов. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. *Труды института системного программирования РАН*, 26(3):127—144, 2014. DOI: 10.15514/ISPRAS-2014-26(3)-7. / V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov. Automated exploit generation method for stack buffer overflow vulnerabilities. *Proceedings of the Institute for System Programming of the RAS*, 26(3):127–144, 2014. DOI: 10.15514/ISPRAS-2014-26(3)-7.

[91]. V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41(6):373–380, 2015. DOI: 10.1134/S0361768815060055.

[92]. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016. DOI: 10.1109/SP.2016.17.

[93]. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. DOI: 10.1145/360248.360252.

[94]. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010. DOI: 10.1109/SP.2010.26.

[95]. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008. URL: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[96]. A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *In Proceedings of the 6th USENIX Workshop on Offensive Technologies, WOOT '12. USENIX Association*, 2012. URL: https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf.

[97]. BARF : binary analysis and reverse engineering framework, Programa STIC. URL: https://github.com/programa-stic/barf-project.

[98]. A. Wailly. nrop: automated return-oriented programming chaining. URL: https://github.com/awailly/nrop.

[99]. HelpSystems. Agafi (advanced gadget finder). URL: https://github.com/helpsystems/Agafi.

[100]. A. Souchet. rp++. URL: https://github.com/0vercl0k/rp.

[101]. packz. ROPEME - ROP exploit made easy. URL: https://github.com/packz/ropeme.

[102]. acez. xrop. URL: https://github.com/acama/xrop.

[103]. C. Heffner. MIPS ROP IDA plugin. URL: https://github.com/devttys0/ida/tree/master/plugins/mipsrop.

[104]. J. Rudloff. universalrop. URL: https://github.com/kokjo/universalrop.

[105]. N. Mosier. A pair of return-oriented programming utilities: a gadget finder and ROP compiler. URL: https://github.com/nmosier/rop-tools.

[106]. lucasg. ROP database plugin for IDA. URL: https://github.com/lucasg/idarop.

[107]. J. Brower. CROP: ROP payload compiler. URL: https://github.com/jbrower95/crop.

[108]. extremecoders-re. ropgen. URL: https://github.com/extremecoders-re/ropgen.

[109]. Thomas. A fast ROP gadget extractor. URL: https://github.com/mephesto1337/rg.

[110]. I. Jager and D. Brumley. Efficient Directionless Weakest Preconditions. Technical report, Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, 2010. URL: http://security.ece.cmu.edu/pubs/CMUCyLab10002.pdf.

[111]. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. DOI: 10.1145/1273442.1250746.

[112]. T. Dullien and S. Porst. REIL: a platform-independent intermediate representation of disassembled code for static code analysis, 2009. URL: https://static.googleusercontent.com/media/zynamics.com/en//downloads/csw09.pdf.

[113]. В. А. Падарян, М. А. Соловьев и А. И. Кононов. Моделирование операционной семантики машинных инструкций. *Труды института системного программирования РАН*, 19:165—186, 2010. URL: https://www.ispras.ru/proceedings/isp_19_2010/isp_19_2010_165/. / V. A. Padaryan, M. A. Soloviev, and A. I. Kononov. Modeling operational semantics of machine instructions. *Proceedings of the Institute for System Programming of the RAS*, 19:165–186, 2010. URL: https://www.ispras.ru/en/proceedings/isp_19_2010/isp_19_2010_165/.

[114]. C. Heitman and I. Arce. BARF: a multiplatform open source binary analysis and reverse engineering framework. In *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*, 2014. URL: http://sedici.unlp.edu.ar/handle/10915/42157.

[115]. А. В. Вишняков. Верификация семантики линейной последовательности машинных инструкций, 2019. URL: https://vishnya.xyz/vishnyakov-coursework2019.pdf. / A. V. Vishnyakov. Semantic verification of linear machine instruction sequence, 2019. URL: https://vishnya.xyz/vishnyakov-coursework2019.pdf.

[116]. A. Vishnyakov, A. Nurmukhametov, S. Kurmangaleev, and S. Gaisaryan. Method for analysis of code-reuse attacks – reverse engineering of ROP exploits. URL: https://vishnya.xyz/vishnyakov-isprasopen2018.pdf.

[117]. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. URL: www.SMT-LIB.org.

[118]. T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search, 2010. URL: https://www.usenix.org/legacy/events/woot10/tech/full_papers/Dullien.pdf.

[119]. A. Follner, A. Bartel, and E. Bodden. Analyzing the gadgets towards a metric to measure gadget quality. In *Engineering Secure Software and Systems*, pages 155–172. Springer International Publishing, 2016. URL: http://arxiv.org/abs/1605.08159.

[120]. O. L. Fraser. *Urschleim in Silicon: Return Oriented Program Evolution with ROPER*. Master's thesis, Dalhouse University, Halifax, Nove Scotia, 2018. URL: https://dalspace.library.dal.ca/handle/10222/73879.

[121]. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec. 1959. DOI: 10.1007/BF01386390.

[122]. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, technologies, and tools*. In 2th edition edition. Addison Wesley, 2th edition edition, 2006. Chapter Code Generation, pages 563–565.

[123]. J. Stewart and V. Dedhia. ROP compiler. URL: https://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf.

[124]. T. Mortimer. Removing ROP gadgets from OpenBSD. *AsiaBSDCon 2019*:13–21, 2019. URL: https://2019.asiabsdcon.org/proc-body-2019.pdf#page=13.

Информация об авторах / Information about authors

Алексей Вадимович ВИШНЯКОВ работает в отделе компиляторных технологий в Институте системного программирования им. В.П. Иванникова РАН, закончил бакалавриат ВМК МГУ, сейчас получает степень магистра там же. Сфера научных интересов: компьютерная безопасность, возвратно-ориентированное программирование, анализ бинарного кода, символьная интерпретация, обратная инженерия и компиляторы.

Alexey Vadimovich VISHNYAKOV works for the Compiler Technology Department at Ivannikov Institute for System Programming of the RAS, obtained BSc degree in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. He is a M.D. student in the same faculty now. Research interests: computer security, return-oriented programming, binary analysis, symbolic execution, reverse engineering, and compilers.

Алексей Раисович НУРМУХАМЕТОВ — младший научный сотрудник отдела компиляторных технологий ИСП им. В.П. Иванникова РАН, закончил МФТИ по специальности прикладные математика и физика в 2013 году. Сферой научных интересов являются: компиляторы, компьютерная безопасность, возратно-ориентированное программирование.

Aleksei Raisovich NURMUKHAMETOV is a research fellow at the Ivannikov Institute for System Programming of the Russian Academy of Science, the Compiler Technology Department. He obtained both his MSc degree and BSc degree in applied mathematics and physics at the Moscow Institute of Physics and Technology in 2011 and 2013, respectively. He has a strong interest in compilers, computer security, return-oriented programming.