# Code Craft

## The Practice of Writing Excellent Code

*Pete Goodliffe*

ПРОФЕССИОНАЛЬНО

# Ремесло программиста

## Практика написания хорошего кода

*Питер Гудлиф*

СИМВОЛ®

*Санкт-Петербург — Москва*
*2009*

Серия «Профессионально»

Питер Гудлиф

# Ремесло программиста.
## Практика написания хорошего кода

Перевод С. Маккавеева


| | |
|---|---|
| Главный редактор | *А. Галунов* |
| Зав. редакцией | *Н. Макарова* |
| Научный редактор | *Б. Попов* |
| Редактор | *А. Петухов* |
| Корректор | *О. Макарова* |
| Верстка | *Д. Орлова* |
| Художник | *О. Макарова* |

*Гудлиф П.*

Ремесло программиста. Практика написания хорошего кода. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 704 с., ил.

ISBN 978-5-93286-127-1



Ничто не сравнится по ценности с советами настоящего программиста-профессионала. Книга Питера Гудлифа «Ремесло программиста» написана ясно, практично и занимательно. Она поможет вам перейти на более высокий уровень мастерства программирования и покажет, как писать код, который больше чем «просто работает». Да, вы умеете писать работающий код, но как написать понятный код? Как добиться его надежности и отсутствия ошибок? Смогут ли другие программисты выяснить логику и цель вашего кода? Выдающиеся программисты не просто обладают техническими знаниями – у них есть правильный подход и отношение к программированию.

Перед вами руководство по выживанию в условиях промышленного производства ПО. Эта книга посвящена тому, чему вас никто не учил: как *правильно* программировать в *реальной* жизни. Здесь вы найдете не связанные с конкретными языками рекомендации, полезные всем разработчикам и касающиеся таких проблем, как стиль представления, выбор имен переменных, обработка ошибок, безопасность, эффективность групповой работы, технологии разработки и составление документации.

Читатель должен обладать опытом программирования, ибо книга не учит программированию – она учит *правильно* программировать. Издание будет полезно и студентам старших курсов, знакомым с принципами программирования.



**ISBN 978-5-93286-127-1**
**ISBN 978- 1-59327-119-0 (англ)**





Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 26.12.2008. Формат 70x100 1/16 . Печать офсетная.
Объем 44 печ. л. Тираж 1500 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

# Оглавление

**Об авторе** . . . . . 16
**Благодарности** . . . . . 17
**Предисловие** . . . . . 19

**I. Перед лицом кода** . . . . . 27

**1. Держим оборону** . . . . . 29

На пути к хорошему коду . . . . . 30
Готовьтесь к худшему . . . . . 31
Что такое защитное программирование? . . . . . 32
Этот страшный, ужасный мир . . . . . 35
Технологии защитного программирования . . . . . 36
Ограничения . . . . . 45
Резюме . . . . . 50
Контрольные вопросы . . . . . 51

**2. Тонкий расчет** . . . . . 53

Да в чем проблема? . . . . . 54
Знайте своих клиентов . . . . . 55
Что такое хорошее представление? . . . . . 56
Размещение скобок . . . . . 57
Единственно верный стиль . . . . . 61
Внутрифирменные стили (и когда их придерживаться) . . . . . 63
Установка стандарта . . . . . 65
Религиозные войны? . . . . . 68
Резюме . . . . . 68
Контрольные вопросы . . . . . 71

**3. Что в имени тебе моем?** . . . . . 73

Зачем нужны хорошие имена? . . . . . 75
Каким объектам мы даем имена? . . . . . 75
Игра в названия . . . . . 76
Технические подробности . . . . . 79

Роза пахнет розой . . . . . 86
Резюме . . . . . 88
Контрольные вопросы . . . . . 90

**4. Литературоведение** . . . . . 93

Самодокументируемый код . . . . . 95
Техника написания самодокументируемого кода . . . . . 98
Практические методологии самодокументирования . . . . . 103
Резюме . . . . . 108
Контрольные вопросы . . . . . 109

**5. Заметки на полях** . . . . . 113

Что есть комментарий в коде? . . . . . 114
Как выглядят комментарии? . . . . . 115
Сколько комментариев требуется? . . . . . 115
Что помещать в комментарии? . . . . . 116
На практике . . . . . 120
Замечание об эстетичности . . . . . 121
Работа с комментариями . . . . . 126
Резюме . . . . . 129
Контрольные вопросы . . . . . 130

**6. Людям свойственно ошибаться** . . . . . 133

Откуда что берется . . . . . 134
Механизмы сообщения об ошибках . . . . . 135
Обнаружение ошибок . . . . . 141
Обработка ошибок . . . . . 143
Подымаем скандал . . . . . 151
Управление ошибками . . . . . 153
Резюме . . . . . 154
Контрольные вопросы . . . . . 156

**II. Тайная жизнь кода** . . . . . 157

**7. Инструментарий программиста** . . . . . 159

Что такое инструмент программирования? . . . . . 160
А зачем они нужны – инструменты? . . . . . 162
Электроинструменты . . . . . 164
Какой инструмент необходим? . . . . . 167
Резюме . . . . . 179
Контрольные вопросы . . . . . 180

**8. Время испытаний** . . . . . 183

Проверка на подлинность . . . . . 185

Кто, что, когда, зачем? . . . 186
Тестировать легко... . . . 191
Типы тестирования . . . 195
Выбор контрольных примеров для блочного тестирования . . . 200
Архитектура и тестирование . . . 202
Руками не трогать! . . . 203
Анатомия провала . . . 204
Справлюсь ли я сам? . . . 205
Резюме . . . 208
Контрольные вопросы . . . 209

**9. Поиск ошибок** . . . 211

Реальные факты . . . 212
Природа этого зверя . . . 213
Борьба с вредителями . . . 220
Охота за ошибками . . . 222
Как исправлять ошибки . . . 228
Профилактика . . . 230
Спрей от ос, репеллент для мух, липучки... . . . 231
Резюме . . . 233
Контрольные вопросы . . . 235

**10. Код, который построил Джек** . . . 237

Языковые барьеры . . . 238
Делаем слона из мухи . . . 243
Выполнение сборки . . . 245
Что должна уметь хорошая система сборки? . . . 249
Механика сборки . . . 253
Отпусти меня... . . . 259
Мастер на все руки . . . 262
Резюме . . . 263
Контрольные вопросы . . . 264

**11. Жажда скорости** . . . 267

Что такое оптимизация? . . . 268
От чего страдает оптимальность кода? . . . 270
Доводы против оптимизации . . . 271
Нужна ли оптимизация . . . 273
Технические подробности . . . 274
Методы оптимизации . . . 279
Как писать эффективный код . . . 289
Резюме . . . 291
Контрольные вопросы . . . 292

**12. Комплекс незащищенности** . . . . . 295

Риски . . . . . 296
Наши оппоненты . . . . . 299
Оправдания, оправдания . . . . . 302
Ощущение незащищенности . . . . . 302
Дела защитные . . . . . 306
Резюме . . . . . 311
Контрольные вопросы . . . . . 312

**III. Проектирование кода** . . . . . 315

**13. Важность проектирования** . . . . . 317

Программирование как конструкторская работа . . . . . 318
Что нужно проектировать? . . . . . 319
Из-за чего весь этот шум? . . . . . 320
Хороший проект программного продукта . . . . . 321
Как проектировать код . . . . . 332
Резюме . . . . . 337
Контрольные вопросы . . . . . 339

**14. Программная архитектура** . . . . . 341

Что такое программная архитектура? . . . . . 343
Какими качествами должна обладать архитектура? . . . . . 350
Архитектурные стили . . . . . 351
Резюме . . . . . 359
Контрольные вопросы . . . . . 360

**15. Программное обеспечение – эволюция или революция?** . . . . . 363

Гниение программного обеспечения . . . . . 365
Тревожные симптомы . . . . . 367
Как развивается код? . . . . . 369
Вера в невозможное . . . . . 372
Как с этим бороться? . . . . . 373
Резюме . . . . . 377
Контрольные вопросы . . . . . 378

**IV. Стадо программистов?** . . . . . 381

**16. Кодеры** . . . . . 383

Мартышкин труд . . . . . 384
Идеальный программист . . . . . 398
И что из этого следует? . . . . . 399
Для глупцов . . . . . 400
Резюме . . . . . 401

План действий . . . . . 402
Контрольные вопросы . . . . . 402

**17. Вместе мы – сила** . . . . . 405

Команды – общий взгляд . . . . . 406
Организация команды . . . . . 408
Инструменты для групповой работы . . . . . 412
Болезни, которым подвержены команды . . . . . 413
Личное мастерство и качества, необходимые для работы в команде . . . . . 426
Принципы групповой работы . . . . . 431
Жизненный цикл команды . . . . . 434
Резюме . . . . . 442
План действий . . . . . 444
Контрольные вопросы . . . . . 444

**18. Защита исходного кода** . . . . . 447

Наши обязанности . . . . . 448
Управление версиями исходного кода . . . . . 449
Управление конфигурацией . . . . . 456
Резервное копирование . . . . . 458
Выпуск исходного кода . . . . . 459
Где я оставлю свой код... . . . . 461
Резюме . . . . . 462
Контрольные вопросы . . . . . 463

**V. Часть процесса** . . . . . 465

**19. Спецификации** . . . . . 467

Что же это такое, конкретно? . . . . . 469
Типы спецификаций . . . . . 470
Что должны содержать спецификации? . . . . . 478
Процесс составления спецификаций . . . . . 481
Почему мы не пишем спецификации? . . . . . 484
Резюме . . . . . 486
Контрольные вопросы . . . . . 487

**20. Рецензия на отстрел** . . . . . 489

Что такое «рецензирование кода»? . . . . . 490
Когда проводить рецензирование? . . . . . 491
Проведение рецензирования кода . . . . . 494
Пересмотрите свое отношение . . . . . 498
Идеальный код . . . . . 501
За пределами рецензирования кода . . . . . 502

Резюме . . . . . 502
Контрольный список . . . . . 504
Контрольные вопросы . . . . . 504

**21. Какой длины веревочка?** . . . . . 507

Выстрел в темноте . . . . . 508
Почему трудно делать оценки? . . . . . 509
Под давлением . . . . . 512
Практические способы оценки . . . . . 513
Игры с планами . . . . . 517
Не отставай! . . . . . 521
Резюме . . . . . 524
Контрольные вопросы . . . . . 525

**VI. Вид сверху** . . . . . 527

**22. Рецепт программы** . . . . . 529

Стили программирования . . . . . 530
Рецепты: как и что . . . . . 535
Процессы разработки . . . . . 536
Спасибо, хватит! . . . . . 549
Выбор процесса . . . . . 550
Резюме . . . . . 551
Контрольные вопросы . . . . . 552

**23. За гранью возможного** . . . . . 555

Программирование приложений . . . . . 556
Программирование игр . . . . . 560
Системное программирование . . . . . 561
Встроенное программное обеспечение . . . . . 563
Программирование распределенных систем . . . . . 566
Программирование веб-приложений . . . . . 569
Программирование масштаба предприятия . . . . . 571
Численное программирование . . . . . 572
И что дальше? . . . . . 574
Резюме . . . . . 574
Контрольные вопросы . . . . . 576

**24. Что дальше?** . . . . . 577

Но что же дальше? . . . . . 578

**Ответы и обсуждение** . . . . . 581

Глава 1. Держим оборону . . . . . 581
Глава 2. Тонкий расчет . . . . . 585

Глава 3. Что в имени тебе моем? . . . . . 594
Глава 4. Литературоведение . . . . . 599
Глава 5. Заметки на полях . . . . . 605
Глава 6. Людям свойственно ошибаться . . . . . 607
Глава 7. Инструментарий программиста . . . . . 612
Глава 8. Время испытаний . . . . . 615
Глава 9. Поиск ошибок . . . . . 622
Глава 10. Код, который построил Джек . . . . . 624
Глава 11. Жажда скорости . . . . . 633
Глава 12. Комплекс незащищенности . . . . . 638
Глава 13. Важность проектирования . . . . . 643
Глава 14. Программная архитектура . . . . . 646
Глава 15. Программное обеспечение – эволюция или революция? . . 652
Глава 16. Кодеры . . . . . 657
Глава 17. Вместе мы – сила . . . . . 659
Глава 18. Защита исходного кода . . . . . 665
Глава 19. Спецификации . . . . . 671
Глава 20. Рецензия на отстрел . . . . . 675
Глава 21. Какой длины веревочка? . . . . . 678
Глава 22. Рецепт программы . . . . . 681
Глава 23. За гранью возможного . . . . . 685

**Библиография** . . . . . 688

**Алфавитный указатель** . . . . . 693

# Отзывы на книгу «Ремесло программиста»

«Овладеть ремеслом – это не только освоить приемы и инструменты; здесь нужны позиция и мастерство. Те программисты, которые неравнодушны к своему делу, именно это и узнают из книги. При содействии большого числа обезьянок эта книга приглашает читателя задуматься над тем, чем они занимаются».

Кевлин Хенни (Kevlin Henney),
независимый консультант

«Легко читается, занимательно, даже забавно... Книга полна мудрости, накопленной за годы реальной работы, мучений и побед в сфере программных разработок... Жаль, что у меня не было такой книги, когда я начинал работать программистом».

Стив Лав (Steve Love), старший разработчик

«Эта книга – кладезь информации, необходимой каждому профессиональному разработчику программного обеспечения».

Тим Пенхи (Tim Penhey), редактор C VU

«Здравость суждений приходит с опытом. А опыт – вы получаете его в результате своих «нездравых» суждений! Здесь у вас есть возможность поучиться на чужом, дорого доставшемся опыте, с бóльшей пользой и меньшими муками».

Луи Гольдтвейт (Lois Goldthwaite),
член комитетов по стандартизации C++ и POSIX BSI

«Именно та книга, которая нужна необстрелянным новобранцам. Рассказывает всю правду, легко читается и охватывает широкий круг тем, которые должен знать новичок».

Джон Джеггер (Jon Jagger),
наставник, консультант, преподаватель, программист

«Уникальное и практическое руководство для становления профессионального программиста в современных условиях».

Эндрю Берроуз (Andrew Burrows), разработчик ПО

«Пит обладает редким талантом. Он может не только определить технику, которой пользуются лучшие профессиональные разработчики программ (часто не догадываясь об этом), но и описать ее четким и сжатым образом».

Грег Лоу (Greg Law), CEO, UNDO Ltd.

«Жаль, что этой книги не существовало в начале моей карьеры, когда учили меня. Но хотя бы теперь я могу воспользоваться ею, когда учу других».

Доктор Эндрю Беннет (Andrew Bennett),
старший программист, B.ENG., Ph.D., MIET, MIEEE

«Те, кому посчастливилось присутствовать на лекциях Пита Гудлифа, сразу узнают его манеру рассказывать о предмете понятно и с юмором. В учебной среде это оборачивается направленным структурированным преподаванием, которое позволяет учиться и развиваться как новичку, так и опытному профессионалу».

Роберт Д. Шофилд (Robert D. Schofield), M.SC.,
основатель MIET, SCIENTIFIC SOFTWARE SERVICES Ltd.

«Пит хочет, чтобы программисты писали не просто код, а хороший код, пользуясь правильными инструментами и методами. Книга исследует широкий круг проблем программирования и излагает нормы и принципы, с которыми должен быть знаком каждый разработчик, которому небезразлично его дело».

Крис Рид (Chris Reed), программист

«Увлеченность Пита Гудлифа идеей повышения профессионализма в области разработки ПО давно известна. Обладая солидными знаниями вкупе с даром занимательно и понятно излагать их, Пит проявил себя как прекрасный наставник и для новичков, и для опытных разработчиков».

Роб Войси (Rob Voisey),
технический директор, AKAI DIGITAL Ltd.

«Мне больше всего понравились обезьянки».

Алиса Гудлиф, $4\,{}^{1}/_{2}$ года

# Об авторе

Пит Гудлиф (Pete Goodliffe) – опытный разработчик программного обеспечения, постоянно меняющий свою роль в цепи программных разработок; он занимался разработками на многих языках в различных проектах. Кроме того, у него большой опыт обучения и повышения квалификации программистов. Пит ведет регулярную колонку «Professionalism in Programming» в журнале *C Vu*, издаваемом ACCU (*www.accu.org*). Он любит писать превосходный код, в котором нет ошибок, благодаря чему он может больше времени проводить со своими детьми.

# Предисловие

*Есть много вещей, о которых умный человек*
*предпочел бы не знать.*

Ральф Уолдо Эмерсон

В основе этой книги лежит полученный боевой опыт. На самом деле, она отражает глубинные процессы, идущие там, где разрабатывают программы, но часто между тем и другим мало различий. Книга написана для программистов, которым *небезразлично* дело, которым они занимаются. Если вы не из их числа, можете сразу закрыть ее и аккуратно поставить обратно на полку.

## Какая мне от этого может быть польза?

Программирование – ваша страсть. Печально, но это так. И как закоренелый технарь вы программируете чуть ли не во время ночного сна. И вот вы попали в центр реального мира, в саму эту отрасль, и занимаетесь тем, о чем и не мечтали: забавляетесь с компьютером, а вам за это еще и деньги платят. Ведь *вы сами* готовы были заплатить за то, чтобы иметь такую возможность!

Но все не так просто и не похоже на то, чего вы ожидали. Огорошенные назначением вам нереальных сроков выполнения задач и неумелым руководством (если его можно назвать этим словом), непрерывным изменением технического задания и необходимостью разбираться в дрянном коде, доставшемся вам от предшественников, вы начинаете сомневаться в том, что выбрали для себя *правильный путь*. Все вокруг мешает вам писать тот код, о котором вы мечтали. Что ж, таковы условия существования в организациях, где пишут программы. Вы попали на передний край упорной битвы за создание шедевров художественного мастерства и научного гения. Удачи вам!

Вот тут вам и может пригодиться «Ремесло программиста». Эта книга посвящена тому, чему вас никто не учил: как правильно программировать *в реальной жизни*. Конечно, в ней рассказывается о технических приемах и хитростях, позволяющих писать хороший код. Но в ней

говорится и кое о чем еще: о том, как писать *правильный* код *правильным* образом.

Что это значит? Есть много аспектов написания хорошего кода в реальном мире:

- Разработка технически элегантного кода
- Создание кода, доступного для сопровождения, т. е. понятного другим
- Способность разобраться в чужом запутанном коде и переделать его
- Умение работать вместе с другими программистами

Все эти навыки (и многие другие) необходимы, чтобы стать настоящим кодером. Вы должны знать скрытую жизнь своего кода: что происходит с ним после того, как вы его набрали. У вас должно быть развито эстетическое чувство: красивый код отличается от уродливого. И нужно обладать практицизмом: решать, когда оправданы упрощения, когда требуется поработать над архитектурой кода, а когда нужно все бросить и двигаться дальше (прагматический принцип «*не трогай то, что уже работает*»). Эта книга поможет вам решать такие задачи. Вы узнаете, как выжить в условиях промышленного производства программ, как вести разведку и выяснять замыслы противника, какой тактики придерживаться, чтобы не угодить в расставленные противником ловушки, и как, несмотря на все препятствия, все-таки создавать *прекрасные* программы.

Разработка программ – интересная профессия. Она динамична, в ней множество преходящих модных поветрий, схем быстрого обогащения и проповедников новых идеологий. Она еще не достигла зрелости. Я не претендую на изобретение чудодейственных средств, но у меня есть некоторые практичные и полезные рекомендации, которыми я хочу поделиться. Это не теория башни из слоновой кости, а реальный опыт и добросовестная практика.

К тому моменту, когда вы переварите этот материал, вы не просто научитесь лучше программировать. Вы сможете успешнее выживать в условиях этой отрасли. Станете настоящим бойцом. Вы освоите ремесло кодировщика. Если такая перспектива вас не вдохновляет, вам, возможно, стоит подумать о военной карьере.

## Стремление к совершенству

Так чем же *хорошие* программисты отличаются от *плохих*? Или лучше – чем *отличные* программисты разнятся от *удовлетворительных*? Секрет заключается не только в технической компетенции – я встречал толковых программистов, способных энергично и впечатляюще писать на C++, знающих этот язык на зубок, но код их просто ужасал. Знавал я и более скромных программистов, которые старались писать очень простой код, но их программы были чрезвычайно элегантны и хорошо продуманы.

В чем реальная разница? В основе хорошего программирования лежит ваша *позиция*. Она состоит в умении профессионально решать задачи и в стремлении всегда как можно лучше писать код вопреки давлению, оказываемому на вас условиями работы. Позиция – это очки, через которые мы смотрим на вещи. Через нее мы воспринимаем свою работу и свое поведение. Хороший код появляется в результате тщательного труда мастера, а не бездумного хакерства неряшливого программиста.

## Позиция – угол сближения

Чем дольше я изучал и систематизировал область разработки программного обеспечения, тем более убеждался, что незаурядные программисты отличаются именно особой позицией. В словаре значение слова «позиция» (attitude) объясняется примерно так:

**attitude (at.ti.tude)**

1. **Состояние ума или чувств; настрой к чему-либо.**
2. **Положение самолета в воздухе относительно некоторого эталонного.**

Первое определение не содержит ничего неожиданного, но второе... Оно оказывается интереснее первого.

Сквозь самолет проводят три воображаемые осевые линии: одну через крылья, другую от носа до хвоста и третью вертикально через пересечение первых двух. Летчик поворачивает самолет вокруг этих осей: они определяют угол наклона траектории. Это и есть угловое положение самолета. Если самолет, находясь в неверном угловом положении, слегка прибавит мощности, то он существенно отклонится от цели. Пилот должен постоянно контролировать угловое положение летательного аппарата, особенно в такие критические периоды, как взлет или посадка.

Рискуя уподобиться дрянному мотивационному фильму, я все же отмечу большое сходство этой ситуации с разработкой программного обеспечения. Позиция самолета в пространстве определяет его угловое положение, а наша позиция определяет наше угловое положение относительно задачи кодирования. Неважно, насколько грамотен программист технически – если его способности не сдерживаются разумной позицией, от этого пострадает результат.

Неверная позиция может привести к краху программного проекта, поэтому в программировании очень важно сохранять правильное угловое положение. Ваша позиция окажется либо тормозом, либо ускорителем вашего личного роста. Чтобы совершенствоваться как программисты, мы должны обеспечить себе правильную позицию.

*Путь к гибельному коду вымощен благими намерениями*. Чтобы стать отличным программистом, нужно научиться быть выше своих намерений, искать положительные перспективы и развивать в себе такую здравую позицию.

В этой книге будет показано, как это делается. Она охватывает широкий спектр – от практических приемов написания кода до крупных организационных проблем. И во всех этих темах я подчеркиваю, какие позиции и подходы будут правильными.

## Кому адресована эта книга?

Очевидно, что читать эту книгу следует тем, кто заинтересован в улучшении качества своего кода. Мы все должны стремиться к совершенствованию своего программистского мастерства. Если у вас нет такого стремления, эта книга вам не нужна. Она написана для профессиональных программистов, занимающихся этой работой несколько лет. Книга будет полезна и студентам старших курсов, которые знакомы с принципами программирования, но не уверены в том, как лучше их применять.

Читатель должен обладать опытом программирования. Эта книга не учит программированию; она учит *правильно* программировать. Я старался не навязывать определенных языков и не быть категоричным, но в книгу необходимо было включить примеры кода. Большинство из них написано на популярных в данное время языках: C, C++ и Java. Для понимания этих примеров не требуется глубокого знания языка, поэтому не стоит пугаться, если вы не являетесь классным специалистом по C++.

Предполагается, что читатель активно занимается разработкой кода в условиях некоего программного производства или планирует заняться ею в будущем. Это может означать работу в коммерческой организации, либо участие в каком-нибудь хаотичном проекте свободно распространяемого программного обеспечения, либо разработку программ по контракту.

## Какие темы освещаются

В книге рассматриваются вопросы позиции и отношения программиста, но это не учебник психологии. В число обсуждаемых тем входят:

- Представление исходного кода
- Технологии защитного кодирования
- Эффективная отладка программ
- Особенности работы в группе
- Управление исходным кодом

Взглянув на оглавление, вы сможете точно выяснить, какие темы освещены. Чем я руководствовался при выборе тем? Я многие годы занимаюсь обучением программистов, поэтому выбрал вопросы, которые периодически возникают в этом процессе. Я также достаточно долго участвовал в разработке программ и знаю, какие проблемы при этом постоянно возникают – к ним я также обращаюсь.

Если вы сможете победить всех этих злых духов программирования, то превратитесь из подмастерья в настоящего мастера программирования.

## Структура книги

Я постарался максимально облегчить чтение книги. Здравый смысл подсказывает, что книгу нужно читать последовательно с начала до конца. К данной книге это не относится. Можете открыть любую главу, которая вас заинтересовала, и начать чтение с нее. Каждая глава самостоятельна, но содержит полезные перекрестные ссылки, благодаря которым видна общая взаимосвязь. Конечно, если вы предпочитаете традиционное чтение, ничто не мешает начать с самого начала.

Структура всех глав одинакова и не сулит неприятных сюрпризов. Главы состоят из следующих разделов:

**В этой главе**

Сначала перечисляются основные темы главы. В нескольких строках дается обзор содержания. Можете прочесть их и выяснить, о чем будет рассказано.

**Глава**

Тот захватывающий материал, за возможность прочесть который вы заплатили приличные деньги.

По всей главе встречаются «золотые правила». Таким образом я выделяю важные советы, проблемы и позиции, поэтому обратите на них внимание. Выглядят они так:


*Это важно. Обратите внимание!*

**Резюме**

Этот маленький раздел в конце каждой главы подытоживает изложение. В нем дан общий обзор материала. Если вас действительно поджимает время, можете прочесть только «золотые правила» и этот завершающий раздел. Только никому не говорите, что я вам это посоветовал.

После этого я сравниваю подходы, применяемые хорошими и плохими программистами, чтобы вывести из них правильные установки, которые вам необходимо в себе выработать. При достаточной

смелости можете оценить себя по этим примерам; будем надеяться, что правда окажется не слишком болезненной!

**См. также**

Перечисляются родственные главы и объясняется, каким образом они связаны с рассматриваемой темой.

**Контрольные вопросы**

В конце задается несколько вопросов. Они включены не для того, чтобы «раздуть» книгу, а входят неотъемлемой частью в каждую главу. Вопросы рассчитаны не на простое перелистывание прочитанного материала, а имеют целью *заставить читателя задуматься*, не ограничиваясь при этом содержанием конкретной главы. Вопросы делятся на две группы:

- **Вопросы для размышления.** В них углубленно разбирается тема главы и поднимаются некоторые важные проблемы.
- **Вопросы личного характера.** Эти вопросы анализируют практику работы и качество кодирования, характерные для вас и группы разработчиков, в которую вы входите.

Не проходите мимо этих вопросов! Даже если вам лень сесть и серьезно поискать ответ на каждый вопрос (поверьте, это принесло бы вам большую пользу), хотя бы прочтите их и слегка задумайтесь.

В последней части книги есть *ответы* на эти вопросы и их *обсуждение*. Это не просто список ответов, потому что многие вопросы не позволяют четко ответить *да* или *нет*. После обдумывания вопросов сравните свои ответы с моими. Мои «ответы» часто содержат дополнительную информацию, которой нет в основной главе.

## Содержание глав

Каждая глава посвящена одной теме – одной из современных проблем разработки программного обеспечения. Это обычные причины, по которым программисты пишут плохой код или плохо пишут код. В каждой главе описаны правильные подходы и установки, делающие более сносной жизнь бойца на передовой.

Главы распределены по шести частям. В начале каждой части приводится оглавление и краткое описание содержимого каждой главы. Части организованы так, что сначала рассматривается, *какой* код мы пишем, а в конце – *как* мы его пишем.

Наше исследование начинается с внешнего вида кода, сосредоточиваясь на микроуровне написания исходного кода. Я умышленно начинаю с этих вопросов – программисты *редко* обращают на них внимание:

**Часть I. Перед лицом кода**

В этой части рассматриваются основные элементы разработки исходного кода. Мы изучим методы защитного программирования

и способы форматирования кода. Затем займемся способами выбора имен и документирования кода. Обсуждаются также стандарты написания комментариев и методики обработки ошибок.

**Часть II. Тайная жизнь кода**

Здесь мы переходим к *процессу* написания кода: как мы создаем его и как с ним работаем. Мы рассмотрим строительный инструмент, методы тестирования, технику отладки, правильную процедуру сборки выполняемых модулей и оптимизацию. В конце остановимся на том, как писать безопасные программы.

**Часть III. Проектирование кода**

Эта часть посвящена более общим проблемам построения исходного кода. Мы обсудим проектирование кода, архитектуру программного обеспечения и развитие (или распад) кода с течением времени.

Затем мы перейдем на *макроуровень* – оторвем взгляд от земли и посмотрим, что делается вокруг – как идет жизнь в организации, разрабатывающей программы. Большие программы создаются только группами разработчиков, и в следующих трех частях мы узнаем о приемах и методах повышения эффективности работы таких групп.

**Часть IV. Стадо программистов?**

Программисты редко живут в вакууме. (Для этого им нужно специальное дыхательное оборудование.) В этой части мы расширим свой кругозор и рассмотрим правильные приемы разработки и их место в повседневной жизни профессионального программиста. Здесь будет рассказано о хорошей практике программирования отдельного программиста и группы, а также об использовании системы контроля версий.

**Часть V. Часть процесса**

Рассматриваются некоторые процедуры и ритуалы процесса разработки программного обеспечения: составление спецификаций, ревизии кода, черная магия составления графиков работ.

**Часть VI. Вид сверху**

В заключительной части процесс разработки рассматривается на более высоком уровне и исследуются методологии разработки программного обеспечения, а также различные дисциплины, входящие в программирование.

## Как пользоваться этой книгой

Начнете ли вы читать эту книгу с первой страницы или выберете те места, которые вам интересны, значения не имеет.

Важно, чтобы вы читали «Ремесло программиста» непредвзято и размышляли о том, как применить прочитанное в своей работе. *Умный учится на своих ошибках, а тот, кто еще умнее, – на чужих.* Всегда

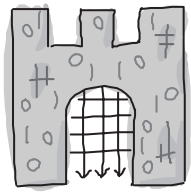
полезно узнать что-то из опыта других людей, поэтому изучая материал этой книги, обсудите его с программистом, чье мнение вы цените. Рассмотрите вместе с ним предлагаемые вопросы.

Я надеюсь, что процесс освоения мастерства кодировщика доставит вам удовольствие. Завершив чтение, оцените, насколько лучше вы стали разбираться в этом ремесле, насколько выросло ваше мастерство и как улучшились ваши установки. Если не произошло никаких перемен, значит, книга не достигла своей цели. Уверен, что такого не случится.

## Замечание для тех, кто занимается обучением

Эта книга очень полезна при обучении менее опытных программистов. Она специально задумывалась для этих целей, и ее содействие росту их мастерства и понимания несомненно.

Не советую методически прорабатывать на занятиях каждый раздел. Пусть лучше ученики прочтут главу самостоятельно, а потом ее содержание можно обсудить вместе. Включенные в главу вопросы послужат трамплином для обсуждения, поэтому полезно начать с них.



# 12

# Комплекс незащищенности

*Как писать защищенные программы*

*В этой главе:*

- Угрозы безопасности для действующего кода
- Как взломщики могут злоупотребить вашим кодом
- Технология снижения уязвимости кода

*Безопасность обычно оказывается предрассудком. Ее не существует в природе... Жизнь интересна, только когда она – смелое приключение.*

Хелен Келлер

Не столь далеко в прошлое ушли времена, когда доступ к компьютеру был редкой возможностью. Во всем мире существовала лишь горстка машин, принадлежавших нескольким организациям, и работали на них небольшие группы высококвалифицированных специалистов. В те дни компьютерная безопасность состояла в надевании лабораторного халата и предъявлении пропуска охране у входа.

Возьмем сегодняшний день. Мы носим в карманах мини-компьютеры такой вычислительной мощности, о которой операторы прежних лет даже и не мечтали. Компьютеры во множестве присутствуют повсюду, и, как правило, между ними установлена хорошая связь.

Объем данных, циркулирующих в компьютерных системах, растет с фантастической скоростью. Мы пишем программы для хранения, обработки, интерпретации и передачи этих данных. Наше программное обеспечение не должно допускать попадания этой информации в чужие руки: к злоумышленникам, случайным людям или просто в эфир. Это очень важно; утечка из компании сверхсекретной информации может привести ее к финансовому краху. Нельзя, чтобы конфиденциальные личные данные (например, данные вашего банковского счета или кредитной карты) стали известны постороннему, который может ими воспользоваться. В большинстве компьютерных систем требуется некоторый уровень безопасности.[1]

Кто несет ответственность за безопасность создаваемого программного обеспечения? К несчастью, это *наша* забота. Если мы не уделим пристальное внимание безопасности результатов своего труда, то будем выпускать ненадежные, дырявые программы и пожнем все плоды этого.

Безопасность программного обеспечения – это большая проблема, с которой мы обычно плохо справляемся. Едва ли не каждый день становится известно об обнаружении новой уязвимости в системе защиты популярного продукта или компрометации систем в результате действия вирусов.

Это слишком обширная тема, чтобы подробно осветить ее в данной книге. Работа в этой области требует большой подготовки и опыта. Однако даже элементарные ее проблемы не находят адекватного отражения при обучении разработке программного обеспечения. Задача данной главы – осветить задачи системы безопасности, исследовать возникающие проблемы и рассказать о некоторых базовых технологиях защиты кода.

## Риски

*Лучше терпеть насмешки над чрезмерностью своих страхов, чем попасть в беду, переоценив свою безопасность.*

Эдмунд Берк

Кому может понадобиться атаковать вашу систему? Тому, кто захочет воспользоваться тем, чем вы располагаете. Это могут быть:

- Вычислительные мощности.
- Возможность отправки данных (например, для рассылки спама).
- Ваши конфиденциальные данные.
- Ваши возможности, например определенное программное обеспечение, которое у вас установлено.

[1] Как мы увидим далее, это не зависит от того, работают ли они с секретными данными. Если у малозначительной компоненты есть открытый интерфейс, она несет в себе риск для безопасности системы в целом.

- Ваше подключение к представляющим интерес удаленным системам.

Вас могут атаковать просто потому, что вы кому-то не понравились и он хочет навредить вам, нарушив работу вашего компьютера. С одной стороны, вокруг полно злоумышленников, ищущих легкой добычи, а с другой – сама программа может ошибочно направить информацию не тому, кому она предназначалась. Тот, к кому она попала, может воспользоваться утечкой и нанести вам вред.


*Следует учитывать, какими важными ресурсами вы располагаете. Есть ли у вас особо деликатная информация или специальные возможности, за которыми могут охотиться? Обеспечьте их защиту.*

Чтобы понять, какого типа атакам вы можете подвергнуться, важно провести различие между защитой целой компьютерной *системы* (состоящей из нескольких компьютеров, сети и ряда взаимодействующих приложений) и написанием отдельных защищенных *программ*. То и другое – важные аспекты компьютерной безопасности; они сливаются между собой, поскольку оба необходимы. Второе – это часть первого. Достаточно одной небезопасной программы, чтобы сделать всю компьютерную систему (или сеть) незащищенной.

Вот стандартные риски системы защиты и способы компрометации действующих компьютерных систем:

- Вор, укравший ноутбук или PDA, может прочесть все незащищенные конфиденциальные данные. Украденное устройство может оказаться настроенным для автоматического входа в закрытую сеть, позволяя благополучно пройти все системы защиты вашей компании. Это серьезная угроза системе безопасности, которой не так просто противодействовать при написании кода! Но мы можем писать системы так, чтобы тот, кто украл компьютер, не получал к ним доступа сразу.
- Недостатки процедур ввода данных могут допускать злонамеренное их использование (эксплойт, exploit), что приводит к различным видам компрометации – вплоть до полного захвата машины атакующим (мы продемонстрируем это в разделе «Переполнение буфера» на стр. 303).

  Особенно неприятно проникновение через незащищенный открытый сетевой интерфейс. Если уязвимостью в GUI могут воспользоваться только те, кто фактически пользуется этим интерфейсом, то незащищенность системы, подключенной к открытой сети, может привести к тому, что к вам может попытаться вломиться кто угодно.
- *Подъем привилегий* (*privilege escalation*) возникает, когда пользователю с ограниченными правами доступа удается обмануть систему и получить себе права более высокого уровня. Атакующий при этом может быть законным пользователем или злоумышленником, проникшим в систему. Его конечной целью является получение

привилегий *суперпользователя*, или *администратора*, которые дают атакующему полный контроль над машиной.

- Если данные передаются в незашифрованном виде и проходят по незащищенным каналам (например, через Интернет), тогда любой встретившийся по дороге компьютер может прочесть не предназначенные для него данные – подобно прослушиванию телефонных разговоров. Разновидность этого случая носит название атаки *человек посередине* (*man-in-the-middle attack*): атакующая машина маскирует себя под участника связи и, находясь между корреспондентами, перехватывает данные их обоих.
- В каждой системе есть небольшая группа доверяющих друг другу участников. Злонамеренные законные пользователи могут сеять смуту, предоставляя другим лицам данные, для этого не предназначенные, или вводя данные, компрометирующие качество вашей компьютерной системы.

  Бороться с этим трудно. Приходится верить, что каждый пользователь ответственно подходит к тому уровню доступа к системе, который ему предоставлен. Если же пользователь не заслуживает доверия, вы не сможете исправить положение программным образом. Это показывает, что безопасность в такой же мере зависит от администрирования и политики, в какой от защищенности кода.
- Беспечные пользователи (или администраторы) могут сделать систему неоправданно открытой и уязвимой. Например:
  - Пользователи забывают разрегистрироваться; если нет таймаута для закрытия сеанса, кто угодно может позднее начать пользоваться вашей программой.
  - Многие злоумышленники применяют средства подбора паролей по словарям, которые выполняют многократные попытки зарегистрироваться в системе, пока один из паролей не сработает. Пользователи выбирают такие пароли, которые легко запомнить, но их легко и угадать. Уязвимой является любая система, которая допускает использование слабых, легко угадываемых паролей. Более надежные системы делают учетную запись пользователя временно недоступной, если несколько его попыток зарегистрироваться оказались безуспешными.
  - *Социальная инженерия* – умение получать важную информацию от людей, от находящихся в офисе предметов или даже из мусорного бачка – обычно действует гораздо проще и быстрее, чем проникновение в компьютерную систему. Человека обмануть проще, чем компьютер, и злоумышленники знают это.
  - В старых программах может обнаружиться много прорех. Многие поставщики ПО выпускают *извещения* о компрометации системы защиты и программные заплатки для них. Если администратор не следит за свежими сообщениями, система может оказаться уязвимой для атаки.

- Установка мягких прав доступа может открыть пользователям доступ к конфиденциальным частям системы – например, позволив им читать данные любых пользователей в ведомости выдачи зарплаты. Для исправления бывает достаточно изменить права доступа к файлам базы данных.
- Атаки вирусов (саморазмножающихся злонамеренных программ, обычно распространяемых через приложения к электронным письмам), трояны (скрытые злонамеренные участки в безобидных с виду программах) и шпионские программы (трояны, следящие за вашими действиями, тем, какие страницы вы посещаете, и пр.) заражают машины и могут вызывать самые разнообразные неприятности. Например, они могут с помощью регистраторов нажатий на клавиши получать пароли независимо от их сложности.
- Хранение данных «в открытом» (незашифрованном) виде – даже в оперативной памяти – небезопасно. Память не так безопасна, как кажется многим программистам; вирус или троян может просканировать память компьютера и вытащить из нее массу интересных вещей, которыми не преминет воспользоваться взломщик.

Риск увеличивается с ростом числа путей, которыми можно попасть в компьютер, количества методов ввода (доступ через Веб, командную строку или графические интерфейсы), отдельных средств ввода (различные окна, приглашения терминала, веб-формы или каналы XML) и числа пользователей (растут шансы на то, что кто-то вскроет пароль). Чем больше данных выводится, тем вероятнее проявление ошибок, имеющихся в коде вывода, приводящих к показу лишней информации.


*Чем сложнее компьютерная система, тем более вероятно, что в ее системе безопасности есть пробелы. Следовательно, пишите как можно более простое программное обеспечение!*

## Наши оппоненты

Вероятно, трудно поверить в то, что кто-то не пожалеет сил и времени, чтобы попытаться взломать ваше приложение. Но такие люди есть. Они талантливы, целеустремленны и весьма терпеливы. Если вы хотите писать защищенные программы, следует знать своего противника. Тщательно разберитесь в том, что они делают, как это делают, какими инструментами пользуются и какие задачи перед собой ставят. Только тогда вы сможете выработать правильную стратегию.

**Кто**

Атакующий может оказаться обычным жуликом, талантливым взломщиком (cracker), *script kiddie* (презрительное название взломщиков, применяющих автоматические скрипты для взлома; они используют хорошо известные уязвимости, не вкладывая в них собственное творчество), бесчестным служащим, обманывающим свою

### Как это по-научному...

В обсуждении проблем безопасности участвуют следующие важные термины:

**Пробел (flaw)**

Пробел в защите – это непреднамеренный дефект приложения. Это недостаток программы (см. «Терминология» на стр. 184). Не все пробелы представляют собой проблемы системы защиты.

**Уязвимость (vulnerability)**

Уязвимость возникает, когда из-за наличия пробела появляется возможность преодолеть защиту программы.

**Эксплойт (exploit)**

Это автоматизированное средство (или ручной метод), основанное на имеющейся в программе уязвимости и вызывающее ее непредусмотренное – и нарушающее безопасность – поведение. Не все уязвимости обнаружены или имеют эксплойты (по чистой случайности).

компанию, или бывшим служащим, мстящим за несправедливое увольнение с работы.

Взломщики (крэкеры, кракеры) хорошо подготовлены. Существует крэкерская субкультура, в которой они обмениваются информацией и простыми в эксплуатации инструментами для взлома. От того, что вы не знаете об этом, вы не станете невиннее и чище – просто окажетесь наивными и незащищенными против простейших атак.

**Где**

Благодаря широкому распространению сетей атака может прийти отовсюду – с любого континента и компьютера любого типа. Определить местонахождение злоумышленников, действующих через Интернет, очень трудно; обычно они умеют хорошо заметать следы. Часто сначала взламывают простые машины, чтобы воспользоваться ими как прикрытием для более дерзких атак.

**Когда**

Атака может произойти в любое время суток. Если вы на разных континентах с атакующим, то когда у одного из вас день, у другого ночь. Защищенные программы нужны вам в любое время суток, а не только в рабочее время.

**Зачем**

Потенциальных взломщиков так много, что и мотивы для атаки у них могут быть самыми разными. Цели могут быть преступными (нанести удар вашей компании по политическим мотивам или получить доступ к вашему банковскому счету) или хулиганскими (ка-

кой-нибудь шутник-соученик хочет поместить что-то забавное на вашем сайте). Возможно проявление любопытства (хакеру просто интересно узнать, как устроена ваша сеть, или попрактиковаться в своем искусстве взлома) или авантюризма (пользователь натыкается на данные, которые ему не полагается знать, и пытается извлечь из них какую-нибудь выгоду).

В мире, где почти все компьютеры объединены в одну сеть, обычно узнаешь о том, кто твой враг, уже после того, как он ударит. И даже тогда можно не узнать, кто это был; вашего искусства криминальной экспертизы может оказаться недостаточно, чтобы по еще теплой кучке цифровых останков восстановить картину происшедшего. Но, как юный пионер: *будь готов!* Не проходи мимо уязвимостей и не думай, что твоя машина никому не нужна; всегда кто-нибудь ей заинтересуется.


*Не проходите мимо уязвимостей и не считайте себя непобедимым. Где-нибудь обязательно найдется тот, кто попытается применить эксплойт к вашему коду.*

## Крэкеры и хакеры

Эти два термина часто путают и используют неверным образом. Вот их правильное определение:

**Крэкер**

Тот, кто умышленно пользуется уязвимостями в компьютерных системах для получения несанкционированного доступа.

**Хакер**

Часто некорректно применяется для обозначения крэкера, но в действительности это тот, кто занимается хакерством – работой с кодом. Это имя в 70-х годах с гордостью носил определенный тип знатоков программирования. Хакер – это компьютерный специалист или энтузиаст. Употребляются еще два хакерских термина:

**Герои**

Хакеры «в белых шляпах» думают о том, к каким последствиям приводит их работа, осуждают действия крэкеров и неэтично ведущих себя пользователей компьютеров. Они считают, что трудятся на благо общества.

**Разбойники**

Это темные личности среди программистов, которые получают удовольствие от злоупотребления компьютерными системами. Это крэкеры, активно занимающиеся поиском путей непорядочного использования систем. Они не уважают прав других людей на собственность или личную жизнь.

# Оправдания, оправдания

Как же атакующим удается так часто взламывать код? Они располагают оружием, которого у нас нет или о котором мы по недостатку образования ничего не знаем. Инструменты, знания, мастерство – все это работает на них. Однако они располагают еще одним основополагающим преимуществом – временем. В условиях промышленного производства программного продукта разработчики вынуждены вырабатывать максимальный объем кода, который только доступен их силам (или даже больше), и делать это в сжатые сроки, в противном случае последствия известны. Этот код должен удовлетворять определенным требованиям (функциональности, юзабилити, надежности и т. п.), а это оставляет крайне мало времени на заботу о «второстепенных» качествах, таких как защищенность. У атакующих таких проблем нет; у них достаточно времени, чтобы изучить все хитрости вашей системы, и они научились нападать с разных сторон.

Условия игры им благоприятствуют. Мы, разработчики программ, должны защитить все мыслимые места проникновения в систему; атакующий может выбрать самое слабое место и сосредоточить на нем свои усилия. Мы можем установить защиту только против известных эксплойтов; атакующий может, не жалея времени, отыскивать новые, неизвестные уязвимости. Мы должны постоянно быть настороже, ожидая нападения; атакующий может напасть, когда ему заблагорассудится. Мы обязаны писать хорошие и понятные программы, которые корректно взаимодействуют с окружающим миром; атакующий может поступать самым бессовестным образом.

Безопасность ПО ставит перед бедным, замученным программистом бесчисленное множество других, также важных проблем и задач. И что из этого следует? То, что мы *должны* их переиграть. Мы должны быть лучше информированы, лучше вооружены, лучше знать своего противника и лучше уметь писать код. Вопросы защищенности должны быть с самого начала учтены в проекте, им должно быть отведено должное место в процедуре разработки и в графиках работ.

# Ощущение незащищенности

Задача программиста во всей этой сумятице – написать защищенный код, поэтому проведем обзор слабых мест в наших программах, чтобы определить, куда направить свои усилия. Существуют конкретные виды уязвимостей в коде – брешей, которыми может воспользоваться атакующий.

## Опасный проект и архитектура

Это самая фундаментальная слабость, поэтому ее труднее всего исправить. Если вопросам безопасности не уделено должное внимание на

уровне архитектуры, то нарушения защиты будут преследовать вас всюду: передача незашифрованных данных в открытых сетях, хранение данных на легкодоступных носителях и пользование программными сервисами, пробелы в защите которых общеизвестны.

Вопросы безопасности нужно иметь в виду с самого начала разработки. Каждая компонента системы должна быть изучена с точки зрения наличия в ней незащищенных мест; безопасность компьютерной системы определяется надежностью ее самой слабой части, которой может оказаться даже не ваш код. Например, программа на Java не может быть более защищенной, чем JVM, на которой она выполняется.

## Переполнение буфера

Большинство приложений обращено лицом к публике: они слушают открытый сетевой порт или обрабатывают данные, введенные через веб-броузер или графический интерфейс. Эти процедуры ввода данных – главное место, где происходит нарушение защиты.

В программах на C входные данные часто обрабатываются с помощью функции `sscanf`. Несмотря на то что эта функция принадлежит стандартной библиотеке C и регулярно встречается в C-коде, она совершенно беззастенчиво предлагает возможности для написания опасного кода.[1]

Можно встретить такой код:

```
void parse_user_input(const char *input)
{
    /* синтаксический анализ введенной строки input */
    int my_number;
    char my_string[100];
    sscanf(input, "%d %s", &my_number, my_string);
    ... теперь работа с ней ...
}
```

Вы заметили бросающуюся в глаза проблему? Некорректная входная строка – длина которой больше 100 символов – выйдет за границу буфера `my_string` и запишет произвольные данные по недопустимым адресам памяти.

Результат зависит от того, как используются испорченные адреса памяти. В некоторых случаях поведение программы никак не изменится, и тогда вам весьма повезет.[2] Иногда программа продолжает работать, но с малозаметными отличиями – их бывает трудно заметить и трудно найти причину. Иногда программа аварийно завершается, часто попутно

[1] Этот пример написан на C и характерен для C-кода, но область действия такого типа эксплойта далеко не ограничивается C.

[2] Если взглянуть с другой стороны, то вам очень не повезло. Вы не заметили дефект во время тестирования, поэтому он сохранится в окончательной версии, и там-то взломщик им и воспользуется.

вызывая сбой важных системных компонент. Но хуже всего, если проникшие в память данные окажутся в области кода, выполняемого ЦП. Организовать это не столь сложно, а в результате атакующий сможет выполнить на вашей машине произвольный код и при удаче получит к ней полный доступ.

Проще всего воспользоваться переполнением буфера, когда он располагается в стеке, как в описанном выше случае. Тогда появляется возможность перенацелить ЦП, заменив хранящийся в стеке адрес возврата из функции. Однако эксплойты могут быть созданы и для тех случаев, когда буфер размещается в куче.

## Встроенные строки запросов

Данный тип атаки может быть применен для аварийного завершения программы, выполнения произвольного кода или несанкционированного получения данных. Как и в случае переполнении буфера, в основе атаки лежит неверный анализ входных данных, но при этом вместо прорыва за границы буфера используются действия, которые программа последовательно выполняет над непроверенными входными данными.

Классическим примером этого вида атак в C-программах является использование *форматной строки*. Обычно эксплуатируется функция `printf` или ее разновидности, и происходит это так:

```
void parse_user_input(const char *input)
{
     printf(input);
}
```

Злоумышленник подает на вход строку данных, содержащую маркеры формата `printf` (такие как `%s` и `%x`), и принуждает программу напечатать данные из стека или даже из адресов в памяти в зависимости от конкретного формата обращения к `printf`. Атакующий может также записать произвольные данные в память с помощью аналогичного приема (и маркера формата `%n`).

Решение проблемы отыскивается без труда. Запись оператора печати в виде `printf("%s", input)` гарантирует, что `input` не будет интерпретироваться в качестве форматной строки.

Есть много других случаев, когда встроенный запрос может быть использован злонамеренно. Например, можно скрыто ввести в приложение базы данных произвольный оператор SQL, который найдет в базе данные, нужные взломщику.

Другой вариант эксплойта для слабых веб-приложений известен как *кросс-сайтовый скриптинг* из-за способа прохождения атаки через систему: данные, введенные атакующим, пройдя через веб-приложение, воздействуют на веб-броузер жертвы. Специальный комментарий, помещенный атакующим в веб-систему обмена сообщениями, будет показан всеми броузерами, просматривающими данную страницу.

Если в сообщении есть скрытый код JavaScript, броузеры выполнят его незаметно для пользователей.

## Условия гонки

Существует возможность эксплойта для систем, зависящих от скрытого порядка событий, с целью вызвать их нештатное поведение или аварийное завершение. Обычно это допускают системы со сложными моделями потоков или составленные из многочисленных взаимодействующих процессов.

Многопоточная программа может использовать общий пул памяти для своих рабочих потоков. Если не принять должных мер, то один поток может прочесть из буфера информацию, которую другой поток записал туда, но еще не собирался делать доступной остальным, например часть привилегированной транзакции или данные другого пользователя.

Эта проблема свойственна не только многопоточным приложениям. Рассмотрим следующий фрагмент кода C для UNIX. Его задача – записать некоторые данные в файл, а потом изменить права доступа к этому файлу.

```
fd = open("filename");            /* создать файл */
/* точка A (см. ниже) */
write(fd, some_data, data_size);  /* записать данные */
close(fd);                        /* закрыть файл */
chmod("filename", 0777);          /* назначить ему права доступа */
```

Существует ситуация гонки, которой может воспользоваться атакующий. Удалив файл в точке `A` и заменив его ссылкой на собственный файл, атакующий получит для своего файла специальные права доступа. Этим можно воспользоваться для дальнейшего проникновения в систему.

## Целочисленное переполнение

Небрежное применение математических конструкций может привести к передаче программой управления неожиданными способами. Целочисленное переполнение происходит, когда тип переменной не позволяет представить результат арифметической операции. Применение беззнакового 8-разрядного целого типа (`uint8_t`) приводит к ошибке в следующем вычислении на C:

```
uint8_t a = 254 + 2;
```

Результатом будет 0, а не 256, как вы рассчитывали; 8 разрядов могут содержать числа, не превышающие 255. Атакующий может послать на вход очень большие числа, чтобы спровоцировать переполнение и вызвать непредвиденные результаты. Нетрудно видеть, что в итоге могут возникнуть серьезные проблемы; следующий код C может привести к переполнению кучи из-за целочисленного переполнения:

```
void parse_user_input(const char* input)
{
    uint8_t length = strlen(input) + 11; /* uint8_t может переполниться */
    char *copy = malloc(length);         /* этого может быть недостаточно */
    if (copy)
    {
        sprintf(copy, "Input is: %s", input);
        /* этот буфер может переполниться */
    }
}
```

Конечно, `uint8_t` едва ли будет выбран для переменной, содержащей длину строки, но точно такие же проблемы возникают с типами больших данных. При нормальной работе они маловероятны, но эксплойт на них построить можно.

Те же проблемы возникают при вычитании (тогда это называется *целочисленной потерей* (*integer underflow*), при использовании знаковых и беззнаковых чисел в присваивании, при некорректном приведении типов и при умножении или делении.

## Дела защитные

*Чем больше вы стремитесь к безопасности,*
*тем менее защищенным вы оказываетесь.*

Брайан Трейси

Мы уже видели, что создание программного обеспечения напоминает строительство здания (см. «Действительно ли мы собираем программы?» на стр. 240 и главу 14). Мы должны научиться защищать свои программы точно так же, как мы защищаем дом, закрывая все окна и двери, нанимая сторожа и устанавливая дополнительные механизмы защиты (типа охранной сигнализации, электронных карточек доступа, значков с личными данными и т. п.). И тем не менее нужно постоянно поддерживать бдительность: несмотря на все хитрые замки, можно забыть закрыть дверь или включить охранную сигнализацию.

Стратегии защиты программного обеспечения применимы на разных уровнях:

***Установка* системы**

Конкретная конфигурация ОС, инфраструктура сети, номера версий всех работающих приложений имеют важное значение для состояния безопасности.

***Конструктивные* особенности программной системы**

Необходимо принять правильные конструктивные решения, например, относительно возможности для пользователя оставаться зарегистрированным в системе произвольно долгое время, способов связи между подсистемами, выбора протоколов.

***Реализация* программы**

В ней не должно быть дефектов. Наличие ошибок в коде может быть причиной уязвимости системы.

***Процедура* эксплуатации системы**

При неправильном использовании любая система может представлять собой угрозу. По возможности этому должно препятствовать правильное проектирование, но пользователям нужно объяснить, какие их действия могут привести к проблемам. Ведь сколько людей записывают свое имя и пароль на бумажке, которую кладут рядом с терминалом!

Всегда трудно создавать защищенные системы. При этом неизбежен компромисс между безопасностью и функциональностью. Чем более защищена система, тем менее удобно ею пользоваться. Самая защищенная система – та, которая ничего не вводит и не выводит; ее не атакуешь ни с какой стороны. Однако от такой системы мало пользы. В простейшей системе нет никакой авторизации и каждому позволено делать все, что угодно; но она совершенно не защищена. Необходимо выбирать нечто среднее. Выбор определяется природой приложения, уровнем его секретности и оценкой опасности подвергнуться нападению. Чтобы писать код, обеспечивающий достаточную степень защиты, необходимо ясно представлять, какие *требования безопасности* предъявляются к системе.

Подобно тому как вы предпринимаете некоторые меры для защиты здания, вы применяете некоторые технологии для защиты своего программного обеспечения от злоумышленников.

## Технология установки системы

Каким бы хорошим ни было ваше приложение, но если система, в которой оно устанавливается, не защищена, ваша программа тоже уязвима. Даже самое надежное приложение должно работать в некоторой операционной среде: под конкретной ОС, на конкретном аппаратном устройстве, в сети и с определенной группой пользователей. Атакующий с таким же успехом может скомпрометировать компоненты среды, как и ваш код.

- Не запускайте на своем компьютере программы из ненадежных источников, потенциально опасные.

  Возникает вопрос: что может служить основанием для доверия или недоверия некой программе? Можно провести анализ исходного кода программного обеспечения, если он у вас есть, и убедиться в его корректности (если у вас есть такая склонность). Можно взять программное обеспечение, которым пользуются все, и считать, что вы застрахованы количеством пользователей. (Однако если в таком программном обеспечении будет обнаружена уязвимость, и вам, и всем остальным нужно обновить пакет.) Либо можно выбрать программу

у производителя с хорошей репутацией в надежде, что она послужит надежным показателем.


*Запускайте на своем компьютере только программы, полученные из надежных источников. Установите четкую политику для определения надежности источников.*

- Применяйте защитные технологии, например сетевые экраны и фильтры спама и вирусов. Не оставляйте потайных входов, которые могут быть обнаружены взломщиками.
- Будьте готовы к тому, что злоумышленники могут оказаться среди авторизованных пользователей, ведите регистрацию того, кто, что и когда делал в системе. Периодически делайте резервные копии всех данных, чтобы фальшивая их модификация не уничтожила все ваши прежние труды.
- Установите минимальное количество способов входа в систему, дайте каждому пользователю минимальный набор прав и сократите, если можете, число пользователей, допущенных к системе.
- Правильно настройте систему. В некоторых ОС по умолчанию устанавливается очень слабый уровень безопасности, просто зазывающий взломщиков войти в систему. Если у вас такая система, нужно научиться устанавливать на ней защиту в полном объеме.
- Установите *ловушку*: машину-приманку, которую взломщикам будет легче найти, чем ваши реальные системы. Если она будет выглядеть достаточно привлекательно, атакующие потратят свои силы на ее взлом, а ваши критически важные машины останутся вне их внимания. Если вы обнаружите компрометацию ловушки, постарайтесь дать отпор взломщику, прежде чем он доберется до ваших ценных данных.

## Технология конструирования программного обеспечения

Необходимо начинать защиту программного обеспечения уже на этой ранней стадии. Если вы займетесь внесением в код модификаций, призванных обеспечить его защищенность, лишь на поздних стадиях цикла разработки, вас ждет неудача. Защита должна быть фундаментальной частью архитектуры и конструкции вашей системы.


*Безопасность – важный аспект архитектуры любого программного продукта. Будет ошибкой не позаботиться о ней на ранних стадиях разработки.*

Чем проще конструкция программного обеспечения, тем меньше в нем точек, доступных для атаки, и тем легче обеспечить его защиту. Естественно, что в более сложных конструкциях составные части взаимодействуют активнее, а потому в них больше мест, которые могут подвергнуться атаке взломщика. Если вы принадлежите к тем 99,9% программистов, которые не могут позволить себе запускать свои программы на

опечатанной машине, стоящей в бункере, находящемся в засекреченной точке пустыни, то вам следует позаботиться о том, чтобы сделать свою конструкцию возможно проще.

Проектируя код, подумайте над тем, как активно помешать кому бы то ни было воспользоваться вашим приложением непредусмотренным способом. Вот несколько полезных стратегий для достижения этого:

- Ограничьте в проекте количество точек ввода данных и направьте весь обмен данными через отдельную часть системы. В результате атакующий не сможет гулять по всему вашему коду, а будет ограничен одним (защищенным) узким местом. Он будет действовать лишь в некоем дальнем углу, а вы сможете сосредоточить в том месте свои усилия по защите.[1]
- Выполняйте все программы с минимально возможными правами доступа. Не запускайте программу от имени системного администратора, если это не является совершенно необходимым, а тогда примите *особые* меры предосторожности. Особенно важно это для программ UNIX с атрибутом setuid – их может запустить любой пользователь, но после запуска они получают в системе особые права.
- Избегайте функций, в которых нет реальной надобности. Вы сократите не только время разработки, но и шансы появления в программе ошибок – для них в программе останется меньше места. Чем ниже сложность кода, тем меньше шансы появления в нем опасных мест.
- Не основывайте свой код на опасных библиотеках. Опасной будет всякая библиотека, про которую вы не знаете, что она безопасна. Например, большинство библиотек GUI разработано без учета безопасности, поэтому не пользуйтесь ими в программе, выполняющейся с правами суперпользователя.


*Проектируя программу, рассчитывайте только на известные, защищенные компоненты сторонних разработчиков.*

- Если среда выполнения предоставляет средства обеспечения защиты, используйте их в своем коде. Например, среда .NET располагает инфраструктурой, которая позволяет, например, проверить, что вызывающий код подписан доверенным третьим лицом. Это не решает всех проблем (секретный ключ компании может оказаться скомпрометированным), и нужно уметь правильно применять такую технологию, но все же она способствует созданию защищенных программ.
- Не храните секретные данные. Если это все же необходимо, зашифруйте их, чтобы защитить от любопытствующих. При работе с сек-

[1] Разумеется, все не так просто. Переполнение буфера может произойти в любой части кода, и бдительность нужно проявлять постоянно. Все же большинство уязвимостей возникает в местах ввода в программу данных или поблизости от них.

ретными данными тщательно следите за тем, куда вы их помещаете; блокируйте страницы памяти с секретной информацией, чтобы администратор виртуальной памяти ОС не сбрасывал их на жесткий диск, откуда их сможет прочесть злоумышленник.

- Если пользователь вводит конфиденциальные данные, обращайтесь с ними осторожно. Не показывайте вводимые пароли.

Самая неудачная стратегия безопасности известна как *запутанность как защита* (*security through obscurity*), и она наиболее распространена. Ее цель – спрятать все детали конструкции и реализации, чтобы никто не смог разобраться, как код работает, а потому и придумать, как его взломать. При этом стараются сделать критически важные компьютерные системы как можно менее заметными в надежде, что атакующий их не обнаружит.

Это порочный путь. В один прекрасный день ваша система *будет* обнаружена, а через какое-то время *атакована*.

Не всегда такое решение принимается сознательно. Просто такой прием оказывается удобен, если вы вообще не занимались проблемами защиты во время проектирования системы – причем удобен до того момента, когда кто-нибудь *скомпрометирует* вашу систему. После этого начинается совсем другая история.


*Готовьтесь к тому, что ваша система будет атакована, и проектируйте с учетом этого все ее части.*

## Технологии реализации кода

Так будет ли ваша программа защищена от взлома, если надежно спроектировать систему? К сожалению, это не так. Мы уже видели, как недостатки кода позволяют создавать эксплойты, дающие возможность повернуть выполнение программы непредвиденным образом.

Код, который мы пишем, – это передовая линия фронта, путь, которым атакующий чаще всего пытается воспользоваться для проникновения в систему и где разворачиваются основные сражения. Если система спроектирована плохо, то самый хороший код окажется уязвимым для атаки; но и на фундаменте тщательно продуманной архитектуры мы должны возвести крепкие защитные стены с помощью безопасного кода. Правильный код не всегда оказывается безопасным.

- Защитное программирование – основная технология разработки надежного кода. Его главный принцип – *не принимай ничего на веру* – как нельзя лучше соответствует задачам создания безопасного программного обеспечения. Паранойя – это благо, поскольку никогда нельзя надеяться на то, что пользователи станут взаимодействовать с вашей программой так, как вы от них ожидаете.

  Соблюдение простых правил защиты, таких как «проверяй *все* входные данные» (включая данные, введенные пользователем, ко-

манды запуска программ и переменные окружения) и «проверяй *все* вычисления», позволит избавиться от массы уязвимых мест в вашей программе.

- Осуществляйте *аудит безопасности*. Имеется в виду тщательное рецензирование исходного кода экспертами в области безопасности. При обычном тестировании вы редко обнаружите пробелы в защите; они проявляются при необычном сочетании условий, которое не придет в голову составителю тестов (например, переполнение буфера при вводе очень длинных последовательностей символов).
- С осторожностью порождайте дочерние процессы. Если атакующий сумеет перевести выполнение подзадачи в нужном ему направлении, он сможет получить контроль над любыми средствами. Пользуйтесь в коде C функцией `system`, только если не остается ничего другого.
- Тестируйте и отлаживайте программы нещадно. Давите ошибки со всей строгостью. Не пишите код, который допускает аварийное завершение; он может остановить всю систему.
- Заключите все операции в атомарные транзакции, чтобы атакующие не могли воспользоваться состоянием гонки в своих целях. Пример с `chmod` в разделе «Условия гонки» на стр. 305 можно поправить, применив `fchmod` к дескриптору открытого файла вместо `chmod` к имени файла: даже если атакующий заменит файл, вы точно знаете, какой файл нужно модифицировать.

### Технологии процедуры

В основном здесь требуются тренировка и обучение, хотя желательно отобрать пользователей, которые не слишком беспомощны (если вы можете себе такое позволить).

Пользователей нужно научить приемам безопасной работы: не сообщать никому свой пароль, не устанавливать какое попало программное обеспечение вместе с критически важным и работать на машине только в соответствии с инструкциями. Однако ошибки бывают даже у самых старательных пользователей. При проектировании систем мы стремимся минимизировать риск таких ошибок и надеемся, что их последствия окажутся не слишком тяжелыми.

## Резюме

*Безопасность – это в своем роде смерть.*

Теннесси Уильямс

Программирование – это война.

Безопасность – серьезная проблема в разработке современного программного обеспечения; невозможно спрятать голову в песок, чтобы не замечать ее. Страусы пишут плохой код. С брешами в системе без-

опасности можно бороться с помощью более тщательного проектирования и совершенствования архитектуры системы, а также лучшего информирования о существующих проблемах. Слишком велики риски, чтобы позволить себе не думать о защищенности систем.

**Хорошие программисты...**

- Разбираются в требованиях к безопасности в каждом проекте, над которым трудятся
- Инстинктивно пишут код, в котором нет стандартных уязвимых мест
- Учитывают требования безопасности при проектировании каждой системы, а не приступают к защите, когда продукт почти готов
- Располагают стратегией тестирования безопасности

**Плохие программисты...**

- Не занимаются проблемами защищенности, считая их малозначительными
- Считают себя экспертами в области безопасности (*настоящих экспертов* очень мало)
- Задумываются о недостаточной защищенности своих программ только при обнаружении уязвимостей или, еще хуже, в случае компрометации своего кода
- Думают о безопасности, только когда пишут код, игнорируя ее на уровне проектирования и архитектуры

## См. также

**Глава 1. Держим оборону**

*Защитное программирование* – важная технология для написания безопасного кода.

**Глава 8. Время испытаний**

Необходимо строго тестировать наше программное обеспечение на предмет его безопасности.

**Глава 13. Важность проектирования**

Безопасность необходимо иметь в виду при проектировании каждого раздела кода.

**Глава 14. Программная архитектура**

Безопасность – одно из фундаментальных требований к архитектуре компьютерной системы. Ее проектированием нужно заниматься с самого начала.

# Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 638.


## Вопросы для размышления

1. Что такое «безопасная» программа?
2. Какие входные данные нужно проверять в безопасной программе? Какого типа проверка необходима?
3. Как можно защищаться против атак со стороны группы доверенных пользователей?
4. Где может произойти переполнение буфера, допускающее создание эксплойта? Какие функции особенно подвержены переполнению буфера?
5. Можно ли полностью исключить возможность переполнения буфера?
6. Как защитить память, используемую приложением?
7. Характерна ли для C и C++ принципиально более низкая защищенность, чем для других языков?

8. Учтен ли опыт C для проектирования C++ как более защищенного языка?
9. Как можно узнать, что ваша программа скомпрометирована?

## Вопросы личного характера

1. Каковы требования к безопасности, предъявляемые в вашем текущем проекте? Каким образом были они установлены? Кто осведомлен о них? В каких документах они отражены?
2. Какой была самая страшная ошибка защиты в выпущенных вами приложениях?
3. Сколько бюллетеней по вопросам безопасности было выпущено по поводу вашего приложения?
4. Проводили ли вы *аудит безопасности* программы? Какие проблемы он выявил?
5. Кто, как вам кажется, вероятнее всего может попытаться атаковать вашу систему? В какой мере на это влияют:
   a. Ваша компания
   b. Тип пользователя
   c. Тип продукта
   d. Популярность продукта
   e. Конкуренция
   f. Платформа, на которой он работает
   g. Нахождение в сети и видимость системы широкой публике