# Метод анализа атак повторного использования кода★

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

**Аннотация.** Обеспечение безопасности программного обеспечения является на сегодняшний день одной из первостепенных задач. Сбои в работе программного обеспечения могут привести к серьезным последствиям, а злонамеренная эксплуатация уязвимостей может причинить колоссальный ущерб. Крупные корпорации уделяют особое внимание анализу инцидентов информационной безопасности. Атаки повторного использования кода, основанные на возвратно-ориентированном программировании (ROP), приобретают всю большую популярность с каждым годом и могут быть применены даже в условиях работы защитных механизмов современных операционных систем. В отличие от обычного шелл-кода, где инструкции размещаются последовательно в памяти, ROP-цепочка состоит из множества маленьких блоков инструкций (гаджетов) и использует стек для связывания этих блоков, что затрудняет анализ ROP-эксплойтов. Целью данной работы является упрощение обратной инженерии ROP-эксплойтов. В этой статье предлагается метод анализа атак повторного использования кода, который позволяет восстановить семантику ROP-цепочки: разбить цепочку на гаджеты, определить семантику отдельных гаджетов и восстановить прототипы вызванных в ходе выполнения цепочки функций и системных вызовов и значения их аргументов. Семантика гаджета определяется его принадлежностью параметризованным типам. Каждый тип задается постусловием (булевым предикатом), которое должно быть всегда истинно после выполнения гаджета. Метод был реализован в виде программного инструмента и апробирован на реальных ROP-эксплойтах, найденных в интернете.

★ Работа поддержана грантом РФФИ № 17-01-00600

**Ключевые слова:** компьютерная безопасность; бинарный анализ; уязвимость; возвратно-ориентированное программирование; ROP; классификация гаджетов; атака повторного использования кода; инцидент информационной безопасности

**DOI:** 10.15514/ISPRAS-2018-30(5)-2

**Для цитирования:** Вишняков А.В, Нурмухаметов А.Р., Курмангалеев Ш.Ф., Гайсарян С.С. Метод анализа атак повторного использования кода. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 31-54. DOI: 10.15514/ISPRAS-2018-30(5)-2

## *1. Введение*

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из первостепенных задач. Программные продукты применяются в повседневно окружающих нас вещах: компьютерах, смартфонах, автомобилях, банкоматах, объектах городской инфраструктуры, медицинском оборудовании жизнеобеспечения и технологиях «интернета вещей». Сбои в работе программного обеспечения могут привести к серьезным последствиям: денежным убыткам, деградации средств коммуникации, задержке в работе экстренных служб, авариям и даже причинению вреда здоровью человека. А злонамеренная эксплуатация уязвимостей может причинить колоссальный ущерб. По данным Национального института стандартов и технологий США ежегодно публикуются тысячи описаний новых уязвимостей CVE (рис. 1) [1,2]. Крупные корпорации уделяют особое внимание анализу инцидентов информационной безопасности.

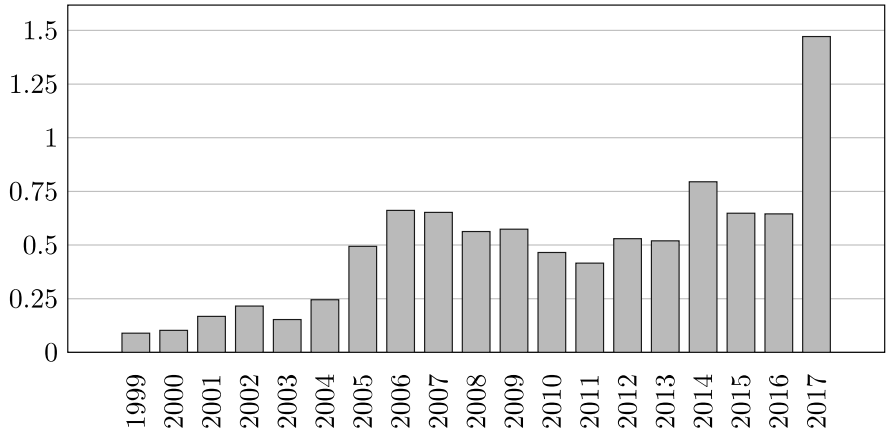


*Рис. 1. Количество (десятки тысяч) новых уязвимостей (CVE) в год*
*Fig. 1. Tens of thousands of vulnerabilities (CVE) by year*

Для эксплуатации уязвимостей в условиях работы защитных механизмов современных операционных систем часто применяется техника возвратно-ориентированного программирования (ROP). Это атака повторного

использования кода, позволяющая обходить защитный механизм, запрещающий региону памяти быть одновременно доступным на запись и исполнение (DEP), и современные реализации рандомизации размещения адресного пространства (ASLR), которые оставляют часть адресного пространства программы нерандомизированной. В частности, в Linux адрес загрузки кода программы часто остается постоянным, а некоторые динамические библиотеки Windows загружаются по фиксированным адресам. Злоумышленник использует кусочки кода из нерандомизированного адресного пространства программы, которые называются гаджетами. Каждый гаджет выполняет некоторые вычисления (например, складывает значения двух регистров) и передает управление следующему гаджету. Гаджеты связываются в цепочку последовательно выполняемых кусочков кода. Таким образом, с помощью цепочки гаджетов можно выполнить некоторые вредоносные действия.

Гаджет – это последовательность инструкций, которая заканчивается инструкцией передачи управления (`ret`). В отличие от обычной программы, инструкции ROP-цепочки не размещаются последовательно в памяти, а вместо этого разбиваются на маленькие гаджеты, связанные инструкциями, которые получают адрес следующего гаджета со стека. Такое стековое связывание инструкций затрудняет анализ ROP-цепочек.

Эксплойт – это программа, входные данные или последовательность команд, использующие уязвимость, чтобы добиться непредусмотренного поведения системы. Целью данной работы является упрощение обратной инженерии ROP-эксплойтов.

В этой статье предлагается метод анализа атак повторного использования кода, который позволяет восстановить семантику ROP-цепочки: разбить цепочку на гаджеты, определить семантику отдельных гаджетов и восстановить вызванные в ходе выполнения цепочки функции и системные вызовы и значения их аргументов.

Статья организована следующим образом. Во втором разделе приводится обзор атак и защитных механизмов, послуживших предпосылками к появлению ROP (подраздел 2.5). В третьем разделе проводится обзор существующих методов анализа ROP-атак. В четвертом разделе описывается предложенный метод анализа атак повторного использования кода. В пятом разделе рассматриваются детали реализации предлагаемого метода. Результаты практического применения приводятся в шестом разделе.

## *2. Обзор атак и защитных механизмов*

В этом разделе приводится обзор атак на переполнение буфера на стеке. Описываются защитные механизмы операционной системы: ограничение исполняемых областей (DEP) и рандомизация размещения адресного пространства (ASLR). В разделе 2.5 дается определение возвратно-ориентированного программирования – метода эксплуатации уязвимости

переполнения буфера на стеке, позволяющего обойти DEP и современные реализации ASLR.

## 2.1 Уязвимость переполнения буфера на стеке

Уязвимость переполнения буфера на стеке возникает, когда размер данных, записываемых в буфер на стеке, превышает размер этого буфера [3]. Например, в приведенной ниже программе на Си уязвимая функция `vul` не проверяет длину сроки `str`, записываемой в буфер на стеке фиксированного размера `buf`. Если длина первого аргумента командной строки `argv[1]` окажется большей или равной размеру буфера `buf`, то произойдет переполнение буфера на стеке.

```
void vul(char *str) {
    char buf[512];
    strcpy(buf, str);
}
int main(int argc, char *argv[]) {
    vul(argv[1]);
    return 0;
}
```

На рисунке 2a показан стековый фрейм функции `vul` до переполнения. Стек в архитектуре x86 растет от больших адресов к меньшим (на рисунке – сверху вниз). Аргументы функции поочередно кладутся на стек справа налево. При вызове функции адрес возврата кладется на стек, после чего функция может сохранить старое значение регистра `ebp` и выделить на стеке память для локальных переменных, в нашем случае – для буфера `buf`. Данные в буфер записываются в порядке возрастания адресов (на рисунке – снизу вверх). Переполнение буфера приводит к перезаписи ячеек выше по стеку, в том числе адреса возврата, после чего почти всегда следует аварийное завершение программы.

Эксплуатация уязвимости переполнения буфера на стеке позволяет выполнять произвольный код. Рассмотрим ситуацию, когда злоумышленник контролирует значение первого аргумента командной строки `argv[1]`, а следовательно, контролирует значения, записываемые в буфер `buf`. В таком случае злоумышленник может добиться перезаписи адреса возврата указателем на размещенный на стеке вредоносный код (рис. 2b). Таким образом, после возврата из функции `vul` управление передастся на сформированный злоумышленником код. Обычно в качестве такого кода используется код, приводящий к вызову командной оболочки операционной системы, который называется шелл-кодом. Чтобы избежать негативных последствий от переполнения буфера на стеке, появились различные защитные механизмы.

## 2.2 Ограничение исполняемых областей (DEP)

Ограничение исполняемых областей (DEP) – защитный механизм операционной системы, запрещающий исполнение кода из областей памяти, помеченных как «данные». Попытка исполнения кода из помеченных областей вызывает исключение и влечет за собой аварийное завершение программы. В частности, стек и куча становятся недоступными для выполнения, что предотвращает выполнение размещенного на них вредоносного кода. Механизм успешно применяется в операционных системах Windows, Linux и др.

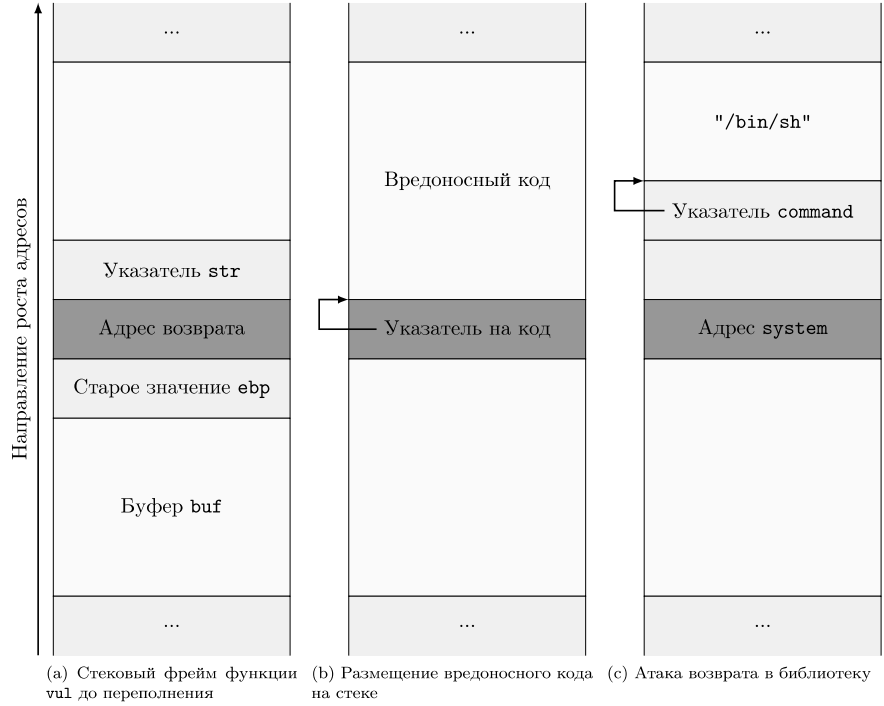


*Рис. 2. Стековый фрейм функции* `vul` *и способы эксплуатации уязвимости переполнения буфера* `buf` *на стеке*
*Fig. 2. A stack frame and different buffer overflow exploitation techniques*

## 2.3 Атака возврата в библиотеку

Для обхода DEP используется атака возврата в библиотеку. Атака заключается в подмене адреса возврата адресом некоторой библиотечной функции, например, функции system из библиотеки `libc`.

На рис. 2c показано состояние стека после переполнения. Адрес возврата перезаписан адресом функции `system(const char *command)`. Выше

лежит произвольный адрес возврата из функции system и ее единственный аргумент `command`, который является указателем на нуль-терминированную строку `"/bin/sh"`, размещенную следом за указателем. Таким образом, после возврата из функции `vul` вызовется библиотечная функция `system("/bin/sh")`, которая в свою очередь вызовет командную оболочку операционной системы.

## 2.4 Рандомизация размещения адресного пространства (ASLR)

Рандомизации размещения адресного пространства (ASLR) – защитный механизм операционной системы, позволяющий размещать ключевые элементы процесса (образ программы, стек, куча, динамические библиотеки) по различным адресам во время загрузки исполняемого файла. Данная защита затрудняет проведение атаки возврата в библиотеку, т.к. адрес библиотечной функции неизвестен до загрузки программы и отличается для каждого запуска.

Следует отметить, что рандомизация адресов исполняемых секций программы или библиотеки требует, чтобы они были скомпилированы в позиционно-независимый код, что не всегда выполняется. Так в Linux адрес загрузки кода программы часто остается постоянным, а некоторые динамические библиотеки Windows загружаются по фиксированным адресам. Таким образом, в условиях работы современных реализаций ASLR часть адресного пространства программы остается нерандомизированной.

## 2.5 Возвратно-ориентированное программирование (ROP)

Возвратно-ориентированное программирование (ROP) [4] – метод эксплуатации уязвимости переполнения буфера на стеке, который по сути является обобщением атаки возврата в библиотеку. Метод так же применим в условиях работы DEP, но представляет большую опасность, т.к. может быть использован для обхода современных реализаций ASLR, когда часть адресного пространства остается нерандомизированной (разд. 2.4).

ROP предполагает использование последовательностей инструкций в нерандомизированных исполняемых областях памяти, которые заканчиваются инструкцией передачи управления (`ret`). Такие последовательности инструкций называются гаджетами. Следует отметить, что архитектура x86 не требует выравнивания адресов инструкций, т.е. позволяет выполнение инструкций, размещенных по произвольным адресам памяти. А значит, некоторая последовательность инструкций в программе может содержать в себе гаджет, отсутствовавший в коде программы. Ниже приводятся бинарный и ассемблерный коды гаджета, который содержится внутри последовательности инструкций оригинальной программы.

```
f7c7070000000f9545c3 → test edi, 0x7 ;
```

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

Гаджеты собираются в цепочки, а их адреса размещаются от адреса возврата на стеке так, чтобы первый гаджет передавал управление второму, второй – третьему и т.д. Таким образом, с помощью цепочки гаджетов можно выполнить некоторые вредоносные действия.

На рисунке 3 приводится состояние стека после размещения на нем ROP-цепочки, которая производит запись значения `memValue` по адресу `memAddr`. Адрес возврата перезаписан адресом первого гаджета. После возврата из функции, в которой произошло переполнение, управление передастся первому гаджету, который загрузит со стека значение `memValue` на регистр `eax`. При возврате (после выполнения инструкции `ret`) первый гаджет передаст управление второму гаджету, который в свою очередь загрузит значение `memAddr` на регистр `edx`. Потом третий гаджет сохранит значение регистра `eax` (`memValue`) по адресу `edx` (`memAddr`). Далее управление передастся четвертому гаджету и т.д.

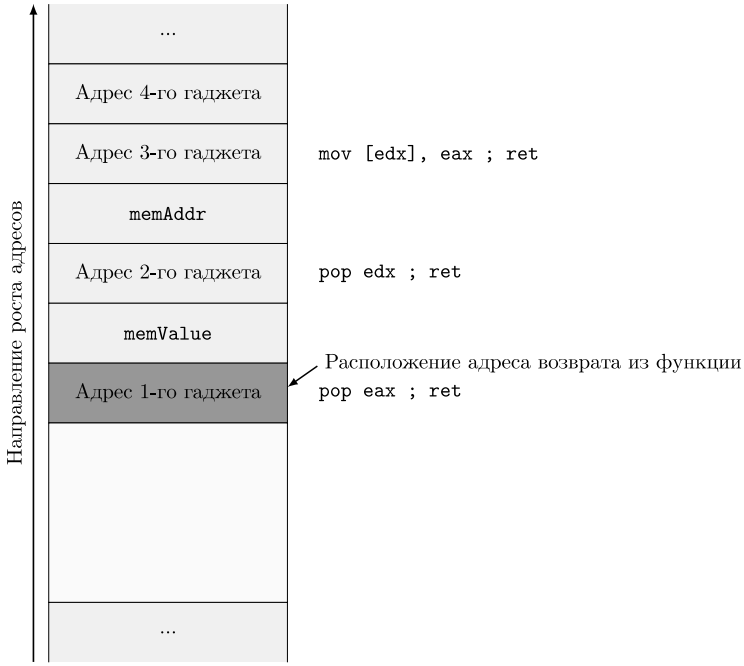


*Рис. 3. Состояние стека после размещения на нем ROP-цепочки, которая производит запись значения* `memValue` *по адресу* `memAddr`

*Fig.3. A ROP chain, storing* `memValue` *to* `memAddr`*, stack frame*

Ниже приводится эта же ROP-цепочка в бинарном виде, которая записывает значение `"/bin"` по адресу `0x0830caa0`. Эта последовательность байтов размещается злоумышленником на стеке от адреса возврата.

```
00000000  47 65 06 08 2f 62 69 6e  3d 76 07 08 a0 ca 30 08  |Ge../bin=v....0.|
00000010  b5 8b 08 08                                       |....|
00000014
```

## *3. Обзор существующих решений*

В этом разделе описываются существующие решения проблем определения семантики гаджета и анализа атак повторного использования кода.

### 3.1 Определение семантики гаджета

Schwartz и др. [5] предложили определять функциональность гаджета его принадлежностью типам, которые приведены в таблице 1. Набор типов гаджетов задает новую архитектуру набора команд (ISA), в которой каждый тип гаджета исполняет роль инструкции. Семантика каждого типа гаджета определяется постусловием (булевым предикатом) $\mathscr{B}$, которое должно быть всегда истинно после выполнения гаджета.

*Табл. 1. Типы гаджетов.* `[Addr]` *означает доступ к памяти по адресу* `Addr`, `∘` *– бинарную операцию.* `a ← b` *означает, что конечное значение* `a` *равно начальному значению* `b`. `X ∘← Y` *– сокращение для* `X ← X ∘ Y`
*Table 1. Gadget types*

| **Тип** | **Параметры** | **Определение семантики** |
|---|---|---|
| `NoOpG` | — | Не меняет ничего в памяти и на регистрах |
| `JumpG` | `AddrReg` | `IP ← AddrReg` |
| `MoveRegG` | `InReg`, `OutReg` | `OutReg ← InReg` |
| `LoadConstG` | `OutReg`, `Offset` | `OutReg ← [SP + Offset]` |
| `ArithmeticG` | `InReg1`, `InReg2`, `OutReg`, `∘` | `OutReg ← InReg1 ∘ InReg2` |
| `LoadMemG` | `AddrReg`, `OutReg`, `Offset` | `OutReg ← [AddrReg + Offset]` |
| `StoreMemG` | `AddrReg`, `InReg`, `Offset` | `[AddrReg + Offset] ← InReg` |
| `ArithmeticLoadG` | `AddrReg`, `OutReg`, `Offset`, `∘` | `OutReg ∘← [AddrReg + Offset]` |
| `ArithmeticStoreG` | `AddrReg`, `InReg`, `Offset`, `∘` | `[AddrReg + Offset] ∘← InReg` |

Будем говорить, что последовательность инструкций $\mathscr{J}$ удовлетворяет постусловию $\mathscr{B}$, если для любого начального состояния после выполнения $\mathscr{J}$ постусловие $\mathscr{B}$ истинно. Начальное состояние состоит из присваиваний регистрам и памяти некоторых начальных значений.

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]`.

### 3.1.1 Семантический анализ

Для того чтобы определить, удовлетворяет ли последовательность инструкций $\mathscr{I}$ постусловию $\mathscr{B}$, Schwartz и др. [5] используют известную технику из формальной верификации – вычисление слабейшего предусловия [6]. Проще говоря, слабейшее предусловие $wp(\mathscr{I}, \mathscr{B})$ для последовательности инструкций $\mathscr{I}$ и постусловия $\mathscr{B}$ – это булево предусловие, которое описывает, когда $\mathscr{I}$ завершается в состоянии, удовлетворяющем $\mathscr{B}$. Слабейшие предусловия используются, чтобы убедиться, что определение семантики гаджета всегда выполняется после выполнения последовательности инструкций $\mathscr{I}$. Для этого достаточно проверить:

$$wp(\mathscr{I}, \mathscr{B}) \equiv true.$$

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

Однако формальная верификация гаджетов показала себя очень медленной на практике. Для ускорения процесса опредеделения, принадлежит ли гаджет тому или иному типу, инструкции гаджета предварительно несколько раз выполняются с использованием случайных входных данных, и проверяется истинность $\mathscr{B}$. Если $\mathscr{B}$ 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 $\mathscr{B}$ истинно для всех выполнений.

Выполнение со случайными входными данными может быть также использовано для выявления возможных значений параметров (табл. 1) гаджетов. Например, посмотрев на значения регистров и на адреса чтения из памяти, можно вычислить набор возможных смещений (`Offset`) для гаджета загрузки из памяти `LoadMemG`.

## 3.2 deRop

В отличие от использования традиционного шелл-кода, который внедряется в память процесса, возвратно-ориентированное программирование позволяет производить произвольные вычисления, используя уже имеющийся в памяти код. Поэтому для анализа ROP-атак затруднительно использовать существующие традиционные инструменты анализа бинарного кода. Для решения озвученной проблемы был предложен инструмент deRop [7] – ROP-эксплойт приводится к семантически эквивалентному обычному шелл-коду, который уже может быть проанализирован существующими инструментами. Авторы преимущественно используют статический анализ и выделяют следующие трудности анализа ROP-атак:

**Обнаружение гаджетов.** При эксплуатации переполнения буфера на стеке перед адресом первого гаджета (которым будет перезаписан адрес возврата) записывается буфер произвольных незначительных данных. Более того, между адресом первого и второго гаджета также могут быть пропущены ячейки (например, когда функция, в которой происходит переполнение, чистит за собой аргументы со стека инструкций `ret n`). Несмотря на то что deRop пытается использовать статический анализ, насколько это возможно, избегая использование динамического анализа, обнаружение первых двух гаджетов производится с использованием отладчика.

**Отслеживание указателя стека.** В ROP-эксплойте указатель стека используется для получения адреса следующего гаджета так же, как указатель инструкции (счетчик команд) – для получения адреса следующей инструкции. Поэтому необходимо отслеживать указатель стека для обнаружения следующего гаджета.

**Расположение стека и констант.** Для загрузки констант в регистр в шелл-коде обычно используются `mov reg, imm`, в то время как в ROP обычно используются `pop reg`. Расположение стека в исходной ROP-цепочке отлично от расположения в выходном семантически эквивалентном ей шелл-коде. Поэтому необходимо отслеживать расположение констант на стеке.

**Вызовы функций.** Некоторые гаджеты в ROP-цепочке используются для вызова функций. Необходимо выявлять такие вызовы функций и вызывать их традиционным образом. Более того, необходимо определять значения аргументов функции (в т.ч. аргументов, которые являются константой или указателем) для каждого вызова.

**Циклы.** ROP-цепочка может содержать циклы. Необходимо уметь их обнаруживать и определять условие выхода из цикла.

### 3.2.1 Постобработка

Как только все гаджеты были проанализированы, производятся несколько этапов постобработки для упрощения выходного кода:

**Данные в памяти.** Вычисляются значения операндов инструкций, обращающихся к памяти, и операнды заменяются константами.

**Нулевые байты.** Обычно требуется, чтобы шелл-код не содержал нулевых байтов, т.к. это приводит к обрезанию шелл-кода после некоторых операций (например, `strcpy`). Данная проблема решается заменой всех нулевых байтов на ненулевые значения и добавлением декодера в начало шелл-кода, который восстановит оригинальные значения.

**Адрес возврата.** Адрес возврата в эксплойте заменяется адресом начала результирующего шелл-кода.

## 3.3 ROPMEMU

ROPMEMU [8] – фреймворк для анализа сложных атак повторного использования кода. Авторы используют динамический подход к анализу бинарного кода и выделяют следующие проблемы анализа ROP-атак (C1—C3 уже были упомянуты в подразделе 3.2):

**[C1] Избыточность** – большинство ROP-гаджетов содержат лишние инструкции. Например, гаджет, предназначенный для инкрементирования `eax`, может также загружать (`pop`) значение со стека до передачи управления следующему гаджету (`ret`).

**[C2] Стековое связывание инструкций** – в отличие от обычной программы, где инструкции размещаются последовательно в памяти, ROP-эксплойт разбивается на маленькие гаджеты, связанные в цепочку инструкциями косвенной передачи управления (`ret`).

**[C3] Нехватка значений констант** – ROP-цепочки обычно составляются из параметризованных гаджетов (например, загрузки произвольного значения в регистр `rax`), которые используют параметры, сохраненные на стеке.

**[C4] Условные ветвления** – в отличие от традиционного изменения указателя инструкции (счетчика команд) условное ветвление в ROP-цепочке изменяет указатель стека. Таким образом, простой условный переход реализизуется несколькими гаджетами (стр. 18—19 [9]). Для приведения цепочки к более читаемому коду необходимо распознавать такие условные ветвления и заменять их одной инструкцией ветвления.

**[C5] Возврат в функции** – вызовы функций в ROP обычно реализуются простым возвратом (`ret`) во входную точку функции. Т.к. обычные гаджеты также часто берутся из кода, расположенного внутри библиотек, необходимо уметь отличать вызов функции от очередного гаджета.

**[C6] Динамически генерируемые цепочки** – ROP-цепочка не обязательно сразу целиком размещается в памяти, а могут быть использованы гаджеты, которые подготовят выполнение других гаджетов в будущем.

**[C7] Условие останова** – авторы предполагают, что аналитик способен определить начало ROP-цепочки в памяти. Однако необходимо завершить процесс эмуляции, когда все гаджеты были извлечены.

В фреймворке ROPMEMU используется набор различных техник для анализа ROP-цепочек и восстановления эквивалентного им кода в форме, которая может быть проанализирована традиционными инструментами обратной инженерии, такими как IDA Pro [10]. Предполагается, что у аналитика имеются дамп памяти и входная точка (первой) ROP-цепочки в нем. Оставшиеся динамически генерируемые цепочки восстанавливаются самим фреймворком, который имеет пять основных фаз анализа.

### 3.3.1 Многопутевая эмуляция

На этом шаге эмулируются ассемблерные инструкции, из которых состоит ROP-цепочка (C2). Исследуются все возможные ветвления и для каждого пути выполнения генерируется независимая трасса (аннотированная значениями регистров и памяти). Эмулятор также распознает возвраты в библиотечные функции, пропускает их тело и симулирует их выполнение, генерируя фиктивные данные и возвращаемое значение (C5).

Эмулятор изначально считывает содержимое памяти из дампа памяти и поддерживает теневую память [11]. Условие останова (C7) определяется набором эвристик, основанных на принципе локальности (эмулятор обнаруживает большое относительное изменение указателя стека) и длине гаджета, исключая обнаруженные вызовы функций. Как только срабатывает условие останова, содержимое теневой памяти и трасса выполнения сохраняются на диск и исследуются на предмет наличия новых ROP-цепочек. Если таковые найдены, эмулятор перезапускается, чтобы проанализировать следующую цепочку, и так до тех пор, пока все динамически генерируемые цепочки не будут обнаружены и проанализированы (C6).

Для ROP-цепочек со сложным потоком управления, простой подход, основанный на эмуляции, не достаточен для анализа всего ROP-эксплойта. Ведь покрытие ограничено только выполненными условными переходами, которые часто зависят от фиктивных возвращаемых значений функций, сгенерированных эмулятором. Данную проблему решает многопутевая эмуляция, которая является адаптированной к ROP-цепочкам версией алгоритма многопутевого выполнения [12]. В частности, эмулятор распознает, когда указатель стека изменяется в зависимости от содержимого регистра флагов. В конце процесса эмуляции из трассы получается список всех точек ветвления вместе со значениями флагов в каждой из них. Далее эмулятор перезапускается с указанием инвертировать переход в точке ветвления. Таким образом, выполнение пройдет по другому пути. Исследование ветвлений прекращается, когда все ветви проанализированы.

Однако, при наличии циклов в ROP-цепочке, эмулятор может застрять в бесконечном пути выполнения. Для решения этой проблемы отслеживается число повторений указателя стека во время выполнения инструкций ветвления. Если это число превосходит некоторый допустимый порог, эмулятор инвертирует переход, чтобы насильно прекратить цикл и исследовать оставшуюся часть графа потока управления.

### 3.3.2 Разбиение трассы

На этой фазе анализируются все сгенерированные эмулятором трассы, удаляются повторения и извлекаются уникальные блоки кода. Каждая трасса разрезается в каждой точке ветвления, генерируется новый блок, который

сохраняется в отдельную трассу. В результате, будет получен набор трасс, ассоциированных с каждым «базовым блоком» в цепочке.

### 3.3.3 Отвязывание инструкций от стека

На этом этапе из трассы удаляются все инструкции безусловной передачи управления (`ret`, `call`, `jmp`) и содержимое последовательно выполняемых гаджетов сливается в один базовый блок (C2). Затем инструкции mov упрощаются, благодаря вычислению их операндов (например, `mov rax, [rsp + 0x30]`). Инструкции `pop` заменяются инструкциями `mov`, все необходимые значения получаются из соответствующих ячеек на стеке (C3).

### 3.3.4 Восстановление графа потока управления

На этом проходе все трассы сливаются в единое графовое представление. Потом граф транслируется в настоящую программу x86, благодаря распознаванию инструкций, ассоциированных с условными ветвлениями, и замене их традиционными, использующими указатель инструкции (счетчик команд) условными переходами (C4).

Следующей задачей данного прохода является обнаружение и сворачивание циклов. ROP-цепочки могут содержать как возвратно-ориентированные циклы, так и развернутые циклы. В первом случае ROP-инструкции используются для повторения одного и того же блока гаджетов на стеке с выходом по условию. Развернутые циклы в свою очередь повторяют одну и ту же последовательность гаджетов заранее определенное (константное) количество раз. Фреймворк автоматически определяет рекуррентные паттерны и заменяет их более компактным куском ассемблерного кода, представляющим из себя цикл с той же семантикой.

Полученный в результате код оборачивается рабочими прологом и эпилогом функции и включается в отдельный ELF файл, чтобы позволить использовать традиционные инструменты обратной инженериии (например, IDA Pro [10]) для работы с ним.

### 3.3.5 Бинарная оптимизация

На заключительном шаге применяются известные компиляторные преобразования для дальнейшего упрощения ассемблерного кода в ELF файле. В частности, удаляется мертвый код, применяются преобразования, описанные в п. 3.2.1, и генерируется чистая и оптимизированная версия эксплойта (C1).

## *4. Метод анализа атак повторного использования кода*

Предлагаемый в данной статье метод анализа атак повторного использования кода позволяет восстановить семантику ROP цепочки и проследить за ходом атаки. По бинарной ROP-цепочке восстанавливается последовательность

вызванных гаджетов. Найденные гаджеты классифицируются по семантическим типам, и определяются значения параметров гаджетов. Помимо того в цепочке выявляются вызовы функций и системные вызовы, восстанавливаются их прототипы и значения аргументов. Следует отметить, что в данной работе не ставится задача проанализировать все пути выполнения ROP-цепочки, а достаточно разбора хотя бы одного из них. Поэтому предлагаемый метод не учитывает условные ветвления в ROP-цепочках.

## 4.1 Фрейм гаджета

Для декомпозиции бинарной ROP-цепочки на гаджеты вводится понятие фрейма гаджета аналогичное стековому кадру x86. Цепочка гаджетов разбивается на фреймы. Фрейм гаджета содержит в себе значения параметров гаджета (например, значение, загружаемое на регистр со стека гаджетом `LoadConstG`) и адрес следующего гаджета. Начало фрейма определяется значением указателя стека перед выполнением первой инструкции гаджета.


*Рис. 4. Фрейм гаджета* `pop eax ; ret 8`
*Fig.4.* `pop eax ; ret 8` *gadget frame*

На рисунке 4 фигурной скобкой обозначены гараницы фрейма гаджета `pop eax ; ret 8`. Гаджет загружает значение со стека в `eax`, что соответствует типу загрузки константы `LoadConstG: eax ← [esp + 0]`. Гаджет имеет размер фрейма `FrameSize = 16`, а адрес следующего гаджета располагается по смещению 4 от начала фрейма (`NextAddr = [esp + 4]`).

## 4.2 Классификация гаджетов

Классификации гаджетов [13] позволяет определить семантику гаджетов. Семантика гаджета определяется набором булевых постусловий (типов гаджета) и значениями их параметров, которым удовлетворяют инструкции

гаджета (разд. 3.1). Для анализа ROP-цепочек, найденных в интернете, предложенного Schwartz и др. [5] набора типов гаджетов (табл. 1) оказалось недостаточно, и он был расширен дополнительными типами, которые приводятся в таблице 2. Более того, были добавлены типы гаджетов, которые не гарантируют сохранения управления (внизу таблицы 2).

*Табл. 2. Дополнительные типы гаджетов.* `[Addr]` *означает доступ к памяти по адресу* `Addr`, `∘` *– бинарную операцию.* `a ← b` *означает, что конечное значение* `a` *равно начальному значению* `b`. `X ∘← Y` *– сокращение для* `X ← X ∘ Y`
*Table 2. Extended gadget types*

| **Тип** | **Параметры** | **Определение семантики** |
|---|---|---|
| `JumpMemG` | `AddrReg`, `Offset` | `IP ← [AddrReg + Offset]` |
| `GetSPG` | `OutReg` | `OutReg ← SP` |
| `InitConstG` | `OutReg`, `Value` | `OutReg ← Value` |
| `InitMemG` | `AddrReg`, `Value`, `Offset`, `Size` | `[AddrReg + Offset] ← Value` |
| `NegG` | `InReg`, `OutReg` | `OutReg ← −InReg` |
| `ArithmeticConstG` | `InReg`, `OutReg`, `Value`, `∘` (+/⊕) | `OutReg ← InReg ∘ Value` |
| `ShiftStackG` | `Offset`, `∘` (+/−) | `SP ∘← Offset` |
| `PushAllG` | — | `([ESP − 4] ← EAX) ∧`<br>`([ESP − 8] ← ECX) ∧`<br>`([ESP − 12] ← EDX) ∧`<br>`([ESP − 16] ← EBX) ∧`<br>`([ESP − 20] ← ESP) ∧`<br>`([ESP − 24] ← EBP) ∧`<br>`([ESP − 28] ← ESI) ∧`<br>`(EIP ← EDI)` |
| **Не сохраняют управление** | | |
| `JumpSPG` | — | `IP ← SP` |
| `CallG` | `AddrReg` | `IP ← AddrReg` |
| `CallMemG` | `AddrReg`, `Offset` | `IP ← [AddrReg + Offset]` |
| `IntG` | `Value` | Вызвать прерывание `Value` |
| `SyscallG` | — | Системный вызов |

Классификация гаджета производится на основе анализа эффектов выполнения гаджета на случайных входных данных. Инструкции гаджета транслируются в промежуточное представление. Далее запускается процесс интерпретации промежуточного представления. 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` 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.

В результате классификации гаджета будут получены семантические типы гаджета и их параметры, а также информация о фрейме гаджета (разд. 4.1) – размер фрейма (`FrameSize`) и смещение ячейки с адресом следующего гаджета относительно начала фрейма (`NextAddr`).

Следует отметить, что классификация гаджета производится в результате выполнения гаджета на ограниченном количестве наборов конкретных входных данных, что в общем случае не гарантирует соответствия семантике результата выполнения гаджета на произвольных входных данных. Для точной классификации необходимо производить формальную верификацию семантики гаджета, как описывается в подразделе 3.1.1. Таким образом, возможна неверная классификация гаджета. Однако доля неверно классифицированных гаджетов после 10 запусков на случайных входных данных незначительна, что является приемлемым для задачи восстановления семантики ROP-цепочек.

## 4.3 Восстановление семантики ROP-цепочек

Бинарная ROP-цепочка загружается на теневой стек. Используя информацию о фрейме предыдущего гаджета, полученную в результате классификации, один за другим классифицируются гаджеты в цепочке. Смещение ячейки с адресом следующего гаджета относительно начала фрейма и размер фрейма по сути показывают, где брать адрес следующего гаджета для классификации и где начинается его фрейм соответственно. Указатель теневого стека всегда указывает на начало фрейма последнего классифицированного гаджета.

Для восстановления значений регистров и памяти перед выполнением гаджета (например, для восстановления аргументов системного вызова или функции) поддерживается общая для всех гаджетов теневая память [11]. Изначально теневая память пуста. Последовательно для каждого классифицированного гаджета цепочки производится несколько запусков процесса интерпретации его промежуточного представления с теневой памятью, выступающей в качестве начальных значений регистров и памяти. Считанные регистры и память, не содержащиеся в теневой памяти, генерируются случайном образом при каждом запуске интерпретации. Конечные значения регистров и памяти, которые не менялись от запуска к запуску, обновляются в теневой памяти.

Значения всех загружаемых ROP-цепочкой констант могут быть восстановлены из теневого стека. Одной лишь классификации гаджетов для этого недостаточно, т.к. она не учитывает данные, расположенные на теневом стеке, а генерирует считанные со стека значения случайным образом.

Классификация гаджета загрузки константы `LoadConstG` позволяет определить регистр `OutReg`, на который производится загрузка константы, и смещение `Offset`, по которому происходит чтение значения константы со стека. После классификации гаджета `LoadConstG` в теневую память добавляется значение регистра `OutReg`, загруженное с теневого стека по смещению `Offset` от указателя теневого стека.

Для обхода DEP в 32-битных Windows программах часто используется гаджет `PushAllG` (`pushad ; ret`), при помощи которого вызывается функция WinAPI `VirtualProtect` [14] (которая сделает стек исполняемым) и передается управление обычному шелл-коду, размещенному выше на стеке (рис. 5). Дело в том, что инструкция `pushad` сохраняет регистры общего назначения на стек. Если предварительно проинициализировать регистры соответствующими значениями, то на стеке окажется обыкновенная ROP-цепочка.

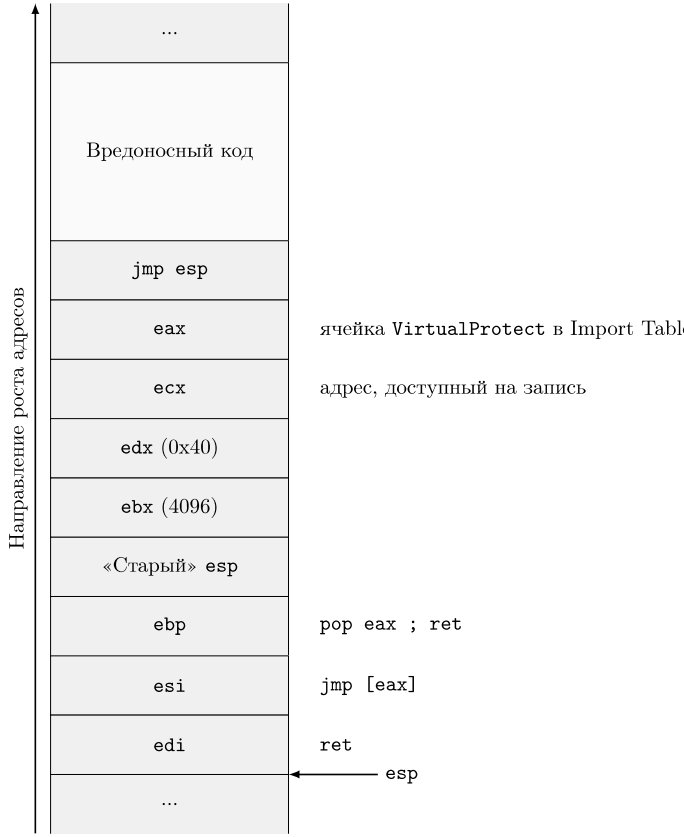


*Рис. 5. Состояние стека после выполнения инструкции* `pushad`
*Fig.5. A* `pushad` *stack frame*

Таким образом, сначала на регистр edi загружается адрес `NoOpG` гаджета (`ret`), на `esi` – адрес гаджета, который вызовет функцию `VirtualProtect` (например, `jmp [eax]`, при этом на `eax` предварительно загружается адрес ячейки `VirtualProtect` в таблице импортированных символов). На регистры `ebx`, `edx` и `ecx` загружаются 2—4 аргументы `VirtualProtect` соответственно. А в качестве адреса возврата из функции `VirtualProtect` на регистр `ebp` загружается адрес `ShiftStackG` гаджета (`pop eax ; ret`), инкрементирующего указатель стека. После выполнения инструкции `pushad` значения этих регистров будут лежать на стеке, как изображено на рисунке 5. В свою очередь выполнение инструкция возврата ret передаст управление по адресу гаджета, записанному в последний сохраненный регистр `edi` (`ret`). Далее управление передастся гаджету, который вызовет `VirtualProtect(esp, ebx, edx, ecx)`. А после возврата из функции `VirtualProtect` стек станет исполняемым и управление передастся гаджету, чей адрес был предварительно загружен на регистр `ebp` (`pop eax ; ret`). В результате, вызовется гаджет `JumpSPG` (`jmp esp`), который передаст управление обычному шелл-коду, размещенному сразу же выше по стеку и доступному теперь на исполнение.

Для такого гаджета `PushAllG` на теневой стек записываются соответсвующие значения регистров. Гаджет `JumpSPG` в свою очередь интерпретируется как передача управления обычному шелл-коду, размещенному на стеке, и производится дизассемблирование его байтов.

Следует отметить, что ROP-цепочка может предварительно записать гаджет в память, чтобы потом его использовать. Ниже приводится пример такой цепочки. Сначала в регистр `edx` загружается машинный код гаджета `mov [eax + ebp * 4], ebx ; ret`. Затем этот гаджет сохраняется в память по адресу `eax`. Далее загружаются параметры гаджета: `ebx` и `ebp`. Наконец передается управление по адресу `eax`, куда был предварительно сохранен гаджет, записывающий в память значение регистра `ebx` по адресу `eax + ebp * 4`.

```
pop edx ; ret  // edx = "\x89\x1c\xa8\xc3"
mov [eax], edx ; ret
pop ebx ; pop ebp ; ret
jmp eax  // mov [eax + ebp * 4], ebx ; ret
```

Таким образом, если во время разбора ROP-цепочки адрес следующего гаджета содержится в теневой памяти, то классифицируется гаджет из теневой памяти. Если же после классификации окажется, что этот гаджет нельзя отнести ни к одному из типов, то считается, что это передача управления обычному шелл-коду, предварительно сохраненному в память. Байты шелл-кода из теневой памяти также дизассемблируются.

### 4.3.1 Восстановление функций и системных вызовов

Функция может быть вызвана из ROP-цепочки с использованием гаджетов `JumpG`, `JumpMemG`, `CallG`, `CallMemG`, или же ее адрес может быть просто размещен на стеке. Системный вызов выполняется гажетом `IntG` в 32-разрядной операционной системе, а гаджетом `SyscallG` в 64-разрядной. Номер системного вызова, а также значения аргументов функции и системного вызова восстанавливаются из теневой памяти. Если по адресу аргумента в теневой памяти располагается нуль-терменированная строка, то она тоже восстанавливается.

Для ROP-цепочек под Linux по номеру системного вызова получается его имя. Имена вызванных функций можно восстановить, если функция была вызвана по адресу, считанному из таблицы импортированных символов (GOT в ELF и IAT в PE). Прототипы функций и системных вызовов Linux ищутся в man-pages [15], а прототипы функций Windows – в API Monitor [16].

## *5. Программная реализация*

Описанный метод был реализован в виде программного инструмента. Инструмент получает на вход бинарную ROP-цепочку и исполняемый файл, в котором содержатся использованные в цепочке гаджеты. Следует отметить, что в данной работе не ставится задача поиска ROP-цепочки в эксплойте. Решение этой задачи возлагается на аналитика. Поддерживаются следующие форматы исполняемых файлов: ELF32, ELF64, PE32 и PE32+. Цепочки, использующие гаджеты из разных исполняемых файлов, в данный момент не поддерживаются.

### 5.1 Интерпретация промежуточного представления инструкций гаджета

В данной работе используется разработанное в ИСП РАН промежуточное представление инструкций [17], удовлетворяющее SSA-форме и имеющее трехадресный код. Адресные пространства памяти и регистров представляются в виде двух байтовых массивов. Адресное пространство регистров состоит из всех регистров машины с учетом наложений и пересечений. Для учета побочных эффектов используется слово состояния, аналогичное регистру флагов x86.

Инструкции гаджета транслируются в промежуточное представление, интерпретация которого позволяет получить начальные и конечные значения регистров и памяти. Изначально для каждого адресного пространства карты считанных и сохраненных значений пусты. Инструкции промежуточного представления заменяются эквивалентными блоками инструкций x86-64. При этом инструкции сохранения (STORE) заменяются на вызовы функции, обновляющей карту сохраненных значений. А инструкции чтения (LOAD)

заменяются вызовом функции, возвращающей актуальное значение, которая выполняет одно из следующих действий:

- считывает значение из карты сохраненных значений, если оно там присутствует;
- считывает значение из карты считанных значений, если оно там присутствует и отсутствует в карте сохраненных значений;
- добавляет в карту считанных значений случайно сгенерированное значение при первом обращении по адресу.

Далее производиться выполнение полученного x86-64 кода. В результате будут получены начальное и конечное состояния адресных пространств.

## 5.2 Разбор ROP-цепочки

Эмулируется загрузка исполняемого файла в виртуальное адресное пространство с разрешением релокаций. По адресу каждого гаджета в загруженном исполняемом файле дизассемблируются инструкции до инструкции передачи управления. Полученные инструкции транслируются в промежуточное представление и классифицируются. Аргументы функций и системных вызовов восстанавливаются согласно соглашению о вызове из теневой памяти, а в качестве возвращаемого значения функции в теневую память добавляется некоторое фиктивное значение. Далее происходит обновление теневого стека и теневой памяти, как описано в разделе 4.3.

В результате, будет получен текстовый файл, в котором будут перечислены последовательно вызванные гаджеты, а также их типы и параметры. Более того, будут приведены прототипы вызванных функций и системных вызовов с восстановленными значениями аргументов. Если ROP-эксплойт завершается вызовом обычного шелл-кода, то будут выведены его дизассемблированные инструкции.

# *6. Результаты практического применения*

Предложенный в этой статье метод анализа атак повторного использования кода был апробирован на реальных ROP-эксплойтах, найденных в интернете. Бинарные ROP-цепочки извлекались вручную. Затем определялся исполняемый файл, из которого использовались гаджеты в цепочке.

Хорошими отправными точками для поиска ROP-эксплойтов послужили фреймворк для тестирования на проникновение Metasploit [18] и открытая база данных эксплойтов EDB [19]. К сожалению, исполняемый файл, из которого собирались гаджеты в цепочку, редко прилагается к эксплойту. В лучшем случае будут указаны версия программы, операционная система и/или дистрибутив. Поэтому исполняемые файлы часто приходится искать вручную и проверять, находятся ли по тем же адресам заявленные в эксплойте гаджеты. Для поиска старых версий пакетов дистрибутива Debian существует проект snapshot.debian.org [20], который несколько раз в день сохраняет текущее

состояние дистрибутива Debian, что значительно упрощает задачу поиска старых версий пакетов.

В табл. 3 приводится список ROP-эксплойтов, которые были успешно проанализированы реализованным инструментом анализа ROP-цепочек. Время анализа не превосходило пары секунд.

*Табл. 3. Список проанализированных ROP-эксплойтов*
*Table 3. Analyzed ROP exploits*

| Приложение | Номер CVE | Платформа | Гаджеты из |
|---|---|---|---|
| MongoDB | CVE-2013-1892 | Linux x86 | mongod |
| Nagios3 | CVE-2012-6096 | Linux x86 | history.cgi |
| ProFTPd | CVE-2010-4221 | Linux x86 | proftpd |
| Nginx | CVE-2013-2028 | Linux x64 | nginx |
| AbsoluteFTP | CVE-2011-5164 | Windows x86 | MFC42.dll |
| ComSndFTP | N/A 2012-06-08 | Windows x86 | msvcrt.dll |

## *7. Заключение*

В данной статье был предложен метод анализа атак повторного использования кода, который был реализован в виде программного инструмента. Разработанный метод позволяет упростить для аналитика задачу обратной инженерии ROP-эксплойтов. По бинарной ROP-цепочке восстанавливается список вызванных гаджетов и описывается их функциональность семантически с помощью булевого предиката, который должен быть всегда истинным после выполнения гаджета. Более того, восстанавливаются прототипы и значения аргументов вызванных функций и системных вызовов. Таким образом, аналитик может автоматизированно получить представление о семантике ROP-цепочки. Реализованный метод был апробирован на реальных ROP-эксплойтах, найденных в интернете.

Метод основывается на динамической интерпретации промежуточного представления инструкций ROP-цепочки. Семантика гаджета определяется в результате анализа эффектов выполнения гаджета на различных случайных входных данных. Для восстановления значений аргументов функций и системных вызовов во время анализа поддерживается теневая память.

Перспективным направлением для дальнейших работ является поддержка анализа условных переходов и циклов в ROP-цепочках. А для повышения точности определения семантики гаджета можно использовать известные техники формальной верификации. Также технической задачей является поддержка анализа ROP-цепочек, использующих гаджеты сразу из нескольких исполняемых файлов.

## Список литературы

[1]. Common Vulnerabilities and Exposures (CVE). Available at: https://cve.mitre.org, accepted 10.11.2008.
[2]. Статистика уязвимостей (CVE) по годам. Available at: https://www.cvedetails.com/browse-by-date.php, accepted 10.11.2008.
[3]. CWE-121: Stack-based Buffer Overflow. Available at: https://cwe.mitre.org/data/definitions/121.html, accepted 10.11.2008.
[4]. Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In Proc. of the 14th ACM Conference on Computer and Communications Security, CCS'07, 2007, pp. 552–561.
[5]. Schwartz E.J, Avgerinos T., Brumley D. Q: Exploit Hardening Made Easy. In Proc. of the 20th USENIX Conference on Security, SEC'11, 2011, p. 25.
[6]. Jager I., Brumley D. Efficient Directionless Weakest Preconditions. Technical Report CMU-CyLab-10-002, 2010.
[7]. Lu K., Zou D., Wen W., Gao D. deRop: Removing Return-oriented Programming from Malware. In Proc. of the 27th Annual Computer Security Applications Conference, ACSAC'11, 2011, pp. 363–372.
[8]. Graziano M., Balzarotti D., Zidouemba A. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In Proc. of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS'16, 2016, pp. 47–58.
[9]. Roemer R., Buchanan E., Shacham H., Savage S. Return-Oriented Programming: Systems, Languages, and Applications. ACM Transactions on Information and System Security, vol. 15, no. 1, 2012, pp. 2:1–2:34.
[10]. Инструмент IDA Pro. Available at: https://www.hex-rays.com/products/ida/, accepted 10.11.2008.
[11]. Nethercote N., Seward J. How to Shadow Every Byte of Memory Used by a Program. In Proc. of the 3rd International Conference on Virtual Execution Environments, VEE'07, 2007, pp. 65–74.
[12]. Moser A., Kruegel C., Kirda E. Exploring Multiple Execution Paths for Malware Analysis. In Proc. of the 2007 IEEE Symposium on Security and Privacy, SP'07, 2007, pp. 231–245.
[13]. Вишняков А.В. Классификация ROP гаджетов. Труды ИСП РАН, том 28, вып. 6, 2016 г., стр. 27–36. DOI: 10.15514/ISPRAS-2016-28(6)-2
[14]. VirtualProtect function (Windows). Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx, accepted 10.11.2008.
[15]. The Linux man-pages project. Available at: https://www.kernel.org/doc/man-pages/.
[16]. API Monitor: Spy on API Calls and COM Interfaces. Available at: http://www.rohitab.com/apimonitor, accepted 10.11.2008.
[17]. Падарян В.А., Соловьев М.А., Кононов А.И. Моделирование операционной семантики машинных инструкций. Труды ИСП РАН, том 19, 2010 г., стр. 165–186.
[18]. Metasploit Framework. Available at: https://github.com/rapid7/metasploit-framework, accepted 10.11.2008.
[19]. Exploit Database. Available at: https://www.exploit-db.com, accepted 10.11.2008.
[20]. snapshot.debian.org. Available at: http://snapshot.debian.org, accepted 10.11.2008.

# Method for analysis of code-reuse attacks

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

**Abstract.** Providing security for computer programs is one of the paramount tasks nowadays. Failures in operation of program software can lead to serious consequences and exploitation of vulnerabilities can inflict immense harm. Large corporations pay particular attention to the analysis of computer security incidents. Code-reuse attacks based on return-oriented programming are gaining more and more popularity each year and can bypass even modern operating system protections. Unlike common shellcode, where instructions are placed consequently in memory, ROP chain contains of several small instruction blocks (gadgets) and uses stack to chain them together, which makes analysis of ROP exploits more difficult. The main goal of this work is to simplify reverse engineering of ROP exploits. In this paper I propose the method for analysis of code-reuse attacks, which allows one to split chain into gadgets, restore the semantics of each particular gadget, and restore prototypes and parameters values of system calls and functions called during the execution of ROP chain. Parametrized types define gadget semantics. Each gadget type is defined by a postcondition (boolean predicate) that must always be true after executing the gadget. The proposed method was implemented as a program tool and tested on real ROP exploits found on the internet.

**Keywords:** computer security; binary analysis; vulnerability; return-oriented programming; ROP; gadgets classification; code-reuse attack; computer security incident.

**DOI:** 10.15514/ISPRAS-2018-30(5)-2

**For citation:** Vishnyakov A.V., Nurmukhametov A.R., Kurmangaleev Sh.F., Gaisaryan S.S. Method for analysis of code-reuse attacks. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 31-54 (in Russian). DOI: 10.15514/ISPRAS-2018-30(5)-2

## References

[1]. Common Vulnerabilities and Exposures (CVE). Режим доступа: https://cve.mitre.org, дата обращения 10.11.2008.
[2]. Статистика уязвимостей (CVE) по годам. Режим доступа: https://www.cvedetails.com/browse-by-date.php, дата обращения 10.11.2008.

[3]. CWE-121: Stack-based Buffer Overflow. Режим доступа: https://cwe.mitre.org/data/definitions/121.html, дата обращения 10.11.2008.
[4]. Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In Proc. of the 14th ACM Conference on Computer and Communications Security, CCS'07, 2007, pp. 552–561.
[5]. Schwartz E.J, Avgerinos T., Brumley D. Q: Exploit Hardening Made Easy. In Proc. of the 20th USENIX Conference on Security, SEC'11, 2011, p. 25.
[6]. Jager I., Brumley D. Efficient Directionless Weakest Preconditions. Technical Report CMU-CyLab-10-002, 2010.
[7]. Lu K., Zou D., Wen W., Gao D. deRop: Removing Return-oriented Programming from Malware. In Proc. of the 27th Annual Computer Security Applications Conference, ACSAC'11, 2011, pp. 363–372.
[8]. Graziano M., Balzarotti D., Zidouemba A. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In Proc. of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS'16, 2016, pp. 47–58.
[9]. Roemer R., Buchanan E., Shacham H., Savage S. Return-Oriented Programming: Systems, Languages, and Applications. ACM Transactions on Information and System Security, vol. 15, no. 1, 2012, pp. 2:1–2:34.
[10]. Инструмент IDA Pro. Режим доступа: https://www.hex-rays.com/products/ida/, дата обращения 10.11.2008.
[11]. Nethercote N., Seward J. How to Shadow Every Byte of Memory Used by a Program. In Proc. of the 3rd International Conference on Virtual Execution Environments, VEE'07, 2007, pp. 65–74.
[12]. Moser A., Kruegel C., Kirda E. Exploring Multiple Execution Paths for Malware Analysis. In Proc. of the 2007 IEEE Symposium on Security and Privacy, SP'07, 2007, pp. 231–245.
[13]. Vishnyakov A.V. Classification of ROP gadgets. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 27–36 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-2
[14]. VirtualProtect function (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx.
[15]. The Linux man-pages project. Режим доступа: https://www.kernel.org/doc/man-pages/.
[16]. API Monitor: Spy on API Calls and COM Interfaces. Режим доступа: http://www.rohitab.com/apimonitor, дата обращения 10.11.2008.
[17]. Padaryan V.A., Soloviev M.A., Kononov A.I. Modeling operational semantics of machine instructions. Trudy ISP RAN/Proc. ISP RAS, 2010, vol. 19, pp. 165–186 (in Russian).
[18]. Metasploit Framework. Режим доступа: https://github.com/rapid7/metasploit-framework, дата обращения 10.11.2008.
[19]. Exploit Database. Режим доступа: https://www.exploit-db.com, дата обращения 10.11.2008.
[20]. snapshot.debian.org. Режим доступа: http://snapshot.debian.org, дата обращения 10.11.2008.