Gray Hat Python

Автор: Justin Seitz

Перевод:
reverse4you.org,
M.Chumichev

Редактор:
ks_ks

Введение

Я изучил Python конкретно для хакинга - и я осмелюсь сказать, что это утверждение правдиво для многих других так же. Я провел достаточно много времени в изучении языка, который хорошо приспособлен для хакинга и реверс инженерии, и несколько лет назад стало весьма очевидно, что Python становится настоящим лидером среди языков ориентированных на хакинг. Однако хитрость была в том, что не было стоящего руководства по теме, как использовать Python для различных задач хакинга. Вам приходится копаться в форумах и мануалах, и обычно проводить достаточно много времени времени пошагово просматривать код, чтобы заставить его работать правильно. Эта книга нацелена на заполнение этого разрыва путем предоставления вам беглого курса как использовать Python для хакинга и реверс-инженерии различными способами.

Книга составлена так, что позволит вам изучить некоторые теоретические основы большинства средств и техник хакинга, включающих дебаггеры, бэкдоры, фаззеры, эмуляторы, и инъекции кода, обеспечивая вам некоторое представление о том, как готовые инструменты Python могут быть использованы, когда не требуются обычные решения. Вы изучите не только как использовать инструменты, основанные на Python, но и как создавать инструменты на языке Python. Но предупреждаем, это не исчерпывающее руководство! Существует много-много инструментов для ИБ (информационной безопасности), написанных на Python, которые я не рассматривал. Однако, эта книга позволит вам освоить много подобных навыков по применению приложений, которые вы сможете использовать, отлаживать, расширять, и настраивать любое Python-приложение по вашему выбору.

Есть несколько способов изучения этой книги. Если вы новичок в Python или в разработке инструментов для хакинга, то вам стоит читать книгу от начала до конца по порядку. Вы изучите немного необходимой теории, запрограммируете кучу кода на Python, и получите твчрдые знания о том, как решить множество задач хакинга и реверсинга по прочтению книги. Если вы уже знакомы с Python и хорошо понимаете библиотеку ctypes Python, то переходите сразу к Главе 2. Для тех из вас, кто "в теме вполне достаточно переходить к нужным разделам книги и использовать фрагменты кода или определенные разделы, как вам надо в ваших повседневных задачах.

Я потратил много времени на отладчики, начиная с теории отладки в Главе 2, и продолжая прямо до Immunity Debugger (модификация OllyDbg прим.) в Главе 5. Отладчики это важные инструменты для любого хакера, и я не стесняюсь рассказывать вам о них достаточно подробно. Двигаясь дальше, вы узнаете некоторые техники перехвата (hooking) и инъекций в Главах 6 и 7, которые вы можете добавить в некоторые концепции отладки управления программой и манипулирования памятью.

Следующий раздел книги нацелен на взлом приложений используя фаззеры (fuzzers). В Главе 8 вы начнете изучать фаззинг (fuzzing), и создадите свой простейший файловый фаззер. В Главе 9 мы будем использовать мощный Sulley fuzzing framework чтобы сломать настоящий FTP-демон, и в Главе 10 вы узнаете как создать фаззер для взлома драйверов Windows.

В Главе 11, вы увидите, как автоматизировать статические задачи аналитики в IDA Pro, популярного средства для бинарного статического анализа. Мы завершим книгу темой PyEmu, основанного на Python эмулятора компьютера, в Главе 12.

Я постарался представить исходные коды несколько меньше, с детальными пояснениями о том, как работает код, вставленный в определенных точках. Часть времени при изучении нового языка, или изготовлении новых библиотек, проводится в необходимом усердном переписывании кода и отладки совершенных вами ошибок. Я поощряю ваш ручной ввод кода. Все исходные коды к вашему удовольствию представлены на официальном сайте книги.

Ну а теперь приступим к программированию!

Оглавление

Глава 1

Настройка рабочего окружения

Intro

Прежде чем вы начнете совершенствоваться в искусстве программирования в Серой Шляпе на языке Python, вам следует пройти самую неинтересную часть этой книги - настройки вашей будущей рабочей среды разработки. Весьма важно, чтобы вы имели хорошую и удобную среду разработки, которая позволит вам провести время в поглощении крайне интересной информации в данной книге, а не спотыкаться и набивать шишки, пытаясь заставить ваш код выполняться.

Эта глава быстро покрывает тему установки и настройки Python 2.5 (на момент перевода версия Python уже 2.6, как будет работать с 3 версией без понятия, но всҷ равно для всех примеров в книге будет использоваться Python именно версии 2.5. прим. пер.), конфигурирования вашего рабочего окружения Eclipse, и основы написания Си-совместимого кода на Python. Как только вы настроите окружение и поймете основы, мир будет для вас устрицей, а эта книга покажет вам, как еҷ открыть.

1.1 Требования к системе

Я предполагаю, что вы используете 32-битную ОС, основанную на базе Windows (XP, Vista, 7). Windows имеет широкий спектр инструментов и хорошо поддается для программирования на Python. Все главы этой книги ориентированны в первую очередь на Windows, и большинство примеров будут работать только с ОС Windows.

Однако, будет несколько примеров, которые вы можете запустить на дистрибутиве Linux. Для разработки под Linux я рекомендую вам скачать 32-битный дистрибутив Linux'а как VMware устройства. Проигрыватель VMware устройств бесплатен, и позволяет вам быстро перемещать файлы с вашей рабочей системы на виртуальную Linux машину. Если у вас завалялся лишний компьютер, можете попробовать свои силы и установить полноценную Linux систему. Для целей книги использовался основанный на Red Hat Linux дистрибутив, вроде Fedora Core 7 или Centos 5. Конечно, в качестве альтернативы, вы можете запустить Linux и сэмулировать на нчм Windows. Это дело вашего вкуса.

Бесплатные образы для VMware

На сайте VMware есть каталог бесплатных устройств. Эти устройства позволяют обратному разработчику (реверс инженеру) или исследователю уязвимостей разместить вредоносную программу (malware) или приложения на виртуальной машине, сокращая к минимуму риски для физической инфраструктуры и предоставляют изолированный "черновик"для работы. Вы можете посетить страницу виртуальных устройств по адресу и скачать плеер по адресу.

1.2 Получение и установка Python 2.5

Среда Python устанавливается быстро и безболезненно как на Linux так и на Windows. Пользователям Windows облегчит жизнь установщик, который позаботится обо всех настройках, однако на Linux вы будете собирать и устанавливать Python из исходных кодов (Однако на очень многих дистрибутивах Linux Python 2.5 установлен "из коробки").

1.2.1 Установка Python в Windows

Пользователи Windows могут получить установщик с официального сайта Python. Только дважды щелкните мышкой по файлу установщика и следуйте далее шаг за шагом по этапам установки. Установщик создаст каталог 'C:25. В этом каталоге будут установлены файл python.exe - интерпретатор команд Python, а так же все стандартные библиотеки. Примечание: Вместо этого вы можете установить отладчик Immunity Debugger, который содержит не только сам отладчик, но и установщик Python 2.5. В последних главах книги вы будете использовать Immunity Debugger для многих задач, поэтому вы можете "убить двух зайцев"одной

установкой. Чтобы скачать и установить Immunity Debugger, посетите http://debugger.immunityinc.com

1.2.2 Установка Python в Linux

Для установки Python 2.5 в ОС Linux, вам следует скачать и скомпилировать исходные коды. Это дает вам полный контроль над настройками в процессе установки Python в ОС на базе Red Hat. Процесс установки подразумевает, что вы будете выполнять все следующие команды от имени пользователя root(суперпользователя).

Первым шагом будет загрузка и распаковка исходного кода Python 2.5. В командном терминале (консоли) вводите следующее: Код:

```
# cd /usr/local
# wget http://python.org/ftp/pyton/2.5.1/Python-2.5.1.tgz
# tar -zxvf Python-2.5.1.tgz
# mv Python-2.5.1 Python25
# cd Python25
```

Вы только что скачали и распаковали исходный код в '/usr/local/Python25'. Следующим шагом будет компиляция исходного кода и проверки работоспособности интерпретатора Python: Код:

```
# ./configure --prefix=/usr/local/Python25
# make && make install
# pwd
/usr/local/Python25
#python
Python 2.5.1 (r251:54863, Mar 14 2012, 07:39:18)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on Linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Теперь вы находитесь в интерактивной оболочке Python, которая предоставляет вам полный доступ к интерпретатору Python и любой встроенной библиотеке. Быстрая проверка покажет, что команды интерпретируются верно:
Код:

```
>>> print "Hello World!"
Hello World!
>>> exit()
#
```

Отлично! Все работает, как вам надо. Чтобы застраховаться, что ваше пользовательское окружение находит, где находится интерпретатор Pyton, автоматически, вам следует изменить файл '/root/.bashrc'. Лично я использую текстовый редактор nano для всех моих работ с текстом, но вы вольны выбрать любой текстовый редактор, с которым вам комфортно работать. Откройте файл '/root/.bashrc' и в конце файла добавьте следующую строку: Код:

```
export PATH=/urs/local/Pyton25/:$PATH
```

Эта строка укажет окружению Linux что пользователь root может иметь доступ к интерпретатору Python без указания полного пути к нему. Если вы закончите сессию root и затем войдете под root снова, когда вы введете python в любом месте в командной строке, вы незамедлительно окажетесь в интерпретаторе Python. Примечание переводчика: В Windows можно сделать то же самое. Для этого нужно щелкнуть правой кнопкой мыши на "Мой компьютер> "Свойства> "Дополнительно> "Переменные среды> Выбрать переменную "Path> "Изменить> В строке "Значение переменной"в конце дописать ;C:25. Далее проверяем в командной строке. Вводим python и смотрим, запустился ли интерпретатор, как в случае с Linux или нет.

И так, теперь у вас полностью рабочие интерпретаторы Python как в Windows так и в Linux. Настало время настроить вашу интегрированную среду разработки (IDE). Если у вас есть IDE в которой вам комфортно работать, то можете пропустить следующий раздел.

1.3 Настройка среды Eclipse и PyDev

Как правило для быстрой разработки и отладки Python приложений, абсолютно необходимо использовать полноценную IDE. Взаимодействие популярной среды разработки Eclipse и модуля PyDev к нему дает вам в руки огромное число мощных возможностей, которые не предлагают большинство других средств разработки. Кроме того, Eclipse запускается одинаково в Windows, Linux, и Mac, а так же имеет отличную группу поддержки и разработки. Давайте быстро пройдем все этапы как установить и настроить Eclipse и PyDev: Скачиваете архив Eclipse Classic для вашей платформы с сайта. Распаковываете его в "C:". Запускаете C:.exe. При первом запуске Eclipse спросит вас где хранить ваше рабочее

место, вы можете принять предлагаемое место по умолчанию, и поставить галочку в графу "Use this as default and do not ask again". Щелкните на OK. Как только Eclipse запустится выберите "Help => Install new Software..."Переходите к полю Work with:. Кликните на кнопку Add... В поле Name введите описывающую строку, вроде PyDev Update. Убедитесь, что поле URL содержит http://pydev.org/updates и щ̧лкните OK. Затем щелкните Finish, и вас выбросит в установщик обновлений Eclipse. Диалог обновлений появится через несколько мгновений. Когда он появится, выберите PyDev и нажмите Next для продолжения. Затем прочитайте и примите лицензионное соглашение для PyDev. Если вы согласны с ним, то выберите "I accept the terms in the license agreement". Щелкните Next и наконец Finish. Вы увидите, как Eclipse установит PyDev дополнение. В конце кликните Yes в диалоговом меню, которое появится после установки PyDev. Это перезагрузит Eclipse, и запустит его с вашим новеньким PyDev.

На следующей стадии конфигурирования Eclipse вы убедитесь, что PyDev может найти правильный интерпретатор Python для последующего использования, когда вы будете запускать скрипты в PyDev: Когда запустится Eclipse, выберите "Window => Preferences"; Разверните ветвь PyDev, и выберите Interpreter - Python; Нажмите кнопку New...; Укажите путь в Browse: "C:25.exe"и кликните Open; Следующее диалоговое окно покажет вам список включенных библиотек для интерпретатора. Оставьте все как есть и кликните OK; Затем кликните OK еще раз, чтобы закончить настройку интерпретатора.

Теперь у вас есть установленный и работоспособный PyDev, настроенный для использования свежеустановленного интерпретатора Python 2.5. Прежде чем начнете программировать, вам следует создать новый PyDev проект. Этот проект будет содержать все исходные файлы, данные далее в этой книге. Чтобы настроить новый проект, следуйте следующим образом: Выберите "File => New => Project"; Разверните ветку PyDev, и выберите PyDev Project. Кликните Next для продолжения; Назовите "Gray Hat Python". Щ̧лкните Finish.

Вы заметите, что экран Eclipse перегруппирует себя, и вы увидите проект Gray Hat Python проект в наверху слева. Теперь щелкните правой кнопкой мыши папу src и выберите "New => PyDev Module". В графе Name введите chapter1-test, и кликните на Finish. Вы увидите, что панель вашего проекта обновится, и в этот список будет добавлен файл chapter1-test.py.

Чтобы запустить скрипт Python в среде Eclipse, только кликните кнопку Run As (зеленый кружок с белой стрелкой внутри) на панели инструментов. Чтобы заново запустить последний запущенный скрипт нажмите Ctrl+F11. Когда вы запустите скрипт в Eclipse, вместо командной строки Windows, вы увидите панель внизу среды Eclipse, обозначенной, как Console. Весь вывод ваших скриптов будет отображаться на панели Console. Вы так же можете обратить внимание, что текстовый редактор уже открыл chapter1-test.py и уже ожидает немного сладкого нектара Python.

1.3.1 Лучшие друзья Хакера: ctypes

Модуль ctypes для Python является безусловно одной из самых мощных библиотек, доступных разработчику на Python. Библиотека ctypes позволяет вам вызывать функции в динамически подключаемых библиотеках (dll) и имеет богатые возможности для создания комплексных типов данных языка Си и полезных функций для низкоуровневой работы с памятью. Это необходимо для того, чтобы вы понимали основы того, как использовать библиотеку ctypes, так как вы будете полагаться на эту библиотеку в значительной степени на протяжении всей книги.

1.3.2 Использование динамических библиотек

Первый шаг на пути использования ctypes это понимание, как запускать и получать доступ к функциям в динамически подключаемых библиотеках. Динамически подключаемая библиотека (dynamically linked library) - это скомпилированный бинарный файл, который подключаются к основному процессу во время его выполнения по мере надобности. На платформе Windows эти бинарные файлы называются динамически подключаемые библиотеки, а на платформе Linux - разделяемые объекты (eng. shared objects(SO)). В обоих случаях эти бинарные файлы предоставляют функции через экспортированные имена, какие получают возможность запуска по реальным адресам в памяти. Обычно во время выполнения вам приходится решать, какой будет порядок адресов функций для вызова этих функций, однако с ctypes вся эта грязная работа уже сделана.

Есть три различных способа загрузки dll в ctypes: cdll(), windll(), и oledll(). Разница между ними в том, каким образом вызываются функции этих библиотек, и какие они возвращают значения. Метод cdll() используется для запуска библиотек, которые экспортируют функции, используя стандартное соглашение вызова cdecl. Метод windll() загружает библиотеки, которые экспортируют функции, используя соглашение вызова stdcall, которое является родным соглашением в Microsoft Win32 API. Ну а метод oledll() работает так же, как windll(), однако, он предполагает, что экспортированная функция возвращает Windows код ошибки HRESULT, который используется специально для сообщений об ошибках, возвращаемых функциями Объектной Модели Компонентов (MS Component Object Model, COM).

В качестве небольшого примера, вы возьмете функцию printf() из библиотеки времени исполнения языка C (C runtime) в Windows и Linux, и используете еч для вывода в тестовом сообщении. В Windows C runtime находится в msvcrt.dll, находящемся в папке 'C:32; а в Linux это libc.so.6, которая по умолчанию находится в '/lib/'. Создайте следующий скрипт Python или в Eclipse, или в вашей обычной рабочей директории Python, и введите следующий код:
Код:

```
# chapter1-printf.py: Код для Windows
from ctypes import *

msvcrt = cdll.msvcrt
message_string = "Hello world!\n"
msvcrt.printf("Testing: %s", message_string)
```

Этот скрипт выводит следующее:
Код:

```
C:\Python25\python chapter1-printf.py
Testing: Hello world!
C:\Python25>
```

Под Linux этот пример будет немного отличаться, но делать будет то же самое. Перейдите в Linux и создайте chapter1-printf.py в вашей директории '/root/'

Понимание соглашений вызовов

Соглашение вызовов описывает как правильно вызывать определенную функцию. Оно включает в себя порядок того, как выделяются параметры функции, какие параметры функции помещаются в стек или переходят в регистры, и как раскручивается стек при возвращении функцией значения. Вам надо научиться понимать два соглашения вызовов: cdecl и stdcall. В соглашении cdecl параметры помещаются в стек справа налево, и вызывающий функцию ответственен за очистку стека от аргументов. Это соглашение используется в большинстве C-систем на архитектуре x86.

Вот пример вызова функции с помощью cdecl:

На языке C:
Код:

```
int python_rocks(reason_one, reason_two, reason_tree);
```

На ассемблере x86:
Код:

```
    push reason_tree
    push reason_two
    push reason_one
    call python_rocks
    add esp, 12
```

Вы можете ясно увидеть, как передаются аргументы функции. Последняя линия выделяет (увеличивает) стеку 12 байт (у функции 3 параметра, и каждый параметр стека занимает 4 байта, поэтому в сумме дает 12 бит), которых как раз хватает для этих параметров.

Пример соглашения stdcall, которые используются в Win32 API демонстрируется тут:
На языке C:
Код:

```
int my_socks(color_one, color_two, color_tree);
На языке ассемблера x86:
Код:
    push color_tree
    push color_two
    push color_one
    call my_socks
```

В данном случае вы видите, что порядок параметров такой же, но очистка стека не проводится вызывающим функцию, а функцией перед возвращением ей полученных значений.

Для обоих соглашений важно запомнить, что возвращенные значения функции хранятся в регистре EAX

```
# chapter1-printf.py: пример для Linux
from ctypes import *

libc = CDLL("libc.so.6")
message_string = "Hello world!\n"
libc.printf("Testing: %s", message_string)
```

Данный скрипт в Linux будет выводить следующее:
Код:

```
# python /root/chapter1-printf.py
Testing: Hello world!
#
```

Это легко уметь вызывать функции в динамической библиотеке и использовать функции, которые она экспортирует. Вы будете использовать эту технику много раз на протяжении всей книги, поэтому очень важно, чтобы поняли, как это работает.

1.3.3 Конструирование типов данных C

Создание типа данных языка C в Python - это словно откровенная сексуальность в этом гиковском, странном способе (Дословно: Creating a C datatype in Python is just downright sexy, in that nerdy, weird way. кто предложит перевод лучше, будет молодцом). Наличие этих особенностей позволяет вам полностью использовать компоненты, написанные на C и C++, которые значительно увеличивают мощь языка Python. Кратко просмотрите таблицу 1-1, для понимания того, как взаимосвязаны типы данных C и Python и результирующий тип ctypes.

Видите, как хорошо типы данных конвертируются туда и обратно? Используйте эту таблицу для удобства, в случае, если вы забудете преобразование типов. Типы из ctypes могут быть инициализированы со значением, но оно должно быть надлежащего типа и размера. Для демонстрации, откройте вашу оболочку Python и введите несколько следующих примеров:
код:

```
C:\Python25>python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
Type "help", "copyright", "credits" or "license" for more information.
```

C Type	Python Type	ctypes Type
char	1-character string	c_char
wchar_t	1-character Unicode string	c_wchar
char	int/long	c_byte
char	int/long	c_ubyte
short	int/long	c_short
unsigned short	int/long	c_ushort
int	int/long	C_int
unsigned int	int/long	c_uint
long	int/long	c_long
unsigned long	int/long	c_ulong
long long	int/long	c_longlong
unsigned long long	int/long	c_ulonglong
float	float	c_float
double	float	c_double
char * (NULL terminated)	string or none	c_char_p
wchar_t * (NULL terminated)	unicode or none	c_wchar_p
void *	int/long or none	c_void_p

Рис. 1.1: Python to C Datatype Mapping

```
>>> from ctypes import *
>>> c_int()
c_long(0)
>>> c_char_p("Hello, world!")
c_char_p('Hello, world!')
>>> c_ushort(-5)
c_ushort(65531)
>>>
>>> seitz = c_char_p("loves the python")
>>> print seitz
c_char_p('loves the python')
>>> print seitz.value
loves the python
>>> exit()
```

Последний пример описывает вам как переменной seitz присваивается указатель на строку "loves the python". Для доступа к содержимому указателя используется метод seitz.value, который называется разыменовыванием указателя.

1.3.4 Передача параметров по ссылке

Очень часто в C и C++ есть функции, которые ожидают указатель, как один из параметров. Причина в том, что функция может как записывать в этот участок памяти, или, если при передаче по значению параметр слишком велик. В любом случае, ctypes обладает функционалом чтобы сделать именно это, используя функцию byref(). Когда функция ожидает указатель, в качестве параметра, вы вызываете еҷ так: fuction$_m ain(byref(parametr))$.

1.3.5 Определение структур и объединений

Структуры и объединения - важные типы данных, которые часто используются как Win32 API, так и с libc в Linux. Структура - это простая группа переменных, которые могут быть одного или разных типов данных. Вы можете получить доступ к любой содержащейся в структуре переменной, используя точечную нотацию (dot notation), например такую: beer_recipe.amt_barley. Это даст доступ к переменной amt_barley, находящейся в структуре beer_recipe. Следующий пример определит струк-

туру (или struct как их обычно называют для упрощения) как в C так и в Python.
Код на C:
Код:

```
structure beer_recipe
{
    int amt_barley;
    int amt_water;
};
```

Код на Python:
Код:

```
class beer_recipe(Structure):
    _fields_ = [
    ("amt_barley",c_int),
    ("amt_water",c_int),
    ]
```

Как видите, ctypes делает очень легким создание C-совместимой структуры. Замечу, что это не полный рецепт пива, и я не советую вам пить смесь ячменя и воды.

Объединения, это почти тоже самое, что и структуры. Однако в объединении все участвующие переменные находятся по одному адресу в памяти. Следующий пример демонстрирует объединение, которое позволяет вам отобразить число тремя разными способами.

Код на C:
Код:

```
union {
    long barley_long;
    int barley_int;
     char barley_char[8];
}barley_amount;
```

Код на Python:
Код:

```
class barley_amount(Union):
    _fields_=[
```

```
    ("barley_long", c_long),
    ("barley_int", c_int),
    ("barley_char", c_char * 8),
    ]
```

Если вы присвоите переменной barley_int из объединения barley_amount значение 66, то затем вы сможете использовать переменную barley_char для отображения символа представляющего это число. Для демонстрации создадим новый файл chapter1-unions.py и вобъем следующий код.
Код:

```
# chapter1-unions.py
from ctypes import *

class barley_amount(Union):
    _fields_=[
    ("barley_long", c_long),
    ("barley_int", c_int),
    ("barley_char", c_char * 8),
    ]
value = raw_input("Enter the amount of barley to put into the beer vat:")
my\_barley = barley_amount(int(value))
print "Barley amount as a long: %ld" %my_barley.barley_long
print "Barley amount as an int: %d" %my_barley.barley_long
print "Barley amount as a char: %s" %my_barley.barley_char
```

Вывод будет следующим:
Код:

```
C:\Python25> python chapter1-unions.py
Enter the amount of barley to put into the beer vat:66
Barley amount as a long: 66
Barley amount as an int: 66
Barley amount as a char: B

C:\Python25>
```

Как видите, присвоив объединению одно значение, вы можете получить три разных представления этого значения. Если вас смущает вывод значения переменной barley$_char, 'B', ASCII()$66.

Переменная barley_char из этого объединения - отличный пример того, как определять массив в ctypes. В ctypes массив определяется путем умножения типа на количество элементов, которые вы хотите выделить

в массиве. В предыдущем примере, 8-элементный массив символов был определен для переменной barley_char из объединения.

Теперь у вас есть рабочее окружение Python на двух отдельных системах, и теперь вы понимаете как взаимодействовать с низкоуровневыми библиотеками. Настало время для того, чтобы начать применять эти знания для создания широкого списка инструментов для помощи в реверс-инженерии и хакинге программ. Надеваем шлем.

Глава 2

Отладчики и устройство отладчика

Intro

Отладчик - глазное яблоко хакера. Отладчики позволяют вам выполнять трассировку (отслеживание) выполнения процесса, или проводить динамический анализ. Возможность выполнения динамического анализа абсолютно необходима, когда речь заходит о создании эксплойтов, поддержки фазеров и проверки вредоносного ПО. Это очень важно, чтобы вы понимали что такое отладчик, и принцип его работы. Отладчики предоставляют целое множество конкретных расширений и функционала, которое очень полезно при оценке ошибок в программах. Большинство из них предоставляют возможность запускать, останавливать, или выполнять пошагово процесс, устанавливать точки останова, манипулировать регистрами и памятью, и отлавливать случающиеся исключения в исследуемом процессе.

Но прежде чем мы двинемся дальше, давайте обсудим разницу между отладчиком белого ящика(white-box debugger) и отладчиком черного(black-box debugger). Большинство платформ разработки или IDE, содержат встроенный отладчик, который позволяет разработчикам отслеживать процесс выполнения программы, имея исходный код, и контролируя очень многое. Это называется отладкой белого ящика. Эти отладчики полезны во время разработки ПО, но реверс-инженер, или охотник за багами, редко имеет доступ к исходным кодам, и ему приходится использовать отладчики черного ящика, во время пошагового выполнения программы (трассировки). Отладчик черного ящика предполагает, что исследуемая

программа полностью непрозрачна для хакера, и единственная доступная информация - это дизассемблированный код. При этом методе нахождения ошибок, более сложном и затратном по времени, хорошо подготовленный реверс-инженер в состоянии понять устройство программы на очень высоком уровне. Иногда ребята, ломающие программы, могут получить более глубокие знания и понимание того, как работает программа, нежели разработчик ее создавший!

Очень важно различать два подкласса отладчиков черного ящика: уровня пользователя и уровня ядра. Уровень пользователя (как правило т.н. ring 3 (кольцо третьего уровня, прим.)) - это режим процессора, в котором запускаются ваши пользовательские приложения. Приложения уровня пользователя выполняются с наименьшим количеством привилегий. Когда вы запускаете calc.exe чтобы что-то посчитать, вы порождаете процесс на уровне пользователя; если вы будете пошагово выполнять (тресировать) этот процесс, это будет отладка на уровне пользователя. Уровень ядра (ring 0) - это наибольший уровень привилегий. Это уровень, где работает ядро операционной системы вместе с драйверами и другими низкоуровневыми компонентами. Когда вы анализируете сетевой трафик (сниффаете, от sniffing - "нюхать прим.) с помощью Wireshark, вы взаимодействуете с драйвером, который работает на уровне ядра. Если вы хотите остановить драйвер, и исследовать его состояние в любой точке, то вам понадобится отладчик уровня ядра.

Сейчас я приведу вам короткий список отладчиков уровня ядра, часто используемых реверс-инженерами и хакерами: WinDbg от Microsoft и OllyDbg, бесплатный отладчик от Oleh Yuschuk. При отладке под Linux, обычно используют GNU Debugger (gdb). Все три отладчика достаточно мощны, и каждый предлагает такие возможности, которые не доступны другим отладчикам.

Однако в последние годы стала проявляться тенденция в интеллектуальной отладке, особенно на платформе Windows. Интеллектуальный отладчик поддерживает скрипты (сценарии), поддерживает расширенные возможности, например такие, как вызов перехвата, и что самое главное, имеют много возможностей используемых для охоты на баги(bug hunting) и реверс-инженеринга. Два новых лидера в этой сфере: PyDbg от Pedram Amini и Immunity Debugger от Immunity inc.

2.1 Регистры общего назначения центрального процессора

Регистр - это небольшой объем памяти находящийся прямо на центральном процессоре, и доступ к нему - быстрейший метод для процессора, чтобы получить данные. В наборе инструкций архитектуры x86 используются восемь регистров общего назначения: EAX, EDX, ECX, ESI, EDI, EBP, ESP и EBX. Большинство регистров доступны процессору, но мы рассмотрим их только в конкретных обстоятельствах, когда они потребуются. Каждый из восьми регистров общего назначения разработан для своей конкретной работы, и каждый выполняет свою функцию, которая позволяет процессору эффективно выполнять инструкции. Это очень важно - понимать, какой регистр для чего используется, ибо это знание положит фундамент понимания того, как устроен отладчик. Давайте пройдемся по каждому регистру и его функциям. Мы закончим выполнением простым упражнением реверс-инженерии, для иллюстрации их использования.

Регистр EAX, так же называемый регистром аккумуляции (или аккумулятором), используется для выполнения расчетов, а так же для хранения значений возвращаемых вызванными функциями. Многие оптимизированные инструкции в наборе инструкций x86 разработаны для перемещения данных именно в регистр EAX и извлечения данных из него, а так же для выполнения расчетов с этими данными. Большинство простых операций, таких как сложение, вычитание и сравнение оптимизированы для использования регистра EAX. Кроме того, многие определенные операции, такие как умножение или деление, могут выполняться только в регистре EAX.

Как было замечено ранее, возвращенные значения из вызываемых функций хранятся в EAX. Это важно запомнить, что вы можете легко определить, если не удалось вызвать функцию или наоборот удалось в зависимости от значения находящегося в EAX. Кроме того, вы можете определить актуальное значение, которое возвращает функция.

Регистр EDX это регистр данных (data register). Этот регистр в основном является дополнительным для регистра EAX, и он помогает хранить дополнительные данные для более сложных вычислений, таких как умножение и деление. Он так же может быть хранилищем данных общего назначения, но обычно он используется в расчетах, выполненных в

сочетании с регистром EAX.

Регистр ECX так же называется регистром-счетчиком (count register), он используется в операциях цикла. Часто повторяющиеся операции стоит хранить в упорядоченной пронумерованной строке. Очень важно понимать, что счетчик регистра ECX уменьшает, а не увеличивает значение. Продемонстрируем это на примере отрывка, написанного на Python: Код:

```
counter = 0

while counter < 10:
    print "Loop number: %d" % counter
    counter +=1
```

Если мы переведем этот код в язык ассемблера, то в первом цикле значение в ECX будет равно 10, 9 в следующем цикле и так далее. Вас может немного смутить, что это противоречит тому, что написано в листинге на Python, но просто запомните, что он уменьшается, и все будет хорошо.

В ассемблере x86 циклы, которые обрабатывают данные, полагаются на регистры ESI и EDI для продуктивной манипуляции данными. Регистр ESI это индекс источника (source index) данных операции, и хранит адрес входящего потока данных. Регистр EDI указывает на место, где находится результат обработанных данных, поэтому он называется индексом назначения(приемника) (destination index). Это легко запомнить таким образом: регистр ESI используется для чтения данных, а EDI для записи. Использование регистров индексов источника и приемника значительно увеличивает производительность выполняемой программы.

Регистры ESP и EBP соответственно указатель стека (stack pointer) и указатель базы(base pointer). Эти регистры используются для управления вызовами функций и операциями со стеком. Когда функция вызвана, аргументы функции перемещаются (проталкиваются) в стек и следуют по адресу возврата. Регистр ESP указывает на самый верх стека, поэтому он будет указывать на адрес возврата. Регистр EBP указывает на самый низ стека вызовов. В некоторых случаях компилятор может использовать оптимизацию для удаления регистра EBP как указателя кадра, в этих случаях регистр EBP освобождается и может использоваться точно так же, как любой другой регистр общего назначения.

Единственный регистр, который не был разработан для чего-то конкретного - это EBX. Он может использоваться, как дополнительное хранилище данных.

Единственный дополнительный регистр, который стоит упомянуть отдельно, это регистр EIP. Он указывает на инструкцию, которая выполняется в данный момент. Как процессор проходит по двоичному исполняемому коду, EIP обновляется для отображения адреса, по которому в данный момент происходит выполнение.

Отладчик должен уметь легко читать и изменять содержимое этих регистров. Каждая операционная система предоставляет интерфейс отладчикам для взаимодействия с процессором, возможности извлекать и модифицировать данные в регистрах. Мы рассмотрим отдельные интерфейсы в главах о конкретных операционных системах.

2.2 Стек

Стек - это очень важная структура, которую надо знать при разработке отладчика. Стек хранит информацию о том, как вызывается функция, какие параметры она забирает, и что надо вернуть после выполнения функции. Структура стека представляет собой модель "Первый пришел, последний вышел"(FILO, First In, Last Out), когда аргументы проталкиваются в стек для вызова функции, и извлекаются из стека после того, как функция завершит свое выполнение. Регистр ESP используется для отслеживания самой вершины кадра стека, а регистр EBP используется для отслеживания самого низа стека. Стек "растет"от верхних адресов памяти к нижним адресам памяти. Давайте используем нашу предыдущую рассмотренную функцию my_socks(), как простейший пример того, как работает стек.

На языке C:
Код:

```
int my_socks(color_one, color_two, color_three);
```

Вызов функции в ассемблере x86:
Код:

```
push color_three
```

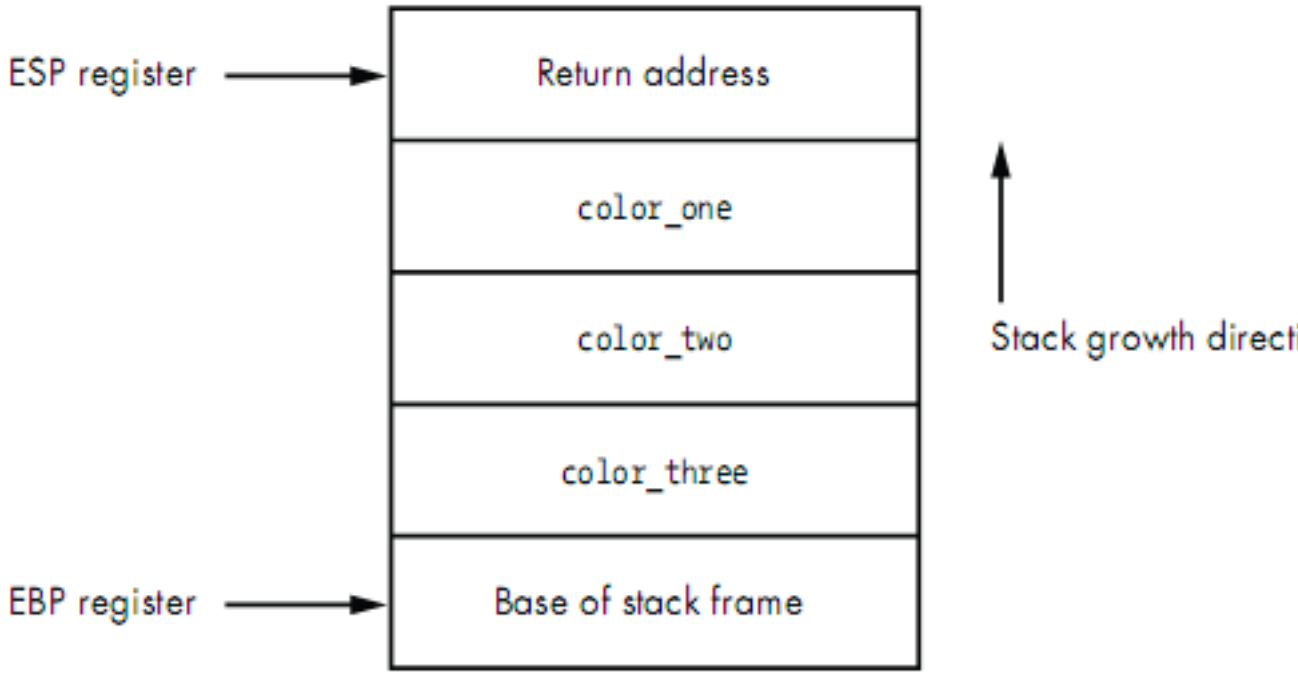


Рис. 2.1: Кадр стека для вызова функции my_socks()

```
push color_two
push color_one
call my_socks
```

Так будет выглядеть стек изображено на Рисунке 2-1.
Перевод обозначений:

Stack growth direciton - направление роста стека. Base of stack frame - основание (база) кадра стека. Return adress - адрес возвращаемого значения.

Как вы можете видеть, это прямая структура данных и это основа для всех вызываемых функций в двоичном коде. Когда функция my_socks() завершает работу (и возвращает какое-либо значение), она выталкивает из стека все значения и переходит к адресу возврата для продолжения выполнения родительской функции, которая еҷ вызвала. Рассмотрим понятие локальных переменных. Локальные переменные это часть памяти, которая доступна только для функции, которая выполняется в данный момент. Немного расширим функцию my_socks(), давайте предположим,

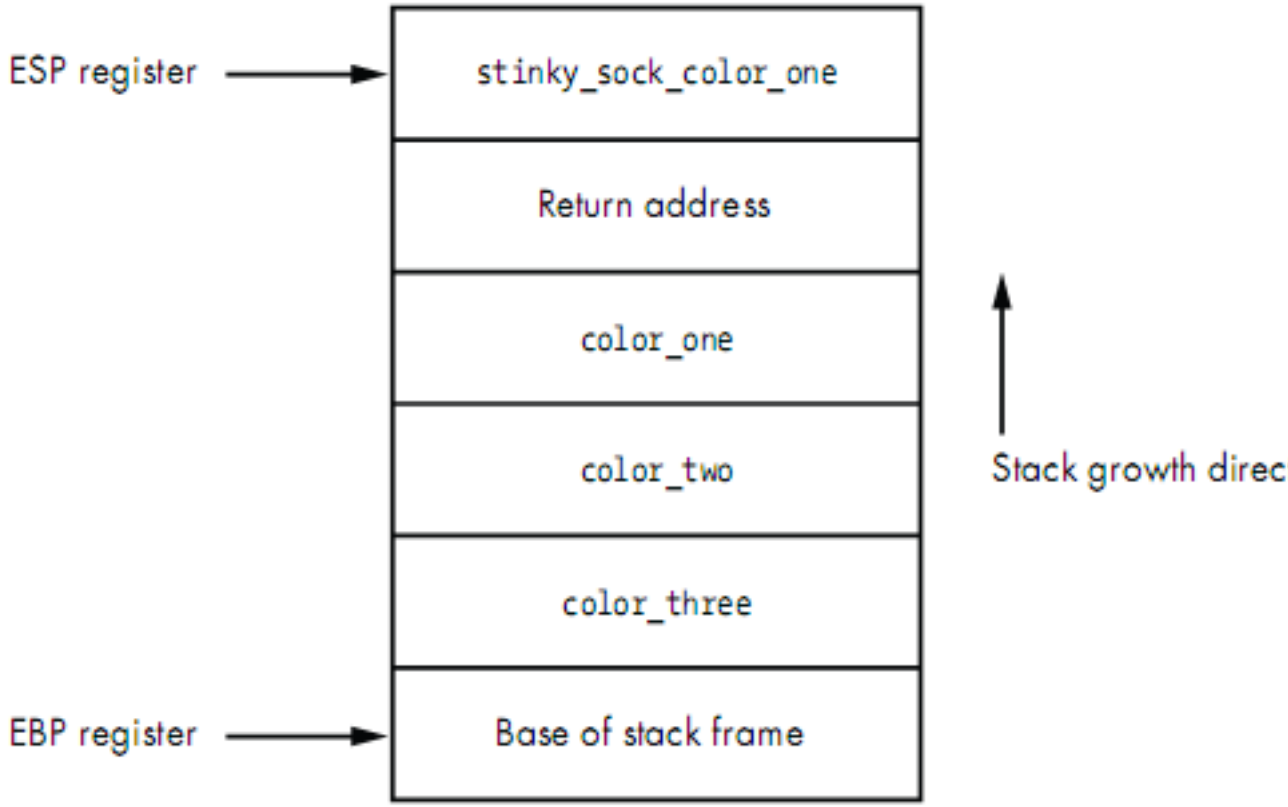


Рис. 2.2: Кадр стека после того, как была объявлена локальная переменная stinky_sock_color_one.

что первое, что будет делать эта функция - это создавать массив символов, в который будет скопирован параметр color_one. Код будет выглядеть так:
Код:

```
int my_socks(color_one, color_two, color_three)
{
char stinky_sock_color_one[10];
...
}
```

Переменная stinky_sock_color_one будет находится в стеке так, что ее можно использовать в конкретном кадре стека. Как только произошло это распределение, кадр стека будет выглядеть как на Рисунке 2-2.

Теперь вы увидели, как локальная переменная размещается в стеке, и как указатель стека увеличивается вплоть до точки вершины стека. Способность захватить кадр стека в отладчике очень полезно для пошагового выполнения (трассировки) функций, захвата состояния стека при аварии программы, и отслеживания переполнений стека.

2.3 События отладчика

Отладчик работает как бесконечный цикл который ждет события отладки. Когда событие для отладки случается, цикл прерывается, и вызывается соответствующий обработчик событий.

Когда вызван обработчик событий, отладчик останавливается и ждет указаний, что ему делать дальше. Вот несколько обычных событий, которые должен улавливать отладчик:

Встреча точки останова (breakpoint) Нарушения памяти (так же называемые нарушениями доступа или нарушением сегментации) Исключения, сгенерированные отлаживаемой программой

У каждой операционной системы разный способ отправки этих событий отладчику, которые будут описаны в главах, посвященных операционным системам. В разных операционных системах могут быть перехвачены другие события, например такие, как создание потока и процесса, или загрузка динамической библиотеки во время выполнения. Мы рассмотрим эти конкретные события, когда понадобится.

Преимущество отладчика с возможностью создания сценариев (скриптов) - возможность построить собственные обработчики событий для автоматизации определенных задач отладки. Для примера, обычная причина переполнения буфера - нарушения целостности памяти, и это представляет большой интерес для хакера. В течение обычной сессии отладки, если случаются переполнение буфера и нарушения памяти, то вы должны перехватить отладчиком и в ручную получить информацию, которая вам интересна. Возможность создавать эти настроенные вами обработчики, не только сохраняют ваше время, но и так же позволяют вам расширить возможность управлять процессом отладки.

2.4 Точки останова

Возможность остановить отлаживаемый процесс достигается установкой точек останова (breakpoints). Остановив процесс, у вас появляется возможность проверить переменные, аргументы стека, и что находится в памяти без изменения переменных процесса, прежде чем вы сможете записать их. Точки останова - это определенно самая частая вещь, которую вы можете использовать при отладке процесса, и рассмотрим их очень внимательно. Есть три основных вида точек останова: программные (soft breakponit), аппаратные (hardware breakpoint), и памяти (memory breakpoint). Они ведут себя очень похоже, но выполняются очень разными способами.

2.4.1 Программные точки останова

Программные точки останова используются специально для остановки процессора при выполнении инструкций и это самый частый вид точек останова, который вы будете использовать при отладке приложений. Программный брейкпоинт - это однобитная интсрукция, которая останавливает выполнение отлаживаемого процесса и передает управление обработчику исключений точек останова. В целях понимания как это работает, вам следует знать разницу между инструкцией (instruction) и опкодом (opcode) в ассемблере x86.

Инструкция ассемблера - это высокоуровневое представление команды на выполнение для процессора. Пример:
Код:

```
MOV EAX, EBX
```

Эта инструкция говорит процессору переместить значение, хранящееся в регистре EBX в регистр EAX. Очень просто, правда? Однако, процессор не знает, как интерпретировать эту инструкцию, ему нужно конвертировать еч в что-то, называемое опкодом. Код операции (code operation) или опкод это команда машинного языка, которую выполняет процессор. Для иллюстрации, давайте переведем предыдущий пример инструкции в его "родной"опкод:
Код:

```
8BC3
```

Как вы можете видеть, он затемняет (обфусцирует) то, что реально происходит "за сценой но это язык, на котором говорит процессор. Думать о инструкциях ассемблера, как DNS (прим. Сервер имчн) для процесора. Инструкции упрощают возможность запомнить команды выполнения (имена сайтов, хостов), вместо запоминания всех индивидуальных опкодов (IP адреса). Вам очень редко понадобится использовать опкоды в вашей обыденной отладке, но они важны для понимания целей программных точек останова.

Если инструкция, которую мы рассматривали ранее, будет находится по адресу 0x44332211, общее представление будет выглядеть так:
Код:

```
0x44332211: 8BC3 MOV EAX, EBX
```

Это отображаются адрес, опкод, и высокоуровневая инструкция ассеблера. Для того, чтобы установить программный брейкпоинт по этому адресу и остановить процессор, мы заменим один байт из 2-байтного опкода 8BC3. Этот одиночный байт представляет собой инструкцию прерывания (interrupt) 3 (INT 3), которая "приказывает"процессору остановится. Инструкция INT3 конвертируется в однобайтный опкод 0xCC. Вот наш предыдущий пример до и после установки точки останова.

Опкод до установки точки останова:
Код:

```
0x44332211: 8BC3 MOV EAX, EBX
```

Модифицированный опкод после установки точки останова:
Код:

```
0x44332211: CCC3 MOV EAX, EBX
```

Вы можете видеть, что заменили байт 8B на байт CC. Когда процессор идет вприпрыжку (кроме шуток, comes skipping along) и натыкается на этот байт, он останавливается и запускает событие INT 3. Отладчики имеют встроенную возможность обрабатывать это событие, но прежде чем вы разработаете ваш собственный отладчик, надо хорошо понимать, как они делают это. Когда отладчик говорит установить точку останова в желаемый адрес, он сначала читает первый байт опкода по запрошенному адресу и сохраняет его. Затем отладчик записывает CC байт по этому адресу. Когда точка останова, или INT 3, срабатывает при интерпретации процессором опкода CC, отладчик перехватывает это. Затем

отладчик проверяет указывает ли указатель инструкций (регистр EIP) на адрес, на который предварительно была установлена точка останова. Если адрес находится во внутреннем списке точек останова, он записывает обратно сохраненный байт по этому адресу, чтобы опкод мог выполниться правильно, после продолжения выполнения процесса. Рисунок 2-3 детально описывает этот процесс.

Перевод обозначений:

1) Отладчик посылает инструкцию установить точку останова по адресу 0x44332211; он считывает и сохраняет первый байт. 2) Переписывает первый байт на опкод 0xCC (INT 3). 3) Когда процессор встречает точку останова, происходит внутренний поиск, и байт возвращается на место. Breakpoint List - список точек останова.

Как видите, отладчик должен словно станцевать, чтобы обработать программную точку останова. Есть два типа точек останова, которые вы можете установить: одноразовые (one-shot) брейкпоинты и стойкие (persistent) точки останова. Одноразовые точки останова подразумевают то, что точка останова встречается, она удаляется из внутреннего списка точек останова отладчика, они удобны для только одной установки. Стойкая точка останова восстанавливается после того, как процессор выполнит оригинальный опкод, поэтому запись в списке отладчика сохраняется.

Однако программные точки останова имеют одно ограничение, когда вы меняете байт исполняемого в памяти, вы изменяете контрольную сумму циклического избыточного кода (Cyclic redundancy code, CRC) выполняемого приложения. CRC - это тип функции, которая используется для определения изменения данных каким либо способом, и она может быть применена к файлам, памяти, тексту, сетевым пакетам, или чего-нибудь еще, за изменением данных которого вам надо наблюдать. CRC возьмет диапазон данных, в данном случае память выполняемого процесса, и получит хэш содержимого. Затем она сравнивает хеш с контрольной суммой для определения были ли изменены данные. Если контрольная сумма отличается от контрольной суммы, которая хранится для подтверждения, проверка CRC собьется. Важно заметить, как часто вредоносное ПО будет проверять свой исполняемый код в памяти для любых изменений CRC и убьчт себя, если обнаружится сбой. Это очень эффективная техника для замедления реверс-инженерии, таким образом предотвращается использование программных точек останова, ограничивая динамический анализ его поведения. Для того, чтобы обой-

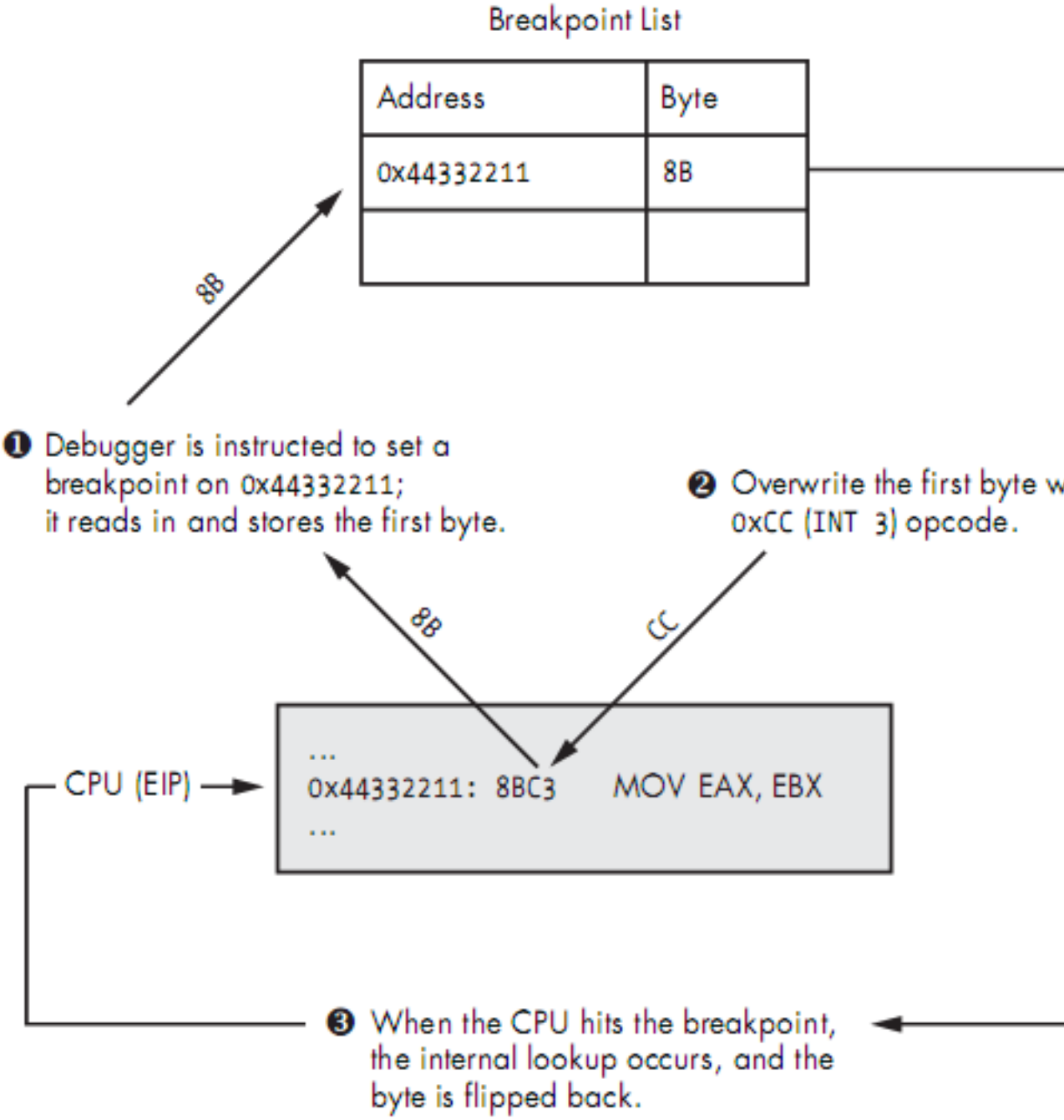


Рис. 2.3: Процесс установки программной точки останова.

ти эти особенности используются аппаратные точки останова.

2.4.2 Аппаратные точки останова

Аппаратные точки останова (hardware breakpoints) полезны, когда нужно установить небольшое число точек останова, и отлаживаемая программа не может быть модифицирована. Этот тип точек устанавливается на уровне процессора, в специальных регистрах, называемых регистрами отладки. Типичный процессор имеет 8 регистров отладки (по порядку с DR0 до DR7 соответственно), которые используются для установки и управлением аппаратных точек. Регистры отладки с DR0 до DR3 зарезервированы для адресов точек останова. Это означает, что вы можете использовать лишь 4 аппаратных точки одновременно. Регистры DR4 и DR5 зарезервированы, а регистр DR6 используется, как регистр статуса, который определяет тип события отладки, вызванного встречей точки останова. Регистр отладки DR7 по существу является выключателем (вкл/выкл) аппаратных точек останова, а так же хранит разные состояния точек останова. При установке специальных флагов в регистр DR7, вы можете создать точки останова в следующих состояниях:

Останов, когда инструкция выполняется по определенному адресу. Останов, когда данные записываются по адресу. Останов на чтение или запись, но не выполнение.

Это очень полезно, если у вас есть возможность выставить до четырех точек останова в конкретные состояния без модифицирования выполняемого процесса. Рисунок 2-4 покажет вам как поля в DR7 связанны с поведением аппаратных точек останова, длиной и адресом.

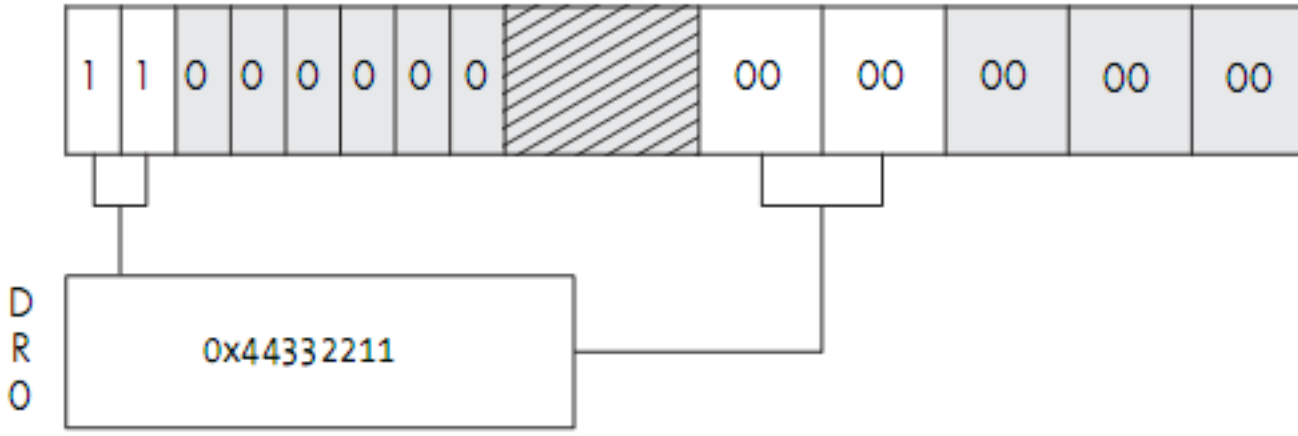
Биты 0-7 по существу переключатели (вкл/выкл) для активации точек останова. Поля L и G в битах 0-7 стоят для глобального и локального масштаба. Я изобразил оба бита, как установленные. Тем не менее, установка только одного из них будет работать, и на моем опыте у меня не было каких либо проблем в отладке на уровне пользователя. Биты 8-15 в DR7 не используются для нормальных целей отладки, которые мы будем осуществлять. Обратитесь к мануалу по архитектуре Intel x86 для дополнительного разъяснения назначения этих битов. Биты 16-31 определяют тип и длину точки останова, которая была установлена для связанного регистра отладки.

Перевод обозначений:

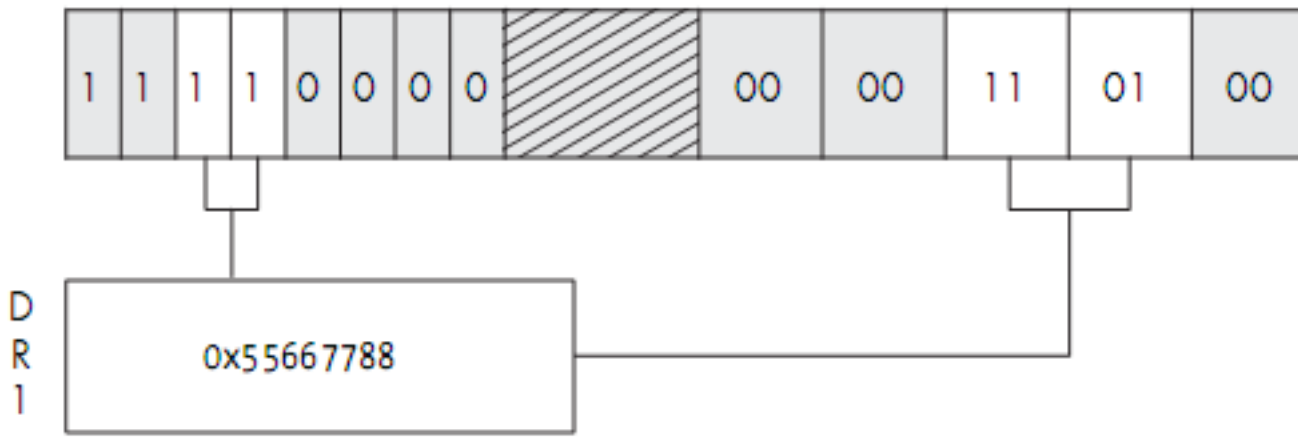
Layout of DR7 Register

	L DR0	G DR0	L DR1	G DR1	L DR2	G DR2	L DR3	G DR3		Type DR0	Len DR0	Type DR1	Len DR1	Type DR2
Bits	0	1	2	3	4	5	6	7	8 – 15	16 17	18 19	20 21	22 23	24 25

DR7 with 1-byte Execution Breakpoint Set at 0x44332211



DR7 with Additional 2-byte Read/Write Breakpoint at 0x5566



Breakpoint Flags Breakpoint L

Layout of DR7 register - вывод регистра DR7. Type - тип Len - сокр. длина. DR7 with 1-byte Execution Breakpoint Set at... - DR7 с точкой останова на исполнение 1 байта установленной по адресу... DR7 with Additional 2-byte Read/Write Breakpoint at... - DR7 с точкой останова на исполнение дополнительного 2-байтного чтения/записи по адресу... Breakpoint Flags - флаги точек останова Breakpoint Length Flags - фдаги длины точек останова Break on execution - останов, когда инструкция выполняется. Break on data writes - останов на запись данных. Break on read or write but not execution - останов на чтение или запись, но не выполнение.

В отличие от программных, которые используют событие INT3, аппаратные точки используют прерывание 1 (INT1). INT1 случается для аппаратных точек останова и одноступенчатых событий. Одноступенчатый означает проход по порядку по инструкциям, что позволяет вам очень близко исследовать критические участки кода во время изменения наблюдаемых данных.

Аппаратные точки останова обрабатываются таким же способом, как и программные, но их механизм находится на низком уровне. Прежде чем процессор попытается выполнить инструкцию, он сначала проверит, не установлена ли на адрес аппаратная точка. Он так же проверит операторов инструкции, не имеют ли они доступ к адресу, на который установлена аппаратная точка. Если адрес хранится в регистрах отладки DR0-DR3 и условия чтения, записи, или выполнения встречаются, прерывание INT1 приказывает процессору остановиться. Если адрес не хранится в регистрах отладки, процессор выполняет инструкцию и переходит к следующей, и эта проверка выполняется снова, и так далее.

Аппаратные точки очень полезны, но у них есть некоторые ограничения. Помимо того, что вы можете выставить только четыре ваших точки в одно время, вы так же можете установить точку только на данные длинной 2 байта. Это может сильно ограничить вас, если вы собираетесь получить доступ к большому участку памяти. Как правило, для обхода этих ограничений вы можете использовать точки останова памяти.

2.4.3 Точки останова памяти

Точки останова памяти являются не совсем точками останова. Когда отладчик устанавливает точку памяти, он меняет разрешения на регион,

или страницу (page) памяти. Страница памяти - это наименьшая порция памяти, которую обрабатывает операционная система. Когда страница памяти выделяется, у нее устанавливаются конкретные разрешения на доступ, которые указывают как эта память может быть доступна. Вот некоторые примеры разрешений на страницу памяти:

Выполнение страницы (page execution) Дает возможность выполнять, но выдает нарушение, если процесс попытается прочитать или записать данные на страницу. Чтение страницы (Page read) Позволяет процессу только считывать со страницы; любая запись или попытка выполнения вызовут нарушение доступа. Запись страницы (Page write) Позволяет процессу записывать на страницу. Сторожевая страница (Guard page) Любой доступ к сторожевой странице возвращает единовременное исключение, и затем страница возвращается к своему первоначальному статусу.

Большинство операционных систем позволяют вам комбинировать эти разрешения. Например, вы можете иметь страницу памяти, куда вы можете писать и читать, в то время, как другая страница может позволить вам прочитать или выполнить. Каждая операционная система так же имеет присущие функции, позволяющие вам запрашивать конкретные разрешения памяти в месте для особой страницы и изменять их, если понадобится. Посмотрите на Рисунок 2-5, чтобы увидеть, как доступ к данным работает с разными установленными разрешениями страницы памяти.

Главная интересующее нас разрешение для страницы - сторожевая страница. Этот тип страниц очень полезен для таких вещей, как отделение кучи от стека или убеждения в том, что часть памяти не разрастется за пределы ожидаемого диапазона. Она так же хороша для остановки процесса, когда он встречает особую часть памяти. Например, если мы реверс-инженируем сетевое серверное приложение, мы должны будем установить точку останова памяти на регион памяти, где хранится полезная нагрузка пакета, после его получения. Это позволит нам определить когда и как приложение использует содержимое полученного пакета, как и любой имеющий доступ к этой странице памяти будет останавливать процессор, кидая отладочное исключение сторожевой страницы. Затем мы можем проверить инструкцию, которая имеет доступ к буферу в памяти и определить, что происходит с содержимым. Эта техника точек останова так же работает с проблемами деформации данных, которые имеют программные точки останова, так как мы не изменяем исполняе-

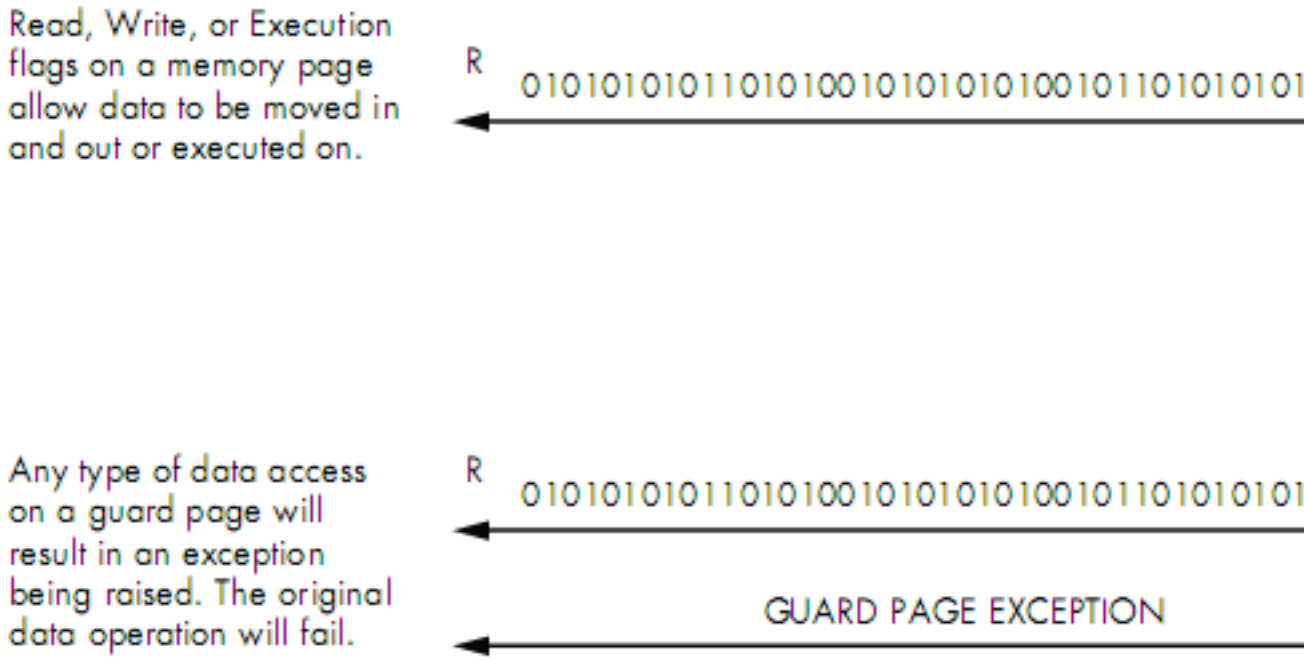


Рис. 2.5: Поведение разных разрешений страниц памяти.

мый код.

Click here to view the original image of 681x335px.
Перевод обозначений:

Read, Write, or Execute flags on a memory page allow data to be moved in and out or executed on - Флаги на чтение, запись или выполнение на странице памяти позволяют данным быть записанными и полученными или выполненным на странице. Any type of data access on a guard page will result in an exception being raised. The original data operation will fail. - Любой тип доступа к данным на сторожевой странице приведут к возникновению исключения. Оригинальная операция с данными не выполнится. GUARD PAGE EXCEPTION - исключение сторожевой страницы.

Наконец, мы рассмотрели некоторые простейшие аспекты того, как работает отладчик и как он взаимодействует с операционной системой, теперь пришло время написать наш первый легковесный отладчик на Python. Мы начнем с создание простенького отладчика в Windows, где накопленные знания о ctypes и внутренностях отладчика нам очень пригодятся. Давайте разогреем наши пальцы для кодинга.

Глава 3

Создание отладчика под Windows

Глава 4

PyDbg - Windows отладчик на чистом Python

Глава 5

Immunity Debugger

Глава 6

Hooking

Глава 7

DLL и Code Injection

Глава 8

Fuzzing

Глава 9

Sulley

Глава 10

Фаззинг драйверов Windows

Глава 11

IDAPython

Глава 12

PyEmu