

Посвящается Ноортъе и Сиетсе

ОГЛАВЛЕНИЕ

Вступительное слово	17
Предисловие	20
Благодарности.....	21
Введение.....	22

ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

Глава 1. Анатомия двоичного файла.....	32
Глава 2. Формат ELF	52
Глава 3. Формат PE: краткое введение	78
Глава 4. Создание двоичного загрузчика с применением libbfd	88

ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

Глава 5. Основы анализа двоичных файлов в Linux.....	109
Глава 6. Основы дизассемблирования и анализа двоичных файлов	135
Глава 7. Простые методы внедрения кода для формата ELF.....	178

ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

Глава 8. Настройка дизассемблирования.....	212
Глава 9. Оснащение двоичных файлов.....	244
Глава 10. Принципы динамического анализа заражения	289
Глава 11. Практический динамический анализ заражения с помощью libdf	305
Глава 12. Принципы символического выполнения.....	335
Глава 13. Практическое символическое выполнение с помощью Triton.....	361

ЧАСТЬ IV. ПРИЛОЖЕНИЯ

Приложение А. Краткий курс ассемблера x86	402
Приложение В. Реализация перезаписи PT_NOTE с помощью libelf.....	422
Приложение С. Перечень инструментов анализа двоичных файлов	443
Приложение Д. Литература для дополнительного чтения.....	447
Предметный указатель	451

СОДЕРЖАНИЕ

Вступительное слово	17
Предисловие	20
Благодарности	21
Введение	22

ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

Глава 1. Анатомия двоичного файла	32
1.1 Процесс компиляции программы на C.....	33
1.1.1 Этап препроцессирования	33
1.1.2 Этап компиляции	35
1.1.3 Этап ассемблирования	37
1.1.4 Этап компоновки	38
1.2 Символы и зачищенные двоичные файлы.....	40
1.2.1 Просмотр информации о символах.....	40
1.2.2 Переход на темную сторону: зачистка двоичного файла.....	42
1.3 Дизассемблирование двоичного файла	42
1.3.1 Заглянем внутрь объектного файла.....	43
1.3.2 Изучение полного исполняемого двоичного файла.....	45
1.4 Загрузка и выполнение двоичного файла	48
1.5 Резюме	50
Глава 2. Формат ELF	52
2.1 Заголовок исполняемого файла	54
2.1.1 Массив e_ident	55
2.1.2 Поля e_type, e_machine и e_version	56

2.1.3	Поле e_entry	57
2.1.4	Поля e_phoff и e_shoff	57
2.1.5	Поле e_flags	57
2.1.6	Поле e_ehsize	58
2.1.7	Поля e_entsize и e_num	58
2.1.8	Поле e_shstrndx	58
2.2	Заголовки секций	59
2.2.1	Поле sh_name	60
2.2.2	Поле sh_type	60
2.2.3	Поле sh_flags	61
2.2.4	Поля sh_addr, sh_offset и sh_size	61
2.2.5	Поле sh_link	62
2.2.6	Поле sh_info	62
2.2.7	Поле sh_addralign	62
2.2.8	Поле sh_entsize	62
2.3	Секции	62
2.3.1	Секции .init и .fini	64
2.3.2	Секция .text	64
2.3.3	Секции .bss, .data и .rodata	66
2.3.4	Позднее связывание и секции .plt, .got, .got.plt	66
2.3.5	Секции .rel.* и .rela.*	70
2.3.6	Секция .dynamic	71
2.3.7	Секции .init_array и .fini_array	72
2.3.8	Секции .shstrtab, .symtab, .strtab, .dynsym и .dynstr	73
2.4	Заголовки программы	74
2.4.1	Поле p_type	75
2.4.2	Поле p_flags	76
2.4.3	Поля p_offset, p_vaddr, p_paddr, p_filesz и p_memsz	76
2.4.4	Поле p_align	76
2.5	Резюме	77

Глава 3. Формат PE: краткое введение..... 78

3.1	Заголовки MS-DOS и заглушка MS-DOS	79
3.2	Сигнатура PE, заголовки PE-файла и факультативный заголовок PE	79
3.2.1	Сигнатура PE	82
3.2.2	Заголовок PE-файла	82
3.2.3	Факультативный заголовок PE	83
3.3	Таблица заголовков секций	83
3.4	Секции	84
3.4.1	Секции .edata и .idata	85
3.4.2	Заполнение в секциях кода PE	86
3.5	Резюме	86

Глава 4. Создание двоичного загрузчика с применением libbfd..... 88

4.1	Что такое libbfd?	89
4.2	Простой интерфейс загрузки двоичных файлов	89

4.2.1	Класс Binary.....	92
4.2.2	Класс Section.....	92
4.2.3	Класс Symbol.....	92
4.3	Реализация загрузчика двоичных файлов.....	93
4.3.1	Инициализация libbfd и открытие двоичного файла.....	94
4.3.2	Разбор основных свойств двоичного файла.....	96
4.3.3	Загрузка символов.....	99
4.3.4	Загрузка секций.....	102
4.4	Тестирование загрузчика двоичных файлов.....	104
4.5	Резюме.....	106

ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

Глава 5. Основы анализа двоичных файлов в Linux.....	109	
5.1	Разрешение кризиса самоопределения с помощью file.....	110
5.2	Использование ldd для изучения зависимостей.....	113
5.3	Просмотр содержимого файла с помощью xxd.....	115
5.4	Разбор выделенного заголовка ELF с помощью readelf.....	117
5.5	Разбор символов с помощью nm.....	119
5.6	Поиск зацепок с помощью strings.....	122
5.7	Трассировка системных и библиотечных вызовов с помощью strace и ltrace.....	125
5.8	Изучение поведения на уровне команд с помощью objdump.....	129
5.9	Получение буфера динамической строки с помощью gdb.....	131
5.10	Резюме.....	134

Глава 6. Основы дизассемблирования и анализа двоичных файлов.....	135	
6.1	Статическое дизассемблирование.....	136
6.1.1	Линейное дизассемблирование.....	136
6.1.2	Рекурсивное дизассемблирование.....	139
6.2	Динамическое дизассемблирование.....	142
6.2.1	Пример: трассировка выполнения двоичного файла в gdb.....	143
6.2.2	Стратегии покрытия кода.....	146
6.3	Структурирование дизассемблированного кода и данных.....	150
6.3.1	Структурирование кода.....	151
6.3.2	Структурирование данных.....	158
6.3.3	Декомпиляция.....	160
6.3.4	Промежуточные представления.....	162
6.4	Фундаментальные методы анализа.....	164
6.4.1	Свойства двоичного анализа.....	164
6.4.2	Анализ потока управления.....	169
6.4.3	Анализ потока данных.....	171
6.5	Влияние настроек компилятора на результат дизассемблирования.....	175
6.6	Резюме.....	177

Глава 7. Простые методы внедрения кода для формата ELF178

7.1	Прямая модификация двоичного файла с помощью шестнадцатеричного редактирования	178
7.1.1	Ошибка на единицу в действии.....	179
7.1.2	Исправление ошибки на единицу.....	182
7.2	Модификация поведения разделяемой библиотеки с помощью LD_PRELOAD	186
7.2.1	Уязвимость, вызванная переполнением кучи	186
7.2.2	Обнаружение переполнения кучи.....	189
7.3	Внедрение секции кода	192
7.3.1	Внедрение секции в ELF-файл: общий обзор.....	192
7.3.2	Использование elfinject для внедрения секции в ELF-файл	195
7.4	Вызов внедренного кода	198
7.4.1	Модификация точки входа.....	199
7.4.2	Перехват конструкторов и деструкторов.....	202
7.4.3	Перехват записей GOT	205
7.4.4	Перехват записей PLT	208
7.4.5	Перенаправление прямых и косвенных вызовов	209
7.5	Резюме	210

ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

Глава 8. Настройка дизассемблирования.....212

8.1	Зачем писать специальный проход дизассемблера?	213
8.1.1	Пример специального дизассемблирования: обфусцированный код	213
8.1.2	Другие причины для написания специального дизассемблера.....	216
8.2	Введение в Capstone	217
8.2.1	Установка Capstone	218
8.2.2	Линейное дизассемблирование с помощью Capstone	219
8.2.3	Изучение Capstone C API.....	224
8.2.4	Рекурсивное дизассемблирование с помощью Capstone	225
8.3	Реализация сканера ROP-гаджетов	234
8.3.1	Введение в возвратно-ориентированное программирование	234
8.3.2	Поиск ROP-гаджетов.....	236
8.4	Резюме	242

Глава 9. Оснащение двоичных файлов244

9.1	Что такое оснащение двоичного файла?.....	244
9.1.1	API оснащения двоичных файлов	245
9.1.2	Статическое и динамическое оснащение двоичных файлов	246
9.2	Статическое оснащение двоичных файлов	248
9.2.1	Подход на основе int 3.....	248
9.2.2	Подход на основе трамплинов	250

9.3	Динамическое оснащение двоичных файлов.....	255
9.3.1	Архитектура DBI-системы.....	255
9.3.2	Введение в Pin.....	257
9.4	Профилирование с помощью Pin.....	259
9.4.1	Структуры данных профилировщика и код инициализации ...	259
9.4.2	Разбор символов функций.....	262
9.4.3	Оснащение простых блоков.....	264
9.4.4	Оснащение команд управления потоком.....	266
9.4.5	Подсчет команд, передач управления и системных вызовов ...	269
9.4.6	Тестирование профилировщика.....	270
9.5	Автоматическая распаковка двоичного файла с помощью Pin.....	274
9.5.1	Введение в упаковщики исполняемых файлов.....	274
9.5.2	Структуры данных и код инициализации распаковщика.....	276
9.5.3	Оснащение команд записи в память.....	278
9.5.4	Оснащение команд управления потоком.....	280
9.5.5	Отслеживание операций записи в память.....	280
9.5.6	Обнаружение оригинальной точки входа и запись распакованного двоичного файла.....	281
9.5.7	Тестирование распаковщика.....	283
9.6	Резюме.....	287

Глава 10. Принципы динамического анализа заражения.....

10.1	Что такое DTA?.....	290
10.2	Три шага DTA: источники заражения, приемники заражения и распространение заражения.....	290
10.2.1	Определение источников заражения.....	291
10.2.2	Определение приемников заражения.....	291
10.2.3	Прослеживание распространения заражения.....	292
10.3	Использование DTA для обнаружения дефекта Heartbleed.....	292
10.3.1	Краткий обзор уязвимости Heartbleed.....	292
10.3.2	Обнаружение Heartbleed с помощью заражения.....	294
10.4	Факторы проектирования DTA: гранулярность, цвета политики заражения.....	296
10.4.1	Гранулярность заражения.....	296
10.4.2	Цвета заражения.....	297
10.4.3	Политики распространения заражения.....	298
10.4.4	Сверхзаражение и недозаражение.....	300
10.4.5	Зависимости по управлению.....	300
10.4.6	Теневая память.....	302
10.5	Резюме.....	304

Глава 11. Практический динамический анализ заражения с помощью libdft.....

11.1	Введение в libdft.....	305
11.1.1	Внутреннее устройство libdft.....	306
11.1.2	Политика заражения.....	309

11.2	Использование DTA для обнаружения удаленного перехвата управления	310
11.2.1	Проверка информации о заражении	313
11.2.2	Источники заражения: заражение принятых байтов	315
11.2.3	Приемники заражения: проверка аргументов exesvc	317
11.2.4	Обнаружение попытки перехвата потока управления	318
11.3	Обход DTA с помощью неявных потоков	323
11.4	Детектор утечки данных на основе DTA	324
11.4.1	Источники заражения: прослеживание заражения для открытых файлов	327
11.4.2	Приемники заражения: мониторинг отправки по сети на предмет утечки данных	330
11.4.3	Обнаружение потенциальной утечки данных	331
11.5	Резюме	334

Глава 12. Принципы символического выполнения.....335

12.1	Краткий обзор символического выполнения	336
12.1.1	Символическое и конкретное выполнение	336
12.1.2	Варианты и ограничения символического выполнения	340
12.1.3	Повышение масштабируемости символического выполнения	347
12.2	Удовлетворение ограничений с помощью Z3	349
12.2.1	Доказательство достижимости команды	350
12.2.2	Доказательство недостижимости команды	354
12.2.3	Доказательство общезначимости формулы	354
12.2.4	Упрощение выражений	356
12.2.5	Моделирование ограничений для машинного кода с битовыми векторами	356
12.2.6	Решение непроницаемого предиката над битовыми векторами	358
12.3	Резюме	359

Глава 13. Практическое символическое выполнение с помощью Triton.....361

13.1	Введение в Triton	362
13.2	Представление символического состояния абстрактными синтаксическими деревьями	363
13.3	Обратное нарезание с помощью Triton	365
13.3.1	Заголовочные файлы и конфигурирование Triton	368
13.3.2	Конфигурационный файл символического выполнения	369
13.3.3	Эмуляция команд	370
13.3.4	Инициализация архитектуры Triton	372
13.3.5	Вычисления обратного среза	373
13.4	Использование Triton для увеличения покрытия кода	374
13.4.1	Создание символических переменных	376
13.4.2	Нахождение модели для нового пути	377

13.4.3	Тестирование инструмента покрытия кода.....	380
13.5	Автоматическая эксплуатация уязвимости	384
13.5.1	Уязвимая программа	385
13.5.2	Нахождение адреса уязвимой команды вызова	388
13.5.3	Построение генератора эксплойта	390
13.5.4	Получение оболочки с правами root	397
13.6	Резюме	400

ЧАСТЬ IV. ПРИЛОЖЕНИЯ

Приложение А. Краткий курс ассемблера x86.....	402
A.1 Структура ассемблерной программы.....	403
A.1.1 Ассемблерные команды, директивы, метки и комментарии....	404
A.1.2 Разделение данных и кода	405
A.1.3 Синтаксис AT&T и Intel	405
A.2 Структура команды x86	405
A.2.1 Ассемблерное представление команд x86	406
A.2.2 Структура команд x86 на машинном уровне	406
A.2.3 Регистровые операнды	407
A.2.4 Операнды в памяти	409
A.2.5 Непосредственные операнды	410
A.3 Употребительные команды x86.....	410
A.3.1 Сравнение операндов и установки флагов состояния	411
A.3.2 Реализация системных вызовов	412
A.3.3 Реализация условных переходов	412
A.3.4 Загрузка адресов памяти	413
A.4 Представление типичных программных конструкций на языке ассемблера	413
A.4.1 Стек.....	413
A.4.2 Вызовы функций и кадры функций	414
A.4.3 Условные предложения.....	419
A.4.4 Циклы.....	420

Приложение В. Реализация перезаписи PT_NOTE с помощью libelf.....	422
V.1 Обязательные заголовки.....	423
V.2 Структуры данных, используемые в elfinject	423
V.3 Инициализация libelf	425
V.4 Получение заголовка исполняемого файла.....	428
V.5 Нахождение сегмента PT_NOTE	429
V.6 Внедрение байтов кода.....	430
V.7 Выравнивание адреса загрузки внедренной секции	431
V.8 Перезапись заголовка секции .note.ABI-tag	432
V.9 Задание имени внедренной секции	437
V.10 Перезапись заголовка программы PT_NOTE.....	439
V.11 Модификация точки входа	441

Приложение С. Перечень инструментов анализа двоичных файлов	443
C.1 Дизассемблеры	443
C.2 Отладчики.....	445
C.3 Каркасы дизассемблирования	445
C.4 Каркасы двоичного анализа.....	446
Приложение D. Литература для дополнительного чтения.....	447
D.1 Стандарты и справочные руководства	447
D.2 Статьи	448
D.3 Книги.....	450
Предметный указатель.....	451

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Дэннис Эндриесс имеет степень доктора по безопасности систем и сетей, и анализ двоичных файлов – неотъемлемая часть его исследований. Он один из основных соразработчиков системы целостности потока управления PathArmor, которая защищает от атак с перехватом потока управления типа возвратно-ориентированного программирования (ROP). Также Эндриесс принимал участие в разработке атаки, которая положила конец P2P-сети ботов GameOver Zeus.

О технических рецензентах

Торстен Хольц (Thorsten Holz) – профессор факультета электротехники и информационных технологий в Рурском университете в Бохуме, Германия. Он занимается исследованиями в области технических аспектов безопасности систем. В настоящее время его интересуют обратная разработка, автоматизированное обнаружение уязвимостей и изучение последних векторов атак.

Tim Vidas – хакер-ас. На протяжении многих лет он возглавлял инфраструктурную команду в соревновании DARPA CGC, внедрял инновации в компании Dell Secureworks и курировал исследовательскую группу CERT в области компьютерно-технической экспертизы. Он получил степень доктора в университете Карнеги-Меллона, является частым участником и докладчиком на конференциях и обладает числом Эрдёша–Бейкона 4-3. Много времени уделяет обязанностям отца и мужа.

ВСТУПИТЕЛЬНОЕ СЛОВО

Внаши дни нетрудно найти книги по языку ассемблера и даже описания двоичных форматов ELF и PE. Растет количество статей, посвященных прослеживанию потока информации и символическому выполнению. Но нет еще ни одной книги, которая вела бы читателя от основ ассемблера к выполнению сложного анализа двоичного кода. Нет ни одной книги, которая научила бы читателя оснащать двоичные программы инструментальными средствами, применять динамический анализ заражения (taint analysis) для прослеживания путей прохождений интересных данных по программе или использовать символическое выполнение в целях автоматизированного генерирования эксплойтов. Иными словами, нет книг, которые учили бы методам, инструментам и образу мыслей, необходимым для анализа двоичного кода. Точнее, до сих пор не было.

Трудность анализа двоичного кода состоит в том, что нужно разбираться в куче разных вещей. Да, конечно, нужно знать язык ассемблера, но, кроме того, понимать форматы двоичных файлов, механизмы компоновки и загрузки, принципы статического и динамического анализов, расположение программы в памяти, соглашения, поддерживаемые компиляторами, – и это только начало. Для конкретных задач анализа или оснащения могут понадобиться и более специальные знания. Конечно, для всего этого нужны инструменты. Многих эта перспектива настолько пугает, что они сдаются, даже не вступив в борьбу. Так много всего предстоит учить. С чего же начать?

Ответ: с этой книги. Здесь все необходимое излагается последовательно, логично и доступно. И интересно к тому же! Даже если вы ничего не знаете о том, как выглядит двоичная программа, как она

загружается и что происходит во время ее выполнения, в книге вы найдете отлично продуманное введение во все эти понятия и инструменты, с помощью которых сможете не только быстро освоить теоретические основы, но и поэкспериментировать в реальных ситуациях. На мой взгляд, это единственный способ приобрести глубокие знания, которые не забудутся на следующий день.

Но и в том случае, если у вас есть богатый опыт анализа двоичного кода с помощью таких инструментов, как Capstone, Radare, IDA Pro, OllyDbg или что там у вас стоит на первом месте, вы найдете здесь материал по душе. В поздних главах описываются продвинутые методики создания таких изощренных инструментов анализа и оснащения, о существовании которых вы даже не подозревали.

Анализ и оснащение двоичного кода – увлекательные, но трудные техники, которыми в совершенстве владеют лишь немногие опытные хакеры. Но внимание к вопросам безопасности растет, а вместе с ним и важность этих вопросов. Мы должны уметь анализировать вредоносные программы, чтобы понимать, что они делают и как им в этом воспрепятствовать. Но поскольку все больше вредоносных программ применяют методы обфускации и приемы противодействия анализу, нам необходимы более хитроумные методы.

Мы также все чаще анализируем и оснащаем вполне благопристойные программы, например, чтобы затруднить атаки на них. Так, может возникнуть желание модифицировать существующий двоичный код программы, написанной на C++, с целью гарантировать, что все вызовы (виртуальных) функций обращаются только к допустимым методам. Для этого мы должны сначала проанализировать двоичный код и найти в нем методы и вызовы функций. Затем нужно добавить оснащение, сохранив при этом семантику оригинальной программы. Все это проще сказать, чем сделать.

Многие начинают изучать такие методы, столкнувшись с интересной задачей, оказавшейся не по зубам. Это может быть что угодно: как превратить игровую консоль в компьютер общего назначения, как взломать какую-то программу или понять, как работает вредонос, проникший в ваш компьютер.

К своему стыду, должен сознаться, что лично для меня все началось с желания снять защиту от копирования с видеоигр, покупка которых была мне не по средствам. Поэтому я выучил язык ассемблера и стал искать проверки в двоичных файлах. Тогда на рынке правил бал 8-разрядный процессор 6510 с аккумулятором и двумя регистрами общего назначения. Хотя для того чтобы использовать все 64 КБ системной памяти, требовалось исполнять танцы с бубнами, в целом система была простой. Но поначалу все было непонятно. Со временем я набирался ума от более опытных друзей, и постепенно туман стал рассеиваться. Маршрут был, без сомнения, интересным, но долгим, трудным и иногда заводил в тупик. Многое я бы отдал тогда за путеводитель! Современные 64-разрядные процессоры x86 не в пример сложнее, как и компиляторы, которые генерируют двоичный код. Понять, что делает код, гораздо труднее, чем раньше. Специалист, кото-

рый покажет путь и прояснит сложные вопросы, которые вы могли бы упустить из виду, сделает путешествие короче и интереснее, превратив его в истинное удовольствие.

Дэннис Эндриесс – эксперт по анализу двоичного кода и в доказательство может предъявить степень доктора в этой области – буквально. Однако он не академический ученый, публикующий статьи для себе подобных. Его работы по большей части сугубо практические. Например, он был в числе тех немногих людей, кто сумел разобраться в коде печально известной сети ботов GameOver Zeus, ущерб от которой оценивается в 100 миллионов долларов. И более того, он вместе с другими экспертами безопасности принимал участие в возглавляемой ФБР операции, положившей конец деятельности этой сети. Работая с вредоносными программами, он на практике имел возможность оценить сильные стороны и ограничения имеющихся средств анализа двоичного кода и придумал, как их улучшить. Новаторские методы дизассемблирования, разработанные Дэннисом, теперь включены в коммерческие продукты, в частности Binary Ninja.

Но быть экспертом еще недостаточно. Автор книги должен еще уметь излагать свои мысли. Дэннис Эндриесс обладает этим редким сочетанием талантов: он эксперт по анализу двоичного кода, способный объяснить самые сложные вещи простыми словами, не упуская сути. У него приятный стиль, а примеры ясные и наглядные.

Лично я давно хотел иметь такую книгу. В течение многих лет я читаю курс по анализу вредоносного ПО в Амстердамском свободном университете без учебника ввиду отсутствия такового. Мне приходилось использовать разнообразные онлайн-источники, пособия и эклектичный набор слайдов. Когда студенты спрашивали, почему бы не использовать печатный учебник (как они привыкли), я отвечал, что хорошего учебника по анализу двоичного кода не существует, но если у меня когда-нибудь выдастся свободное время, я его напишу. Разумеется, этого не случилось.

Это как раз та книга, которую я надеялся написать, но так и не собрался. И она лучше, чем мог бы написать я сам.

Приятного путешествия.

Герберт Бос

ПРЕДИСЛОВИЕ

Анализ двоичного кода – одна из самых увлекательных и трудных тем в хакинге и информатике вообще. Она трудна и для изучения, в немалой степени из-за отсутствия доступной информации.

В книгах по обратной разработке и анализу вредоносного ПО недостатка нет, но этого не скажешь о таких продвинутых вопросах анализа двоичного кода, как оснащение двоичной программы, динамический анализ заражения или символическое выполнение. Начинающий аналитик вынужден выискивать информацию по темным закоулкам интернета, в устаревших, а иногда и откровенно вводящих в заблуждение сообщениях в форумах или в непонятно написанных статьях. Во многих статьях, а также в академической литературе по анализу двоичного кода предполагаются обширные знания, поэтому изучение предмета по таким источникам становится проблемой курицы и яйца. Хуже того, многие инструменты и библиотеки для анализа плохо или никак не документированы.

Я надеюсь, что эта книга, написанная простым языком, сделает анализ двоичного кода более доступным и станет практическим введением во все важные вопросы данной области. Прочитав эту книгу, вы будете в достаточной степени экипированы, чтобы ориентироваться в быстро меняющемся мире анализа двоичного кода и отважиться на самостоятельные исследования.

БЛАГОДАРНОСТИ

Прежде всего я хочу поблагодарить свою жену Ноортье и нашего сына Сиетсе за поддержку во время работы над книгой. Это было невероятно горячее время, но вы видели только мою спину.

Я также благодарен всем сотрудникам издательства No Starch Press, которые помогли воплотить эту книгу в реальность, а особенно Биллу Поллоку и Тайлеру Ортману, предоставившим мне возможность взяться за нее, и Анни Чой, Райли Хоффмана и Ким Уимспетт за отличную работу по редактированию и подготовке книги к печати. Спасибо моим техническим рецензентам, Торстену Хольцу и Тиму Видасу, за пространные отзывы, которые способствовали улучшению текста.

Спасибо Бену Грасу, который помог перенести библиотеку libdft на современную версию Ubuntu, Джонатану Сэлуэну за замечания о главах, посвященных символическому выполнению, а также Лоренцо Кавалларо, Эрику ван дер Коуве и всем остальным, кто готовил слайды, легшие в основу приложения, касающегося языка ассемблера.

Наконец, я признателен Герберту Босу, Эйше Словинска и всем моим коллегам, которые создали замечательную среду для научной работы, благодаря чему у меня и появилась идея написать эту книгу.

ВВЕДЕНИЕ

Подавляющее большинство компьютерных программ написаны на языках высокого уровня типа С или С++, которые компьютер не может исполнять непосредственно. Такие программы необходимо откомпилировать, в результате чего создаются *двоичные исполняемые файлы*, содержащие машинный код, – его компьютер уже может выполнить. Но откуда мы знаем, что откомпилированная программа имеет такую же семантику, как исходная? Ответ может разочаровать – *а мы и не знаем!*

Существует семантическая пропасть между языками высокого уровня и двоичным машинным кодом, и как ее преодолеть, знают немногие. Даже программисты в большинстве своем плохо понимают, как их программы работают на самом нижнем уровне, и просто верят, что откомпилированная программа делает то, что они задумали. Поэтому многие дефекты компилятора, тонкие ошибки реализации, потайные ходы на двоичном уровне и другие вредоносные паразиты остаются незамеченными.

Хуже того, существует бесчисленное множество двоичных программ и библиотек – в промышленности, в банках, во встраиваемых системах, – исходный код которых давно утерян или является коммерческой собственностью. Это означает, что такие программы и библиотеки невозможно исправить или хотя бы оценить их безопасность на уровне исходного кода с применением традиционных методов. Это реальная проблема даже для крупных программных компаний, свидетельством чему – недавний выпуск компанией Microsoft созданного с большим трудом двоичного исправления ошибки переполнения буфера в программе «Редактор формул», являющейся частью Microsoft Office¹.

¹ <https://0patch.blogspot.nl/2017/11/did-microsoft-just-manually-patch-their.html>.

В этой книге вы научитесь анализировать и даже модифицировать программы на двоичном уровне. Будь вы хакер, специалист по безопасности, аналитик вредоносного кода, программист или просто любопытствующий, эти методы позволят вам лучше понимать и контролировать двоичные программы, которые вы создаете и используете каждый день.

Что такое анализ двоичных файлов, и зачем он вам нужен?

Анализ двоичных файлов, или просто *двоичный анализ*, – это наука и искусство анализа свойств двоичных компьютерных программ, а также машинного кода и данных, которые они содержат. Короче говоря, цель анализа двоичных файлов – определить (и, возможно, модифицировать) истинные свойства двоичных программ, т. е. понять, что они делают в действительности, а не доверяться тому, что они, по нашему мнению, должны делать.

Многие отождествляют двоичный анализ с обратной разработкой и дизассемблированием, и отчасти они правы. Дизассемблирование – важный первый шаг многих видов двоичного анализа, а обратная разработка – типичное приложение двоичного анализа и зачастую единственный способ документировать поведение проприетарного или вредоносного программного обеспечения. Однако область двоичного анализа гораздо шире.

Методы анализа двоичных файлов можно отнести к одному из двух классов, хотя возможны и комбинации.

Статический анализ В этом случае мы рассуждаем о программе, не выполняя ее. У такого подхода несколько преимуществ: теоретически возможно проанализировать весь двоичный файл за один присест, и для его выполнения не нужен процессор. Например, можно статически проанализировать двоичный файл для процессора ARM на компьютере с процессором x86. Недостаток же в том, что в процессе статического анализа мы ничего не знаем о состоянии выполнения двоичной программы, что сильно затрудняет анализ.

Динамический анализ Напротив, в случае динамического анализа мы запускаем программу и анализируем ее во время выполнения. Часто этот подход оказывается проще статического анализа, потому что нам известно все состояние программы, включая значения переменных и выбор ветвей при условном выполнении. Однако мы видим лишь тот код, который выполняется, поэтому можем пропустить интересные части программы.

И у статического, и у динамического анализ есть плюсы и минусы. В этой книге мы расскажем об обоих направлениях. Помимо пассивного двоичного анализа, вы узнаете о методах *оснащения* дво-

ичных файлов, которые позволяют модифицировать двоичные программы, не имея исходного кода. Оснащение двоичных файлов опирается на такие методы анализа, как дизассемблирование, но и само оно может помочь при проведении двоичного анализа. Из-за такого симбиоза приемов двоичного анализа и оснащения в этой книге рассматриваются как те, так и другие.

Я уже говорил, что анализ можно использовать, чтобы документировать или тестировать на отсутствие уязвимостей программы, для которых у вас нет исходного кода. Но даже если исходный код имеется, такой анализ может быть полезен для поиска тонких ошибок, которые более отчетливо проявляются на уровне двоичного, а не исходного кода. Многие методы анализа двоичных файлов полезны также для отладки. В этой книге рассматриваются методы, применимые во всех этих ситуациях, и не только.

В чем сложность анализа двоичных файлов?

Двоичный анализ – вещь гораздо более трудная, чем эквивалентный анализ на уровне исходного кода. На самом деле многие задачи в этом случае принципиально неразрешимы, т. е. невозможно сконструировать движок, который всегда возвращает правильный результат! Чтобы вы могли составить представление о том, с какими проблемами предстоит столкнуться, ниже приведен список некоторых вещей, затрудняющих анализ двоичных файлов. Увы, этот список далеко не исчерпывающий.

Отсутствует символическая информация В исходном коде, написанном на языке высокого уровня типа C или C++, мы даем осмысленные имена переменным, функциям, классам и т. п. Эти имена мы называем *символической информацией*, или просто *символами*. Если придерживаться хороших соглашений об именовании, то понять исходный код будет гораздо проще, но на двоичном уровне имена не имеют ни малейшего значения. Поэтому из двоичных файлов информация о символах часто удаляется, из-за чего понять код становится гораздо труднее.

Отсутствует информация о типах Еще одна особенность программ на языках высокого уровня – наличие у переменных четко определенных типов, например `int`, `float`, `string` или более сложных структурных типов. На двоичном же уровне типы нигде явно не упоминаются, поэтому понять структуру и назначение данных нелегко.

Отсутствуют высокоуровневые абстракции Современные программы состоят из классов и функций, но компиляторы отбрасывают эти высокоуровневые конструкции. Это означает, что двоичный файл выглядит как огромный «комок» кода и данных, а не хорошо структурированный код, и восстановить высокоуровневую структуру трудно и чревато ошибками.

Конец ознакомительного фрагмента.

Приобрести книгу можно

в интернет-магазине

«Электронный универс»

e-Univers.ru