КРИС КАСПЕРСКИ

ТЕХНИКА ОТЛАДКИ ПРОГРАММ

БЕЗ ИСХОДНЫХ ТЕКСТОВ

ОСНОВНОЙ ИНСТРУМЕНТАРИЙ ХАКЕРА

ВЗЛОМ ПРОГРАММ С ЗАКРЫТЫМИ ГЛАЗАМИ

"ПРОТИВОУГОННЫЕ" СИСТЕМЫ СВОИМИ РУКАМИ

БОРЬБА С КРИТИЧЕСКИМИ ОШИБКАМИ ПРИЛОЖЕНИЙ

ВНЕДРЕНИЕ И УДАЛЕНИЕ ВИРУСНОГО КОДА ИЗ PE-ФАЙЛОВ

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

УДК 681.3.06
ББК 32.973.26-018.1
К28

Касперски К.

К28 Техника отладки программ без исходных текстов. — СПб.: БХВ-Петербург, 2005. — 832 с.: ил.

ISBN 5-94157-229-8

Даны практические рекомендации по использованию популярных отладчиков, таких как NuMega SoftIce, Microsoft Visual Studio Debugger и Microsoft Kernel Debugger. Показано, как работают отладчики и как противостоять дизасемблированию программы. Описаны основные защитные механизмы коммерческих программ, а также способы восстановления и изменения алгоритма программы без исходных текстов. Большое внимание уделено внедрению и удалению кода из PE-файлов. Материал сопровождается практическими примерами.

Компакт-диск содержит исходные тексты приведенных листингов и полезные утилиты.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.08.05.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 67,08.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-229-8

© Касперски К., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Предисловие ... 1

Об авторе ... 1
О чем и для кого эта книга ... 3

Введение ... 11

История хакерства ... 11
История происхождения термина "хакер" ... 14
Психология хакера ... 16
Лаборатория искусственного интеллекта и PDP-1 ... 20
Сеть ... 23
Си и UNIX ... 26
Конец хакеров шестидесятых ... 33
RSX-11M ... 36
Intel ... 37
Хаос ... 38
Бытовой компьютер восьмидесятых ... 40
Рождение современных хакеров, или снова Intel ... 41

Глава 1. Знакомство с отладочными инструментами ... 45

1.1. Как работает отладчик ... 48
Обработка исключений ... 50
1.2. Что нам понадобится ... 51
1.3. Особенности отладки в UNIX ... 53
PTrace — фундамент для GDB ... 56
PTrace и ее команды ... 58
Поддержка многопоточности в GDB ... 60
Краткое руководство по GDB ... 61
Трассировка системных функций ... 66
Интересные ссылки ... 67

1.4. Эмулирующие отладчики и эмуляторы ... 68
Минимальные системные требования ... 70
Выбирай эмулятор себе по руке! ... 71
1.5. Обзор эмуляторов ... 74
DOSBox ... 74
Bochs ... 76
Microsoft Virtual PC ... 77
VMware ... 79
Сводная таблица характеристик эмуляторов ... 80
Разные мелочи ... 81
1.6. Области применения эмуляторов ... 82
Пользователям ... 82
Администраторам ... 83
Разработчикам ... 84
Хакерам ... 87
Как настроить SoftIce под VMware ... 89
Экзотические эмуляторы ... 89
1.7. Кратко об эмуляции процессора ... 90
1.8. BoundsChecker ... 96
Быстрый старт ... 98
Подключение нестандартных DLL ... 101
Пункты меню ... 103
1.9. Хакерские инструменты под UNIX ... 107
Отладчики ... 107
Дизассемблеры ... 111
Шпионы ... 112
Шестнадцатеричные редакторы ... 114
Дамперы ... 115
Автоматизированные средства защиты ... 115

Глава 2. Защитные механизмы и их отладка ... 119

2.1. Классификация защит по роду секретного ключа ... 120
2.2. Создаем защиту и пытаемся ее сломать ... 123
2.3. От exe до crk ... 125
2.4. Знакомство с отладчиком ... 142
Бряк на оригинальный пароль ... 143
Прямой поиск введенного пароля в памяти ... 160
Бряк на функции ввода пароля ... 170
Бряк на сообщения ... 173
Механизм сообщений в Windows 9*x* ... 176
2.5. На сцене появляется IDA ... 177
2.6. Дизассемблер & отладчик в связке ... 209
Из языка IDA-Си ... 212

2.7. Дао регистрационных защит ... 216
Как узнать имя функции по ординалу ... 221
Как сделать исполняемые файлы меньше ... 248
Перехват *WM_GETTEXT* ... 249
2.8. Хеширование и его преодоление ... 251
2.9. Ограничение возможностей ... 267
2.10. Ограничение времени использования ... 286
2.11. Ограничение числа запусков ... 291
2.12. NagScreen ... 293
2.13. Ключевой файл ... 302

Глава 3. Противостояние отладке ... 315

3.1. Обзор способов затруднения анализа программ ... 317
3.2. Приемы против отладчиков реального режима ... 319
3.3. Приемы против отладчиков защищенного режима ... 335
3.4. Как противостоять трассировке ... 348
3.5. Как противостоять контрольным точкам останова ... 354
Несколько грязных хаков, или как не стоит защищать свои программы ... 362
Серединный вызов API-функций ... 363
Вызов API-функций через мертвую зону ... 382
Копирование API-функций целиком ... 385
Как обнаружить отладку средствами Windows ... 388
3.6. Антиотладочные приемы под UNIX ... 389
Паразитные файловые дескрипторы ... 390
Аргументы командной строки и окружение ... 391
Дерево процессов ... 392
Сигналы, дампы и исключения ... 392
Распознавание программных точек останова ... 393
Мы трассируем, нас трассируют ... 394
Прямой поиск отладчика в памяти ... 395
Измерение времени выполнения ... 396
3.7. Основы самомодификации ... 396
Проблемы обновления кода через Интернет ... 409
3.8. Неявный самоконтроль как средство создания неломаемых защит ... 411
Техника неявного контроля ... 413
Практическая реализация ... 415
Как это ломают? ... 425
3.9. Ментальная отладка и дизассемблирование ... 435
Маленькие хитрости ... 459
3.10. Ментальное ассемблирование ... 460
3.11. Краткое руководство по защите ПО ... 465
Джинн из бутылки, или недостатки решений из коробки ... 466

Защита от копирования, распространения серийного номера 466
Защита испытательным сроком 467
Защита от реконструкции алгоритма 467
Защита от модификации на диске и в памяти 469
Антидизассемблер 470
Антиотладка 470
Антимонитор 471
Антидамп 471
Как защищаться 473
Мысли о защитах 474
Противодействие изучению исходных текстов 474
Противодействие анализу бинарного кода 478
3.12. Как сделать свои программы надежнее? 481
Причины и последствия ошибок переполнения 482
Переход на другой язык 483
Использование кучи для создания массивов 484
Отказ от индикатора завершения 484
Обработка структурных исключений 485
Традиции и надежность 487
Как с ними борются? 488
Поиск уязвимых программ 489
Неудачный выбор приоритетов в Си 493
3.13. Тестирование программного обеспечения 495
Тестирование на микроуровне 497
Регистрация ошибок 498
Бета-тестирование 499
Вывод диагностической информации 502
Верификаторы кода языков Си/Си++ 504
Демонстрация ошибок накопления 505

Глава 4. Примеры реальных взломов 509

4.1. Intel C++ 5.0.1 compiler 510
4.2. Intel Fortran 4.5 518
4.3. Intel C++ 7.0 compiler 523
4.4. Record Now 532
4.5. Alcohol 120% 535
4.6. UniLink v1.03 от Юрия Харона 549
UniLink v1.03 от Юрия Харона II, или переходим от штурма к осаде 571
Entry Point и ее окружение 572
Передача управления по структурному исключению 574
Внутри обработчика 581
Таинства stealth-импорта API-функций,
или как устроена *HaronLoadLibrary* 586

Таинства stealth-импорта: как устроена *HaronGetProcAddress* 589
Таинства *IsDebuggerPresent* 599
Таинства загрузки USER32.DLL и ADVAAPI32.DLL 601
Конец таинств, или где тот trial, который expired 604
4.7. ARJ 613
4.8. AVPVE: разбор полетов 614
Формат файлов (общее) 615
Формат файла языковой поддержки avp.lng 616
Формат файлов HLP 617
4.9. Bounds-Checker 5 620
4.10. CD-MAN EGA Version 621
4.11. F-PROT 2.19 622
4.12. FDR 2.1 627
4.13. HEXEDIT.EXE Version 1.5 629
4.14. SGWW password protection WhiteEagle 630
4.15. SOURCER 5.10 631
4.16. Emulated Solar CPU 632
4.17. POCSAG 32 635

Глава 5. Критические ошибки приложений и операционной системы 639

5.1. Приложения, недопустимые операции и все-все-все 640
Доктор Ватсон 642
Отладчик Microsoft Visual Studio Debug 649
5.2. Обитатели сумеречной зоны, или из морга в реанимацию 650
Принудительный выход из функции 651
Раскрутка стека 654
Передача управления на функцию обработки сообщений 658
5.3. Как подключить дамп памяти 666
Восстановление системы после критического сбоя 676
Подключение дампа памяти 677

Глава 6. Формат PE-файлов 683

6.1. Особенности структуры PE-файлов в конкретных реализациях 684
6.2. Общие концепции и требования, предъявляемые к PE-файлам 686
6.3. Структура PE-файла 689
6.4. Что можно и что нельзя делать с PE-файлом 692
6.5. Описание основных полей PE-файла 695
[old-exe] e_magic 695
[old-exe] e_cparhdr 695
[old-exe] e_lfanew 696
[image_file_header] Machine 696
[image_file_header] NumberOfSections 696
[image_file_header] PointerToSymbolTable/NumberOfSymbols 697

[image_file_header] SizeOfOptionalHeader 697
[image_file_header] Characteristics 698
[image_optional_header] Magic 700
[image_optional_header] SizeOfCode/SizeOfInitializedData/ SizeOfUninitializedData 700
[image_optional_header] BaseOfCode/BaseOfData 701
[image_optional_header] AddressOfEntryPoint 701
[image_optional_header] ImageBase 702
[image_optional_header] FileAlignment/SectionAlignment 702
[image_optional_header] SizeOfImage 703
[image_optional_header] SizeOfHeaders 703
[image_optional_header] CheckSum 704
[image_optional_header] Subsystem 704
[image_optional_header] DllCharacteristics 705
[image_optional_header] SizeOfStackReserve/SizeOfStackCommit, SizeOfHeapReserve/SizeOfHeapCommit 706
[image_optional_header] NumberOfRvaAndSizes 706
DATA_ DIRECTORY 707
Таблица секций 709
Экспорт 714
Импорт 718
Перемещаемые элементы 727

Глава 7. Техника внедрения и удаления кода из PE-файлов 731

7.1. Понятие X-кода и другие условные обозначения 732
7.2. Цели и задачи X-кода 733
7.3. Требования, предъявляемые к X-коду 736
7.4. Внедрение 737
Предотвращение повторного внедрения 738
Классификация механизмов внедрения 740
Категория A: внедрение в пустое место файла 741
Внедрение в регулярную последовательность байтов 750
Категория A: внедрение путем сжатия части файла 756
Категория A: создание нового NTFS-потока внутри файла 758
Категория B: растяжение заголовка 761
Категория B: сброс части секции в оверлей 763
Категория B: создание своего собственного оверлея 767
Категория C: расширение последней секции файла 768
Категория C: создание своей собственной секции 771
Категория C: расширение серединных секций файла 773
Категория Z: внедрение через автозагружаемые DLL 776

ПРИЛОЖЕНИЯ 777

Приложение 1. Разгон и торможение Windows NT 779

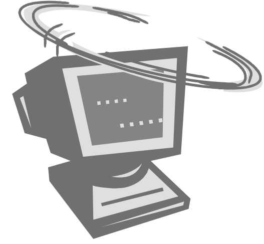
Структура ядра 780
Типы ядер 782
Почему непригодны тестовые пакеты 784
Обсуждение методик тестирования 786
  Разность таймеров 787
  Синхронизация 791
  ACPI и IRQ 792
  Переключение контекста 796
  Длительность квантов 802
Обсуждение полученных результатов 805

Приложение 2. Практические советы по восстановлению системы в боевых условиях 807

Аппаратная часть 808
  Оперативная память 809
  Блок питания 811
  И все-все-все 812

Приложение 3. Описание компакт-диска 815

Предметный указатель 817



Глава 1

Знакомство с отладочными инструментами

...Отладка подобна охоте или рыбной ловле: те же эмоции, страсть и азарт, долгое сидение в засаде в конце концов вознаграждается очередной невидимой миру победой...

Евгений Коцюба

Отладчики прошли долгий путь. Раньше всех появился debug.com — пародия, отдаленно напоминающая отладчик, зато входящая в штатную поставку MS-DOS. Сегодня этот инструмент годится разве что для забавы и изучения ассемблера. Впрочем, и тогда от него мало кто был в восторге, и новые отладчики росли, как грибы после дождя. Правда, большинство из них недалеко ушло от своего прототипа, отличаясь от оригинала разве что интерфейсом.

Это было золотое время разработчиков защит. Стоило лишь "запереть" клавиатуру, запретить прерывания, сбросить флаг трассировки, и отладка программы становилась невозможной. Первые мало-мальски пригодные для взлома отладчики появились только после оснащения компьютеров процессором 80286. В памяти хакеров навсегда останутся примеры: AFD PRO, написанный в 1987 г. компанией AdTec GmbH, знаменитый Turbo Debugger, созданный годом позже братьями Вильямс (Chris and Rich Williams), первый эмулирующий отладчик Сергея Пачковки 1991 г. Разработчики защит крякнули, но выдержали: эти отладчики по-прежнему позволяли отлаживаемой программе захватить над собой контроль и очень плохо переносили "извращения" со стеком, экраном, клавиатурой.

Ситуация изменилась с выходом процессора 80386: резкое усложнение программного обеспечения диктовало необходимость наличия развитых отладочных средств в самом процессоре. И в 386 они появились!

Так или иначе, разработчикам защит стали наступать на пятки. Особенно подлила масла в огонь компания NuMega, выпустившая замечательный отладчик SoftIce, до сих пор пользующийся у хакеров огромной популярностью.

NuMega была первой фирмой, которая не только поставила ориентированное на хакеров программное обеспечение на коммерческие рельсы, но и умело манипулировала ситуацией. С одной стороны, ее продукты не объявлялись как хакерские и официально предназначались для отладки своих программ, однако подавляющее большинство пользователей применяло их иначе.

Без сомнений, NuMega — первая и, быть может, единственная хакерская фирма, работающая на легальной основе. Очевидно, что возможности фирмы не идут в сравнение с возможностями одиночек. Разработка полноценного отладчика требует значительного времени, обширных познаний в разных областях (от аппаратного обеспечения до операционной системы) и усилий многих людей. Надо ли говорить, что вскоре написание полнофункционального отладчика уже представляло практически неразрешимую задачу для одиночек.

С этого момента воюющие стороны на некоторое время поменялись местами. За разработчиками защит стояли небольшие фирмы — порой из одного-двух десятков человек, а то и индивидуальные программисты; за хакерами — вся мощь NuMega. Теперь даже не обладающий глубокими познаниями в области системного программирования взломщик мог успешно пользоваться готовым инструментарием, не особо интересуясь, как последний работает. Численность взломщиков росла, и в хакерское общество вливались многие малоприятные и малокультурные личности, постепенно изменившие значение слова "хакер".

Разумеется, разработчики защит, не желая погибать в гордом одиночестве, начали объединяться в крупные фирмы, вовлекая в них хакеров, которым стало душно среди нового поколения компьютерного андеграунда. Тогда никто не сомневался в успехе этих фирм. И все бы на этом могло закончиться. Опытные хакеры старшего поколения при финансовой поддержке фирм-гигантов создали бы хитроумные защиты, с которыми новички бы просто не справились. А всех, кто оказался способен их взломать, нетрудно было бы склонить к сотрудничеству. Хакеру всегда интереснее работать в коллективе единомышленников и профессионалов, нежели бахвальствовать и поливать грязью всех разработчиков.

Жизнь распорядилась иначе. Руководства многих компаний скептически относились к хакерам и неохотно брали их на работу. И даже если такое случалось, хакеры вскоре уходили из приютивших их компаний, где программистов делали "каменщиками", вынужденными работать строго по плану, спущенному "сверху" людьми, далекими от компьютеров и программирования.

Возрастающая сложность программного обеспечения ограничивалась не только ресурсами машины, но и отсутствием возможности поиска и исправления ошибок в коде. Программисты шутили, что один процент времени

уходит на написание программы, а 99% — на ее отладку. Поэтому в своем новом процессоре 80386 фирма Intel ввела специальные механизмы, обеспечивающие контроль исполнения кода на аппаратном уровне. Это означало, что за вечер можно было написать отладчик, против которого были бы бессильны все существующие защитные механизмы, поскольку они были созданы еще в то время, когда о 386 процессоре никто не слышал и, разумеется, никак не пытался защититься от его возможностей.

Первой фирмой, достаточно быстро отреагировавшей на ситуацию, оказалась, естественно, NuMega, выпустившая новую версию своего отладчика для 386 процессора. Заметим, что защиты, использующие предоставленные новым процессором возможности, появились только годы спустя, безнадежно проигрывая в развитии хакерским средствам.

Однако судьба преподнесла враждующим сторонам большой сюрприз в лице новой операционной системы Windows. Принципиально новая архитектура сделала бесполезными все существующие отладчики. Разумеется, без отладчика был немыслим ни один серьезный компилятор, и его появление было просто неизбежным. Однако Microsoft отказалась опубликовать информацию, необходимую для его написания, предоставляя программистам только свои продукты. Что это были за отладчики! Медленные, неповоротливые и вовсе не предназначенные для работы с исполняемым кодом.

Впрочем, остальные фирмы этого так не оставили и довели дело до судебного разбирательства, на котором Microsoft была вынуждена выложить недостающую документацию и дать возможность разрабатывать компиляторы и отладчики сторонним фирмам. Но созданные ими отладчики не превосходили Microsoft debugger и не были ориентированы на работу с исполняемым кодом без исходных текстов.

NuMega в очередной раз удивила мир своим шедевром. Ее отладчик оказался выше всяких похвал. Он ни в чем не полагался на операционную систему и работал непосредственно с аппаратурой, а точнее — между Windows и аппаратурой. В результате стало возможно отлаживать даже ядро операционной системы! Это был триумф, которому никто больше не рискнул подражать. NuMega до сих пор остается единственным поставщиком высококачественных хакер-ориентированных инструментов под Windows, оперативно реагируя на все изменения операционной системы.

Но, так или иначе, SoftIce задал копоти всем защитам и их разработчикам. Пускай он не был (да и сегодня не стал) полностью Stealth-отладчиком, невидимым для отлаживаемых программ, имел и имеет ряд ошибок, позволяющих обнаружить отладчик, завести его или вырваться защите из-под контроля, но в умелых руках отладчик справлялся со всеми этими ограничениями и обходил заботливо расставленные "капканы". И с каждой новой версией SoftIce противостоять ему становилось все труднее и труднее (старые ошибки устранялись быстрее, чем вносились новые).

Постепенно мода на антиотладочные приемы сошла на нет и уж совсем заглохла под победное шествие Windows. Распространилось совершенно нелепое убеждение, что под Windows на прикладном уровне дернуть хвост человеку с отладчиком — невозможно. Это вызывает ухмылку профессионалов, эпизодически встраивающих разные ловушки в свои программы — так, больше для разминки, чем для серьезной борьбы с хакерами.

При современном уровне средств для анализа приложений сегодня, кроме хакеров, серьезную угрозу представляют вчерашние желторотые пользователи, начитавшиеся различных faq, как ломать программы (благо, сейчас они доступны всем кому ни попадя), и теперь только и ищущие, на чем испытать свою богатырскую силу.

Однако технологии, разработанные нашими предшественниками, от этого отнюдь не становятся менее интересными. Мы рассмотрим их с самого начала. С того самого времени, когда на столе разработчиков гордо красовался новый компьютер XT.

1.1. Как работает отладчик

...древним с их мыслящими машинами было куда легче.

Ф. Херберт. "Дюна"

Общаться с отладчиком, не представляя себе, как он работает, было бы, по меньшей мере, невежливо, поэтому далее будут рассмотрены базовые принципы, лежащие в его основе. На полноту это изложение не претендует, но позволяет читателю составить общее представление о вопросе. Технические подробности исчерпывающе изложены в главе "Debugging and Performance Monitoring" технического руководства "Intel Architecture Software Developer's Manual Volume 3: System Programming Guide".

Все существующие отладчики можно разделить на две категории: первые используют отладочные средства процессора, а вторые самостоятельно эмулируют процессор, полностью контролируя выполнение "подопытной" программы.

Качественный эмулирующий отладчик отлаживаемому коду ни обнаружить, ни обойти невозможно, но полноценных эмуляторов Pentium-процессоров на сегодняшний день нет, и вряд ли они появятся в обозримом будущем.

Да и есть ли смысл их создавать? Микропроцессоры Pentium предоставляют в распоряжение разработчика богатейшие отладочные возможности, позволяющие контролировать даже привилегированный код! Они поддерживают пошаговое исполнение программы, отслеживают выполнение инструкции по заданному адресу, контролируют обращения к заданным ячейкам памяти (или портам ввода/вывода), сигнализируют о переключениях задач и т. д.

Если бит трассировки регистра флагов установлен, то после выполнения каждой машинной инструкции автоматически генерируется отладочное исключение INT 1, передавая управление отладчику. Отлаживаемый код может обнаружить трассировку анализом регистра флагов, поэтому для обеспечения собственной невидимости отладчик должен распознавать команды чтения регистра флагов и эмулировать их выполнение, возвращая нулевое значение флага трассировки. Следует обратить внимание на одно важное обстоятельство: после выполнения команды, модифицирующей значение регистра `SS`, отладочное исключение не генерируется! Отладчик должен уметь распознавать такую ситуацию и самостоятельно устанавливать точку останова на следующую инструкцию. В противном случае войти в процедуру, предваренную инструкцией `POP SS`:

```
PUSH SS;
POP SS;
CALL MySecretProc
```

автоматический трассировщик не сможет. Не все современные отладчики учитывают эту тонкость, и такой прием, несмотря на свою архаичность, может оказаться далеко не бесполезным.

Четыре отладочных регистра DR0—DR3 хранят линейные адреса четырех контрольных точек, а управляющий регистр DR7 содержит для каждой из них условие, при выполнении которого процессор генерирует исключение `INT 0x1`, передавая управление отладчику. Всего существует четыре различных условия:

- прерывание при выполнении команды;
- прерывание при модификации ячейки памяти;
- прерывание при чтении или модификации (но не исполнении);
- прерывание при обращении к порту ввода/вывода.

Установкой специального бита можно добиться генерации отладочного исключения при всяком обращении к отладочным регистрам, которое возникает даже в том случае, если их пытается прочесть (модифицировать) привилегированный код. Грамотно спроектированный отладчик может скрыть факт своего присутствия, не позволяя отлаживаемому коду себя обнаружить, какие бы ни были у него привилегии (правда, если "подопытный" код отлаживает сам себя, задействовав все четыре контрольные точки, отладчик не сможет работать).

Если бит T в TSS отлаживаемой задачи установлен, то при каждом переключении на нее будет генерироваться отладочное исключение до выполнения первой команды задачи. Чтобы предотвратить собственное обнаружение, отладчик может отслеживать всякие обращения к TSS и возвращать программе подложные данные. Необходимо заметить: Windows NT, по сооб-

ражениям производительности, не использует TSS (точнее использует, но всего один), и эта отладочная возможность совершенно бесполезна.

Программная точка останова — единственное, что нельзя замаскировать, не прибегая к написанию полноценного эмулятора процессора. Она представляет собой однобайтовый код `0xCC`, который, будучи помещенным в начало инструкции, вызывает исключение `INT 0x3` при попытке ее выполнения. Отлаживаемой программе достаточно подсчитать свою контрольную сумму, чтобы выяснить: была ли установлена хоть одна точка останова или нет. Для достижения этой цели она может воспользоваться командами `MOV`, `MOVS`, `LODS`, `POP`, `CMP`, `CMPS` или любыми другими — никакому отладчику невозможно их всех отследить и проэмулировать.

Настоятельно рекомендуется использовать программные точки останова в тех, и только в тех случаях, когда аппаратных уже не хватает. Однако практически все современные отладчики (в том числе и SoftIce) всегда устанавливают программные точки останова, а не аппаратные. Это обстоятельство может быть с успехом использовано в защитных механизмах, примеры реализаций которых приведены в *разд. 3.4 главы 3*.

Обработка исключений

Когда возникает отладочное исключение (как, впрочем, и любое другое исключение вообще), процессор заносит в стек регистр флагов, адрес следующей (или текущей — в зависимости от рода исключения) выполняемой инструкции и лишь затем передает управление отладчику.

В реальном режиме флаги с адресом возврата заносятся в стек отлаживаемой программы, поэтому факт отладки обнаружить очень просто — достаточно контролировать целостность содержимого, лежащего выше указателя стека. Или, как вариант, установить указатель на его вершину, — тогда добавление новых данных в стек окажется невозможным, и отладчик не сможет функционировать.

Иная ситуация складывается при работе в защищенном режиме — обработчик исключения может находиться в своем собственном адресном пространстве и не использовать никаких ресурсов отлаживаемого приложения, в том числе и стека. Грамотно спроектированный отладчик защищенного режима ни обнаружить, ни блокировать принципиально невозможно, даже привилегированному коду, исполняющемуся в нулевом кольце.

Сказанное справедливо для Windows NT, но неприменимо к Windows 9*x* — эта операционная система не использует должным образом всех преимуществ защищенного режима и всегда "замусоривает" стек отлаживаемой задачи, независимо от того, находится ли она под отладкой или нет.

1.2. Что нам понадобится

Выбор рабочего инструментария — дело сугубо личное и интимное. Тут на вкус и цвет товарищей нет. Поэтому примите все сказанное не как догму, а как рекомендацию к действию. Итак, для чтения книги нам понадобятся:

- отладчик SoftIce версии 3.25 или более старший;
- дизассемблер IDA версии 3.7x или более старшей;
- HEX-редактор HIEW любой версии;
- любой Си\Си++ и Pascal-компилятор, по вкусу;
- пакеты SDK и DDK (последний не обязателен, но очень желателен);
- операционная система — любая из семейства Windows, но лучше Windows 2000.

Теперь обо всем этом подробнее.

- Отладчик SoftIce — основное оружие хакера. Хотя с ним конкурируют windeb (от Microsoft), TRW (от LiuTaoTao) и OllyDbg (от Oleh Yuschuk) — SoftIce много лучше и удобнее всех их вместе взятых. Для наших экспериментов подойдет практически любая версия SoftIce, например, я использую давно апробированную и устойчиво работающую 3.26, замечательно уживающуюся с Windows 2000. К тому же, из всех новых возможностей четвертой версии полезна лишь поддержка FPO (Frame point omission) — локальных переменных, напрямую адресуемых через регистр ESP. Бесспорно, полезная фишка, но без нее можно и обойтись. Найти SoftIce можно на дисках известного происхождения или у российского дистрибьютора **http://www.quarta.ru/bin/soft/winntutils/softicent.asp?ID=59**.
- IDA Pro — это, бесспорно, самый мощный дизассемблер в мире. Прожить без него, конечно, можно, но нужно ли? IDA обеспечивает удобную навигацию по исследуемому тексту, автоматически распознает библиотечные функции и локальные переменные, в том числе и адресуемые через ESP, поддерживает множество процессоров и форматов файлов. Одним словом, хакер без IDA — не хакер. Фирма-разработчик жестоко пресекает любые попытки несанкционированного распространения своего продукта, и единственный надежный путь его приобретения — покупка в самой фирме или у российского дистрибьютора (GelioSoft Ltd., **gav@geliosoft.mtu-net.ru**). К сожалению, с дизассемблером не распространяется никакой документации (не считая встроенного хелпа — очень короткого и бессистемного), поэтому мне ничего не остается, как порекомендовать "Образ мышления — дизассемблер IDA" Криса Касперски, подробно рассказывающей и о самой IDA, и о дизассемблировании вообще.
- HIEW — это не только HEX-редактор, но и дизассемблер, ассемблер и крипт "в одном флаконе". Он не избавит от необходимости приобретения

IDA, но с лихвой заменит IDA в ряде случаев (IDA очень медленно работает и обидно тратить кучу времени, если все, что нам нужно — посмотреть на препарируемый файл "одним глазком"). Впрочем, основное назначение HIEW — отнюдь не дизассемблирование, а bit hack — небольшое хирургическое вмешательство в двоичный файл, обычное вырезание жизненно важного органа защитного механизма, без которого он не может функционировать.

- SDK (Software Development Kit, комплект прикладного разработчика). Из пакета SDK нам, в первую очередь, понадобится документация по Win32 API и утилита для работы с PE-файлами BUMPBIN. Без документации ни хакерам, ни разработчикам никак не обойтись. По крайней мере, необходимо знать прототипы и назначение основных функций системы. Эту информацию, в принципе, можно почерпнуть и из многочисленных русскоязычных книг по программированию, но ни одна из них не может похвастаться полнотой и глубиной изложения. Поэтому, рано или поздно, но все-таки придется обращаться к SDK. Правда, некоторым придется плотно засесть за английский, поскольку документация написана именно на этом языке. Где приобрести SDK? Во-первых, SDK входит в состав MSDN, а сам MSDN ежеквартально издается на компакт-дисках и распространяется по подписке (на официальном сайте **msdn. microsoft.com**). Прилагается MSDN и к компилятору Microsoft Visual C++ 6.0, но, увы, далеко не первой свежести. Впрочем, пользоваться им вполне можно, во всяком случае, для чтения данной книги его будет вполне достаточно.
- DDK (Driver Development Kit — комплект разработчика драйверов). Какую пользу может извлечь хакер из пакета DDK? Ну, в первую очередь, он поможет разобраться, как устроены, работают (и ломаются) драйверы. Помимо основополагающей документации и множества примеров, в него входит очень ценный файл NTDDK.h, содержащий определения большинства недокументированных структур и буквально нашпигованный комментариями, раскрывающими некоторые любопытные подробности функционирования системы. Не лишним будет и инструментарий, прилагающийся к DDK. Среди прочего сюда входит и отладчик windeb. Весьма неплохой, кстати, отладчик, но все же значительно уступающий SoftIce, поэтому и не рассматриваемый в данной книге (но если вы не найдете SoftIce — сгодится и windeb). Более полезным окажется ассемблер MASM, на котором собственно и пишутся драйверы, а также другие полезные программки, облегчающие жизнь хакеру. Последнюю версию DKK можно бесплатно скачать с сайта Microsoft, только имейте в виду, что для NT полный DKK занимает свыше 40 Мбайт в упакованном виде и еще больше требует места на диске.
- Операционная система. Вовсе не собираясь навязывать читателю собственные вкусы и пристрастия, я, тем не менее, настоятельно рекомендую установить именно Windows 2000. Мотивация: это действительно ста-

бильная и устойчиво работающая операционная система, мужественно переносящая все критические ошибки приложений. Потом, Windows 2000 позволяет загружать SoftIce в любой момент без необходимости перезагрузки, что очень удобно! К слову сказать, Windows ME — это не то же самое, что Windows 2000.

Итак, худо-бедно разобравшись с инструментарием, поговорим о сером веществе, ибо при его отсутствии весь собранный инструмент бесполезен. Автор предполагает, что читатель знаком с ассемблером и, если не пишет программ на этом языке, то, по крайней мере, представляет себе, что такое регистры, сегменты, машинные инструкции и т. д. В противном случае эта книга рискует показаться чересчур сложной и непонятной. Отыщите в магазине любой учебник по ассемблеру: например, В. И. Юрова "ASSEMBLER — учебник", П. И. Рудакова "Программируем на языке ассемблера IBM PC" или "Assembler — язык неограниченных возможностей" Зубкова С. В и основательно проштудируйте его.

Помимо знания ассемблера также потребуется иметь хотя бы общие понятия о функционировании операционной системы. Купите и вдумчиво изучите "Windows для профессионалов" Джефри Рихтера и "Секреты системного программирования в Windows 95" Мэтта Питрека. Хоть его книга посвящена Windows 95, частично она справедлива и для Windows 2000. Для знакомства с архитектурой собственно Windows 2000 рекомендуется ознакомиться с шедевром Хелен Кастер "Основы Windows NT" и брошюрой "Недокументированные возможности Windows NT" А. В. Коберниченко.

Касаемо общей теории информатики и алгоритмов — бесспорный авторитет Кнут. Впрочем, на мой вкус, монография М. Броя "Информатика" куда лучше, притом она намного короче, а круг охватываемых ею тем и глубина изложения — много шире. Зачем хакеру теория информатики? Да куда же без нее! Вот, скажем, встретится ему защита с движком-эмулятором машины Тьюринга или Маркова. Слету ее не сломать, надо, как минимум, опознать сам алгоритм: что это вообще такое — Тьюринг, Марков или сеть Петри, а потом отобразить на язык высокого уровня, дабы в удобочитаемом виде анализировать работу защиты. Куда же тут без теории информатики!

За сим все, разве что стоит дополнить наш походный рюкзачок парой учебников по английскому и выкачать с сайтов Intel и AMD всю имеющуюся там документацию по процессорам. На худой конец, подойдет и ее русский перевод, например, Ровдо А. А. "Микропроцессоры от 8086 до Pentium III Xeon и AMD K6-3". Ну-с, рюкзачок на плечо и в путь...

1.3. Особенности отладки в UNIX

Первое знакомство в GDB (что-то вроде debug.com для MS-DOS, только мощнее) вызывает у поклонников Windows смесь разочарования с отвращением, а увесистая документация вгоняет в глубокое уныние, граничащее

с суицидом. Отовсюду торчат рычаги управления, но нету газа и руля. Не хватает только каменных топоров и звериных шкур. Как юниксоиды ухитряются выжить в агрессивной среде этого первобытного мира — загадка.

Несколько строчек исходного кода UNIX еще помнят те древние времена, когда ничего похожего на интерактивную отладку не существовало, и единственным средством борьбы с ошибками был аварийный дамп памяти. Программистам приходилось месяцами (!) ползать по вороху распечаток, собирая рассыпавшийся код в стройную картину. Чуть позже появилась отладочная печать — операторы вывода, понатыканные в ключевых местах и распечатывающие содержимое важнейших переменных. Если происходит сбой, простыня распечаток (в просторечии — "портянка") позволяет установить, чем занималась программа до этого, и кто ее так.

Отладочная печать сохранила свою актуальность и по сей день. В мире Windows она в основном используется лишь в отладочных версиях программы (листинг 1.1) и убирается из финальной (листинг 1.2), что не есть хорошо: когда у конечных пользователей происходит сбой, в руках остается лишь аварийный дамп, на котором далеко не уедешь. Согласен, отладочная печать кушает ресурсы и отнимает время. Вот почему в UNIX так много систем управления протоколированием — от стандартного `syslog` до продвинутого Enterprise Event Logging'га (**http://evlog.sourceforge.net/**). Они сокращают накладные расходы на вывод и журнал, значительно увеличивая скорость выполнения программы.

Листинг 1.1. Неправильный пример использования отладочной печати

```
#ifdef __DEBUG__
       fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
#endif
```

Листинг 1.2. Правильный пример использования отладочной печати

```
if (__DEBUG__)
       fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
```

Отладочная печать на 80% устраняет потребности в отладке, ведь отладчик используется в основном для того, чтобы определить, как ведет себя программа в данном конкретном месте: выполняется ли условный переход или нет, что возвращает функция, какие значения содержатся в переменных и т. д. Просто влепите сюда `fprintf/syslog` и посмотрите на результат!

Человек — не слуга компьютера! Это компьютер придуман для автоматизации человеческой деятельности (в мире Windows — наоборот), поэтому

UNIX "механизирует" поиск ошибок настолько, насколько это только возможно. Включите максимальный режим предупреждений компилятора или возьмите автономные верификаторы кода (самый известный из которых — LINT), и баги побегут из программы, как мыщъх'и с тонущего корабля. (Windows-компиляторы тоже могут генерировать сообщения об ошибках, по строгости не уступающие gcc, но большинство программистов пропускает их. Культура программирования, блин!)

Пошаговое выполнение программы и контрольные точки останова в UNIX используются лишь в клинических случаях (типа трепанации черепа), когда все остальные средства оказываются бессильными. Поклонникам Windows такой подход кажется несовременным, ущербным и жутко неудобным, но это все потому, что Windows-отладчики эффективно решают проблемы, которые в UNIX просто не возникают. Разница культур программирования между Windows и UNIX в действительности очень и очень значительна, поэтому прежде чем кидать камни в чужой огород, наведите порядок у себя. "Непривычное" еще не означает "неправильное". Точно такой же дискомфорт ощущает матерый юниксоид, очутившийся в Windows (рис. 1.1).

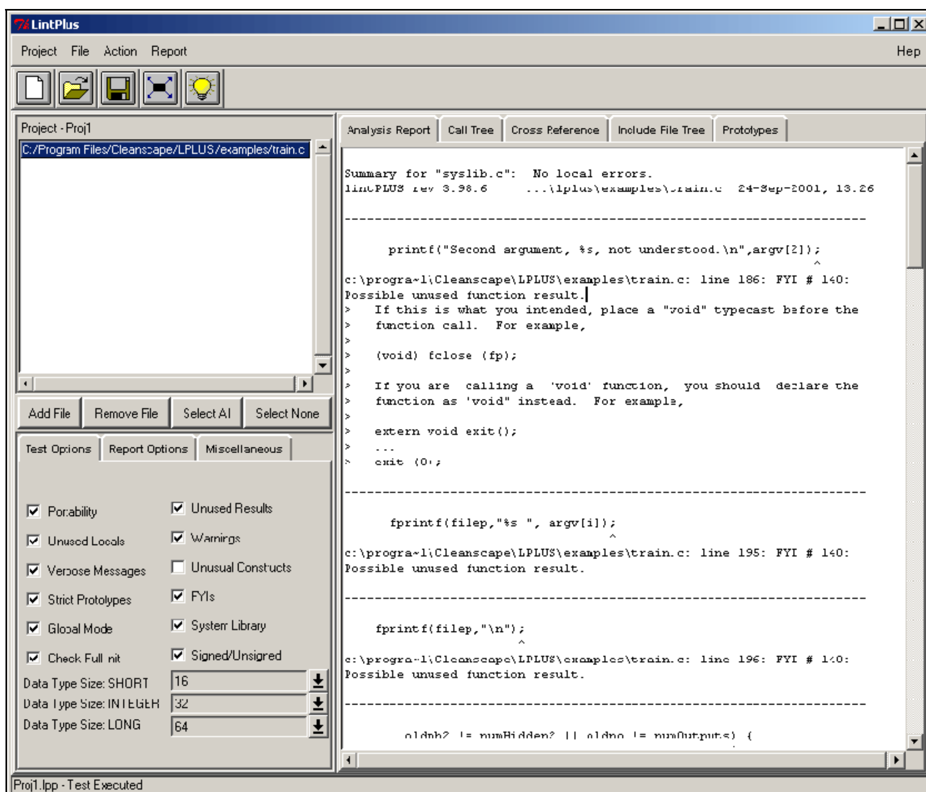


Рис. 1.1. LINT в охоте на багов

PTrace — фундамент для GDB

GDB — это системно независимый кроссплатформенный отладчик. Как и большинство UNIX-отладчиков, он основан на библиотеке PTrace, реализующей низкоуровневые отладочные примитивы. Для отладки многопоточных процессов и параллельных приложений рекомендуется использовать дополнительные библиотеки, а еще лучше — специализированные отладчики типа Total View (**http://www.etnus.com**), поскольку GDB с многопоточностью справляется не самым лучшим образом (рис. 1.2).

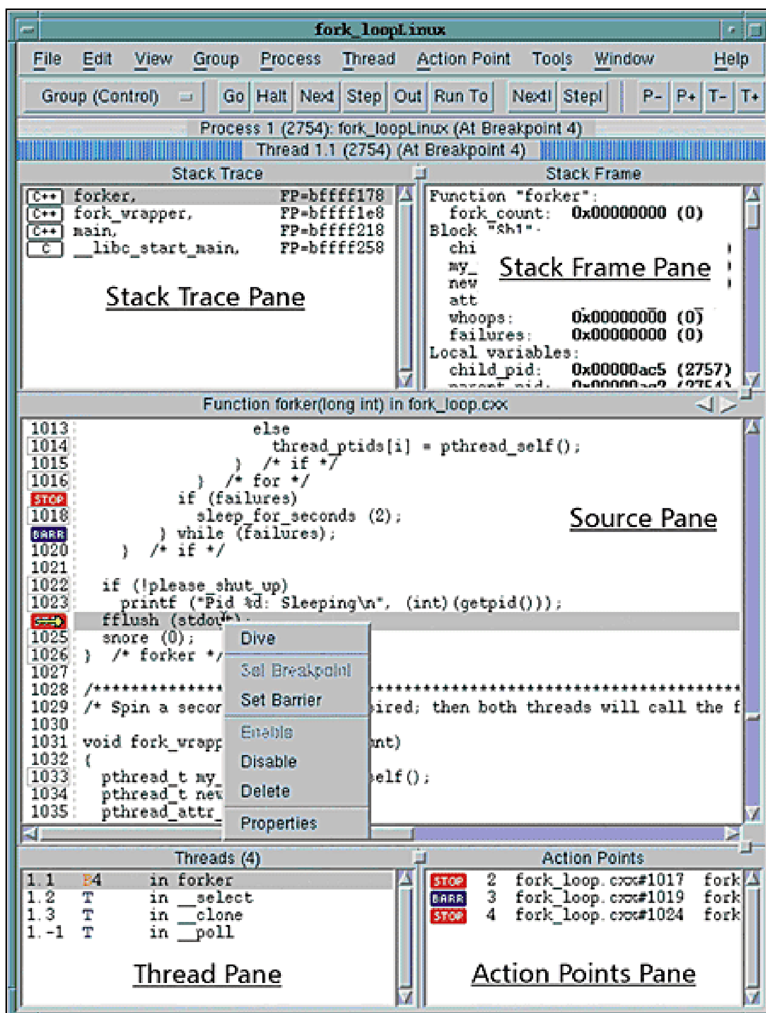


Рис. 1.2. Внешний вид отладчика Total View, специализирующихся на параллельных приложениях

PTrace может переводить процесс в состояние останова/возобновлять его выполнение, читать/записывать данные в адресном пространстве отлаживаемого процесса, читать/записывать регистры центрального процессора. На i386 это регистры общего назначения, сегментные регистры, регистры "сопроцессора" (включая SSE) и отладочные регистры семейства DRx (они нужны для организации аппаратных точек останова). В Linux еще можно манипулировать служебными структурами отлаживаемого процесса и отслеживать вызов системных функций. В "правильных" UNIX этого нет, и недостающую функциональность приходится реализовывать уже в отладчике.

Пример использования PTrace в подсчете количества машинных команд приведен в листинге 1.3, для компиляции под Linux замените `PT_TRACE_ME` на `PTRACE_TRACEME`, а `PT_STEP` на `PTRACE_SINGLESTEP`.

Листинг 1.3. Пример использования PTrace на FreeBSD

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

main()
{
	int pid;			// pid отлаживаемого процесса
	int wait_val;			// сюда wait записывает
					// возвращаемое значение
	long long counter = 1;		// счетчик трассируемых инструкций


	// расщепляем процесс на два
	// родитель будет отлаживать потомка
	// (обработка ошибок для наглядности опущена)
	switch (pid = fork())
	{
		case 0:			// дочерний процесс (его отлаживают)

			// папаша, ну-ка потрассируй меня!
			ptrace(PT_TRACE_ME, 0, 0, 0);

			// вызываем программу, которую надо отрассировать
```

```
            // (для программ, упакованных Шифрой, это не сработет)
            execl("/bin/ls", "ls", 0);
            break;

        default:             // родительский процесс (он отлаживает)

            // ждем, пока отлаживаемый процесс
            // не перейдет в состояние останова
            wait(&wait_val);

            // трассируем дочерний процесс, пока он не завершится
            while (WIFSTOPPED(wait_val) /* 1407 */)
            {
                // выполнить следующую машинную инструкцию
                // и перейти в состояние останова
                if (ptrace(PT_STEP, pid, (caddr_t) 1, 0)) break;

                // ждем, пока отлаживаемый процесс
                // не перейдет в состояние останова
                wait(&wait_val);

                // увеличиваем счетчик выполненных
                // машинных инструкций на единицу
                counter++;
            }
        }
        // вывод кол-ва выполненных машинных инструкций на экран
        printf("== %lld\n", counter);
}
```

PTrace и ее команды

В user-mode доступа всего лишь одна функция:

```
ptrace((int _request, pid_t _pid, caddr_t _addr, int _data))
```

Но зато эта функция делает все! При желании вы можете за пару часов написать собственный мини-отладчик, специально заточенный под вашу проблему.

Аргумент `_request` функции `ptrace` важнейший из всех — он определяет, что мы будем делать. Заголовочные файлы в BSD и Linux используют различные определения, затрудняя перенос приложений PTrace с одной плат-

формы на другую. По умолчанию мы будем использовать определения из заголовочных файлов BSD.

- PT_TRACE_ME (в Linux — PTRACE_TRACEME) переводит текущий процесс в состояние останова. Обычно используется совместно с fork, хотя встречаются также и самотрассирующиеся приложения. Для каждого из процессов вызов PT_TRACE_ME может быть сделан лишь однажды. Трассировать уже трассируемый процесс не получится (менее значимое следствие — процесс не может трассировать сам себя, сначала он должен расщепиться). На этом основано большое количество антиотладочных приемов, для преодоления которых приходится использовать отладчики, работающие в обход ptrace. Отлаживаемому процессу посылается сигнал, переводящий его в состояние останова, из которого он может быть выведен командой PT_CONTINUE или PT_STEP, вызванной из контекста родительского процесса. Функция wait задерживает управление материнского процесса до тех пор, пока отлаживаемый процесс не перейдет в состояние останова или не завершится (тогда она возвращает значение 1407). Остальные аргументы игнорируются.
- PT_ATTACH (в Linux — PTRACE_ATTACH) — переводит в состояние останова уже запущенный процесс с заданным pid, при этом процесс-отладчик становится его предком. Остальные аргументы игнорируются. Процесс должен иметь тот же самый uid, что и отлаживающий процесс, и не быть процессом setuid/setduid (или отлаживаться каталогом root).
- PT_DETACH (в Linux — PTRACE_DETACH) прекращает отладку процесса с заданным pid (как по PT_ATTACH, так и по PT_TRACE_ME) и возобновляет его нормальное выполнение. Все остальные аргументы игнорируются.
- PT_CONTINUE (в Linux — PTRACE_CONT) — возобновляет выполнение отлаживаемого процесса с заданным pid без разрыва связи с процессом-отладчиком. Если addr == 1 (в Linux — 0), выполнение продолжается с места последнего останова, в противном случае — с указанного адреса. Аргумент _data задает номер сигнала, посылаемого отлаживаемому процессу (ноль — нет сигналов).
- PT_STEP (в Linux — PTRACE_SINGLESTEP) — пошаговое выполнение процесса с заданным pid: выполнить следующую машинную инструкцию и перейти в состояние останова (под i386 это достигается взводом флага трассировки, хотя некоторые хакерские библиотеки используют аппаратные точки останова). BSD требует, чтобы аргумент addr был равен 1, Linux хочет видеть здесь 0. Остальные аргументы игнорируются.
- PT_READ_I/PT_REEAD_D (в Linux — PTRACE_PEEKTEXT/PTRACE_PEEKDATA) — чтение машинного слова из кодовой области и области данных адресного пространства отлаживаемого процесса соответственно. На большинстве современных платформ обе команды полностью эквивалентны. Функция ptrace принимает целевой addr и возвращает считанный результат.

- PT_WRITE_I/PR_READ_D (в Linux — PTRACE_POKETEXT, PTRACE_POKEDATA) — запись машинного слова, переданного в _data, по адресу addr.
- PT_GETREGS/PT_GETFPREGS/PT_GETDBREGS (в Linux — PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_GETFPXREGS) — чтение регистров общего назначения, сегментных и отладочных регистров в область памяти процесса-отладчика, заданную указателем _addr. Это системно зависимые команды, приемлемые только для i386 платформы. Описание регистровой структуры содержится в файле <machine/reg.h>.
- PT_SETREGS/PT_SETFPREGS/PT_SETDBREGS (в Linux — PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_SETFPXREGS) — установка значения регистров отлаживаемого процесса путем копирования содержимого региона памяти по указателю _addr.
- PT_KILL (в Linux — PTRACE_KILL) — посылает отлаживаемому процессу сигнал sigkill, который делает ему харакири.

Поддержка многопоточности в GDB

Определить, поддерживает ли ваша версия GDB многопоточность или нет, можно при помощи команды

```
info thread
```

(вывод сведений о потоках), а для переключений между потоками используйте

```
thread N
```

Если поддержка многопоточности отсутствует, обновите GDB до версии 5x или установите специальный патч, поставляемый вместе с вашим клоном UNIX или распространяемый отдельно от него (листинги 1.4–1.5).

Листинг 1.4. Отладка многопоточных приложений не поддерживается

```
(gdb) info threads
(gdb)
```

Листинг 1.5. Отладка многопоточных приложений поддерживается

```
info threads
  4 Thread 2051 (LWP 29448) RunEuler (lpvParam=0x80a67ac) at eu_kern.cpp:633
  3 Thread 1026 (LWP 29443) 0x4020ef14 in __libc_read () from /lib/libc.so.6
* 2 Thread 2049 (LWP 29442) 0x40214260 in __poll (fds=0x80e0380,
nfds=1, timeout=2000)
  1 Thread 1024 (LWP 29441)  0x4017caea in __sigsuspend (set=0xbffff11c)
(gdb) thread 4
```