HIGH TECH

МАЙКЛ САТТОН
АДАМ ГРИН
ПЕДРАМ АМИНИ

FUZZING

ИССЛЕДОВАНИЕ УЯЗВИМОСТЕЙ МЕТОДОМ ГРУБОЙ СИЛЫ

Символ®

FUZZING

Brute Force Vulnerability Discovery

Michael Sutton, Adam Greene
and Pedram Amini

Addison-Wesley

HIGH TECH

FUZZING

ИССЛЕДОВАНИЕ УЯЗВИМОСТЕЙ МЕТОДОМ ГРУБОЙ СИЛЫ

Майкл Саттон, Адам Грин и Педрам Амини

Санкт-Петербург — Москва
2009

Серия «High tech»

Майкл Саттон, Адам Грин и Педрам Амини

Fuzzing: исследование уязвимостей методом грубой силы

Перевод А. Коробейникова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*А. Пасечник*
Редактор	*Н. Рощина*
Научный редактор	*Б. Попов*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Саттон М., Грин А., Амини П.

Fuzzing: исследование уязвимостей методом грубой силы. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 560 с., ил.

ISBN: 978-5-93286-147-9

Фаззинг – это процесс отсылки намеренно некорректных данных в исследуемый объект с целью вызвать ситуацию сбоя или ошибку. Настоящих правил фаззинга нет. Это такая технология, при которой успех измеряется исключительно результатами теста. Для любого отдельно взятого продукта количество вводимых данных может быть бесконечным. Фаззинг – это процесс предсказания, какие типы программных ошибок могут оказаться в продукте, какие именно значения ввода вызовут эти ошибки. Таким образом, фаззинг – это более искусство, чем наука.

Настоящая книга – первая попытка отдать должное фаззингу как технологии. Знаний, которые даются в книге, достаточно для того, чтобы начать подвергать фаззингу новые продукты и строить собственные эффективные фаззеры. Ключ к эффективному фаззингу состоит в знании того, какие данные и для каких продуктов нужно использовать и какие инструменты необходимы для управления процессом фаззинга.

Книга представляет интерес для обширной аудитории: как для тех читателей, которым ничего не известно о фаззинге, так и для тех, кто уже имеет существенный опыт.

ISBN: 978-5-93286-147-9
ISBN: 978-0-321-44611-4 (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 30.07.2009. Формат 70×100¹/16. Печать офсетная.
Объем 35 печ. л. Тираж 1200 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Эта книга посвящается двум главным женщинам в моей жизни. Мама, без всех твоих жертв я ничего бы не смог. Эта книга – всего лишь один небольшой пример. Аманда, твоя неизменная любовь и поддержка каждый день вдохновляют меня на новые подвиги. Мне несказанно повезло быть мужем такой женщины.

Майкл Саттон

Эта работа посвящается моей семье и друзьям. Спасибо всем вам за поддержку и терпение.

Адам Грин

Посвящаю эту книгу Джорджу Бушу-младшему, моему главнокомандующему, чьи впечатляющие карьерные достижения при скромных лингвистических способностях заставили меня поверить, что и я могу написать книгу.

Педрам Амини

Оглавление

Введение

Поиск уязвимостей – это краеугольный камень исследования безопасности. При проведении проникающего теста, оценке нового продукта или проверке исходного кода важнейшего компонента уязвимости управляют вашими решениями, обосновывают затраченное время и влияют на ваш выбор на протяжении многих лет.

Проверка исходного кода – это тестирование методом белого ящика, который в течение длительного времени является популярным подходом к исследованию уязвимостей в программных продуктах. Этот метод требует от аудитора знания всех идей программы и использованных в продукте функций, а также глубокого понимания операционной среды продукта. Проверка исходного кода при этом содержит очевидную западню: сам исходный код, разумеется, должен быть доступен.

К счастью, существует альтернативный метод черного ящика, при котором доступ к исходному коду не требуется. Одна из таких альтернатив – это технология фаззинга, которая неплохо зарекомендовала себя при нахождении серьезных уязвимостей, которые иначе обнаружить было бы невозможно. Фаззинг – это процесс отсылки намеренно некорректных данных в объект с целью вызвать ситуацию сбоя или ошибку. Такие ситуации сбоя могут привести к выявлению уязвимостей.

Определенных правил фаззинга не существует. Это такая технология, при которой успех измеряется исключительно результатами теста. Для любого отдельно взятого продукта количество вводимых данных может быть бесконечным. Фаззинг – это процесс предсказания того, какие типы программных ошибок могут обнаружиться в продукте и какие именно введенные значения вызовут эти ошибки. Таким образом, фаззинг – это больше искусство, чем наука.

Фаззинг может быть таким же простым, как случайный стук по клавиатуре. Трехлетний сын одного моего знакомого однажды обнаружил таким способом уязвимость в блокировке экрана операционной системы Mac OS X. Друг заблокировал экран и пошел на кухню попить. Когда он вернулся, его сын ухитрился вскрыть блокировку и запустил веб-броузер, просто барабаня по клавиатуре.

За последние несколько лет я использовал инструменты и технологии фаззинга для обнаружения сотен уязвимостей в большом количестве программ. В декабре 2003 года я написал простенький инструмент,

который отсылал поток случайных UDP-пакетов на удаленный сервис. Этот инструмент помог обнаружить две новых уязвимости сервера Microsoft WINS. Тот же инструмент позже помог выявить серьезные недостатки в некоторых других продуктах. Выяснилось, что случайного потока пакетов UDP достаточно для обнаружения уязвимостей во многих продуктах Computer Associates, в сервисе Norton Ghost и в стандартном сервисе оперативной системы Mac OS X.

Фаззеры полезны не только при работе с сетевыми протоколами. В первом квартале 2006 года я участвовал в работе над тремя различными фаззерами броузеров, которые помогли найти десятки недостатков в большом количестве веб-броузеров. Во втором квартале 2006 года я написал фаззер ActiveX (AxMan), с помощью которого который выявил более 100 неизвестных прежде уязвимостей только в продуктах Microsoft. Многие из них обнаружились во время работы над проектом Month of Browser Bugs («Месяц броузерных багов»), что привело к разработке эксплойтов для Metasploit Framework. Я все еще обнаруживаю своим AxMan новые уязвимости, хотя прошло больше года с момента его выпуска. Фаззеры – это воистину дар, который не иссякает.

Эта книга – первая попытка отдать должное фаззингу как технологии. Сведений, которые даются в книге, достаточно для того, чтобы начать подвергать фаззингу новые продукты и создавать собственные эффективные фаззеры. Ключ к эффективному фаззингу состоит в знании того, какие данные и для каких продуктов нужно использовать и какие инструменты необходимы для управления процессом фаззинга, а также его контролирования. Авторы книги – пионеры в этой области, они проделали большую работу, раскрывая хитрости процесса фаззинга.

Удачной охоты на баги!

Х. Д. Мур

Предисловие

Я уверен, что человек и рыба
могут мирно сосуществовать.

Джордж Буш-мл.,
Сагино, штат Мичиган,
29 сентября 2000 года

Начальные сведения

Идея фаззинга муссировалась почти два десятка лет, но лишь недавно привлекла к себе всеобщее внимание. Волна уязвимостей в популярных клиентских приложениях, в том числе Microsoft Internet Explorer, Microsoft Word и Microsoft Excel, обнаружилась в 2006 году, и солидная их часть была выявлена именно фаззингом. Такая эффективность применения технологий фаззинга подготовила почву для создания новых инструментов и привела к расширению его использования. Тот факт, что эта книга – первая, посвященная данному вопросу, служит дополнительным индикатором возросшего интереса к фаззингу.

Много лет вращаясь в обществе исследователей уязвимостей, в повседневной работе мы использовали множество технологий фаззинга, начиная с незрелых проектов, которые были скорее хобби, и заканчивая коммерческими продуктами высокого уровня. Каждый из авторов занимался и занимается разработкой как частных, так и общедоступных фаззеров. Мы использовали наши общие знания и текущие исследовательские проекты для того, чтобы чуть ли не кровью сердца написать эту книгу, которую вы, надеемся, найдете полезной.

Целевая аудитория

Книги и статьи о безопасности часто пишут исследователи безопасности для других исследователей безопасности. Мы же убеждены, что количество уязвимостей будет постоянно увеличиваться, а качество – улучшаться, пока безопасностью будут заниматься только ответственные за данную область работники. Поэтому мы старались писать для

большой аудитории: как для тех читателей, которым ничего не известно о фаззинге, так и для тех, кто уже имеет существенный опыт его применения.

Не стоит обманывать себя и верить, что безопасные приложения появятся немедленно после того, как мы передадим в отдел безопасности готовые приложения для быстрой проверки продукта перед запуском в обращение. Прошли времена, когда разработчик или сотрудник контроля качества мог сказать: «Безопасность – это не моя проблема, у нас об этом заботится отдел безопасности». Сейчас безопасность – это общая проблема. Обеспечение безопасности должно быть внедрено в жизненный цикл разработки ПО (SDLC), а не приделано к его «хвосту».

Требовать от разработчиков и службы контроля качества, чтобы они сосредоточились на безопасности, может оказаться трудным делом, особенно если раньше от них этого не требовали. Мы считаем, что фаззинг представляет собой уникальную методологию поиска уязвимостей, которая доступна широким массам благодаря тому, что легко поддается автоматизации. Мы надеемся, что из этой книги извлекут пользу как закаленные в боях исследователи безопасности, так и разработчики и сотрудники службы контроля качества. Фаззинг может и должен стать частью любого SDLC не только на стадии тестирования, но и в течение всего срока разработки. Чем раньше определен дефект, тем меньше убытков он сможет нанести.

Предварительные условия

Фаззинг – обширная область. Когда мы рассказываем в книге о множестве вещей, не относящихся к фаззингу, то считаем, что определенные знания у читателя уже имеются. Перед тем как браться за книгу, читатели должны получить как минимум основные сведения о программировании и компьютерных сетях. Фаззинг – это автоматизация тестирования безопасности, поэтому естественно, что многие фрагменты книги посвящены построению инструментов. Для этих целей мы нарочно выбирали разные языки программирования. Языки выбирались в соответствии с задачами, но также для демонстрации того, что фаззинг допускает различные подходы. Совершенно необязательно владеть всеми использованными языками, но знание одного или двух языков, безусловно, поможет вам узнать больше из этих глав.

В книге мы рассказываем о многих уязвимостях, обсуждаем, как их можно было бы обнаружить посредством фаззинга. Однако в нашу задачу не входило определить или рассмотреть сами уязвимости. Этой теме посвящено множество замечательных книг. Если вам нужен букварь по программным уязвимостям, то стоит воспользоваться книгой «Exploiting Software» (Использование уязвимостей) Грега Хоглунда (Greg Hoglund) и Гэри Макгроу (Gary McGraw), книгами из серии «Hacking Exposed», а также «The Shellcoder's Handbook» (Учебник по

программированию оболочек) Джека Козелла (Jack Koziol), Дэвида Литчфилда (David Litchfield) и др.

Подход

То, как лучше использовать эту книгу, зависит от ваших опыта и намерений. Если в фаззинге вы новичок, рекомендуем изучать книгу с начала, поскольку предполагалось вначале дать необходимую общую информацию, а затем перейти к более сложным темам. Однако если вы уже пользовались какими-либо инструментами для фаззинга, не бойтесь погрузиться именно в те темы, которые вас интересуют, поскольку различные логические разделы и группы глав во многом независимы друг от друга.

Часть I призвана подготовить площадку для отдельных типов фаззинга, которые будут обсуждаться в дальнейшем. Если в мире фаззинга вы новичок, считайте, что прочесть эти главы необходимо. Фаззинг можно использовать для поиска уязвимостей практически в любом объекте, но все подходы имеют почти одинаковые принципы. В части I мы хотели определить фаззинг как методологию обнаружения ошибок и подробно рассказать о том, что потребуется знать независимо от типа проводимого фаззинга.

В части II сделан акцент на фаззинге отдельных классов объектов. Каждому объекту посвящено две-три главы. Первая из них содержит базовую информацию, специфичную для данного класса, а последующие рассказывают об автоматизации, подробно раскрывая процесс создания фаззеров для определенного объекта. Об автоматизации говорится в двух главах, если необходимо создать отдельные инструменты для платформ Windows и UNIX. Посмотрите, например, на трио «Фаззинг формата файла», начинающееся с главы 11, которая дает базовую информацию, связанную с фаззерами файлов. В главе 12 «Фаззинг формата файла: автоматизация под UNIX» подробно рассказывается о написании фаззера на платформе UNIX, а в главе 13 «Фаззинг формата файла: автоматизация под Windows» – о создании фаззера формата файла, который будет работать в среде Windows.

В части III описываются продвинутые этапы фаззинга. Для тех, кто уже имеет значительный опыт фаззинга, уместным может быть переход прямо к этой части, в то время как большинство читателей, вероятно, предпочтет потратить время и на части I и II. В части III мы сосредоточиваем внимание на тех технологиях, которые только входят в обращение, но в будущем станут важнейшими при обнаружении уязвимостей с помощью фаззинга.

Наконец, в части IV мы суммируем все, что узнали в этой книге, и затем сквозь магический кристалл пытаемся разглядеть будущее. Хотя фаззинг – это не новая идея, ему еще есть куда расти, и мы надеемся, что эта книга станет толчком к новым поискам в этой области.

О юморе

Написание книги – серьезная работа, в особенности книги о таком сложном предмете, как фаззинг. Однако мы любим посмеяться не меньше, чем любой другой (честно говоря, значительно больше, чем средний) читатель, и поэтому приложили массу усилий к тому, чтобы книга получилась увлекательной. Поэтому мы решили начинать каждую главу краткой цитатой из речей 43-го президента Соединенных Штатов Джорджа Буша-младшего (a.k.a. Dubya). Неважно, к какой партии вы принадлежите, каких взглядов придерживаетесь, – никто не будет спорить, что мистер Буш за годы пребывания у власти изрек множество достойных фраз, с помощью которых можно составить целый календарь![1] Мы решили поделиться некоторыми из наших любимых изречений с вами и надеемся, что вам они покажутся такими же забавными, как и нам. Как вы увидите, прочитав книгу, фаззинг может быть применен к самым разным объектам и, судя по всему, даже к английскому языку.

Об обложке оригинального издания

Порой уязвимости именуются «рыбой». (Смотрите, например, тред «The L Word & Fish»[2] на рассылке о безопасности DailyDave.) Это полезная аналогия, которую можно постоянно применять при обсуждении безопасности и уязвимостей. Исследователя можно назвать рыбаком. Реинжиниринг кода сборки приложения, строка за строкой, в поиске уязвимости – глубоководной рыбалкой.

По сравнению с множеством других тактик анализа фаззинг большей частью скребет по поверхности и высокоэффективен при ловле «легкой рыбки». К тому же медведь гризли – это «мохнатый» (fuzzy) и мощный зверь. Эти две предпосылки и определили наш выбор обложки, на которой медведь, символизирующий фаззера, ловит рыбу, символизирующую уязвимость.

Сопутствующий веб-сайт: www.fuzzing.org

Веб-сайт *fuzzing.org* – неотъемлемая часть этой книги, а вовсе не дополнительный ресурс. Он не только содержит список опечаток, которые, несомненно, появятся при публикации, но и служит центральным хранилищем всех исходных кодов и инструментов, о которых говорится в книге. Мы развивали *fuzzing.org* в направлении от сопутствующего книге веб-сайта к полезному ресурсу, содержащему инструменты и информацию по всем дисциплинам фаззинга. Ваши отзывы приветствуются: они помогут сделать сайт ценной и открытой для всех базой знаний.

---

[1] *http://tinyurl.com/33l54g*

[2] *http://archives.neohapsis.com/archives/dailydave/2004-q1/0023.html*

Благодарности

Коллективные благодарности

Хотя на обложке фигурируют только три имени, за сценой осталась внушительная группа поддержки, благодаря которой эта книга стала реальностью. Прежде всего это наши друзья и семьи, смирившиеся с полуночными сидениями и пропавшими уик-эндами. Мы все коллективно задолжали несколько кружек пива, походов в кино и тихих домашних вечеров, которые были потеряны во время работы над этим проектом, но не бойтесь: тем, кому мы задолжали, мы все вернем сполна. Когда мы пропускали регулярные совещания по поводу книги вечером по четвергам, то обнаруживали, что остальные вовсе не собираются их пропускать.

Питер Девриес (Peter DeVriews) однажды сказал: «Мне нравится быть писателем. Но вот бумажную работу я просто не выношу». Нельзя было выразиться точнее. Новые мысли и идеи – это только половина работы. После написания черновика в борьбу вступила небольшая армия рецензентов, пытаясь убедить мир, что мы можем написать книгу. Особенно мы хотели бы поблагодарить технических редакторов, которые указали на ошибки и развеяли нашу самонадеянность, прежде всего Чарли Миллера (Charlie Miller), который не утонул в бумагах и сумел сделать эту книгу доходчивой. Также мы искренне признательны Х. Д. Муру (H. D. Moore) за те усилия, которые он приложил, написав на книгу рецензию, а к книге – предисловие. Хотим поблагодарить и команду издательства Addison-Wesley, которая помогала нам в процессе написания: Шери Кейн (Sheri Cain), Кристин Вайнбергер (Kristin Weinberger), Ромни Френч (Romny French), Джена Джонс (Jana Jones) и Лори Лайонс (Lori Lyons). Наконец, мы выражаем особую благодарность нашему редактору Джессике Голдстейн, которая решила дать шанс трем парням с дурацкими идеями и наивной верой в то, что написать книгу совсем несложно.

Благодарности от Майкла

Я хотел бы воспользоваться возможностью поблагодарить свою жену Аманду за терпение и понимание в процессе написания этой книги. В течение большей части работы над книгой мы планировали свадьбу,

и слишком много вечеров и уик-эндов прошло перед экраном компьютера, а не с бутылкой вина на балконе. Также я искренне благодарен за поддержку всем членам моей семьи, которые подвигли меня на работу над этим проектом и верили, что мы его осилим. Спасибо команде из iDefense Labs и моим коллегам из SPI Dynamics, которые поддерживали и вдохновляли меня в процессе работы. Наконец, я хочу поблагодарить своих соавторов, которые включились в совместную работу, мирились с моими речами о GOYA, мотивировали меня на GOMOA и с которыми мы создали гораздо лучшую книгу, чем я написал бы в одиночку.

Благодарности от Адама

Я хотел бы поблагодарить свою семью (особенно сестру и родителей), учителей и советников в JTHS, Марка Чегвиддена (Mark Chegwidden), Луиса Коллуччи (Louis Collucci), Криса Буркхарта (Chris Burkhart), sgo, Nadwodny, Дэйва Айтеля (Dave Aitel), Джейми Брейтен (Jamie Breiten), семью Дэвисов, братьев Леонди, Kloub and AE, Лусарди, Лапиллу и, наконец, Ричарда.

Благодарности от Педрама

Я хотел бы поблагодарить своих соавторов за возможность написать эту книгу и за то, что они мотивировали меня во время работы. Моя благодарность распространяется также на Коди Пирса (Cody Pierce), Кэмерона Хотчкиса (Cameron Hotchkies) и Аарона Портного (Aaron Portnoy), моих коллег по TippingPoint за их остроумие и технические консультации. Спасибо Питеру Зильберману (Peter Silberman), Джейми Батлеру (Jamie Butler), Грегу Хоглунду, Халвару Флейку (Halvar Flake) и Эро Каррере (Ero Carrera) за поддержку и то, что они постоянно развлекали меня. Особая благодарность Дэвиду Эндлеру (David Endler), Ральфу Шиндлеру (Ralph Schindler), Сунилу Джеймсу (Sunil James) и Николасу Аугелло (Nicolas Augello), моим «братьям от других матерей», за то, что на них всегда можно было положиться. Наконец, сердечная благодарность моей семье, которая терпеливо переносила мое отсутствие, вызванное работой над книгой.

Об авторах

Майкл Саттон

Майкл Саттон (Michael Sutton) – ответственный за безопасность в SPI Dynamics. В этом качестве Майкл отвечает за обнаружение, исследование и обработку проблем, возникающих в индустрии безопасности веб-приложений. Он часто выступает на крупнейших конференциях по безопасности, является автором многих статей, его часто цитируют в прессе по различным связанным с безопасностью поводам. Также Майкл – член Консорциума по безопасности веб-приложений (WASC), где он руководит проектом статистики безопасности веб-приложений.

До перехода в SPI Dynamics Майкл был директором в iDefense/VeriSign, где возглавлял iDefense Labs, коллектив исследователей мирового класса, занимавшихся обнаружением и исследованием изъянов в безопасности. Майкл также основал семинар «Совещательный орган по безопасности информационных сетей» (ISAAS) на Бермудах для компании Ernst & Young. Он имеет степени университета Альберта и университета Джорджа Вашингтона.

Майкл – настоящий канадец; он считает, что хоккей – это религия, а не спорт. В свободное от работы время он служит сержантом добровольческой пожарной охраны в Фэрфексе.

Адам Грин

Адам Грин (Adam Green) – инженер в крупной нью-йоркской компании, специализирующейся на финансовых новостях. Ранее он работал инженером в iDefense Labs, информационной компании из Рестона, штат Виргиния. Его научные интересы в компьютерной безопасности лежат в основном в области надежных методов эксплуатации, фаззинга и аудита, а также разработки эксплойтов для работающих на UNIX-системах.

Педрам Амини

Педрам Амини (Pedram Amini) в данный момент возглавляет отдел исследования и определения безопасности продукта в TippingPoint. До

того он был заместителем директора и одним из отцов-основателей iDefense Labs. Несмотря на множество звучных титулов, он проводит много времени, занимаясь обычным реинжинирингом: разрабатывает автоматизированные инструменты, плагины и скрипты. Среди самых последних его проектов (так называемых детей) – структура реинжиниринга PaiMei и структура фаззинга Sulley.

Подчиняясь своей страсти, Педрам запустил OpenRCE.org, общественный веб-сайт, который посвящен науке и искусству реинжиниринга. Он выступал на RECon, BlackHat, DefCon, ShmooCon и ToorCon и руководил многими курсами по реинжинирингу, от желающих записаться на которые не было отбоя. Педрам имеет степень по компьютерным наукам университета Тьюлейн.

I

Основы

1

Методологии выявления уязвимости

Станет ли меньше дорог Интернета?

Джордж Буш-мл.,
29 января 2000 года

Спросите любого специалиста по компьютерной защите о том, как он выявляет уязвимости системы, и вы получите множество ответов. Почему? Есть множество подходов, и у каждого свои достоинства и недостатки. Ни один из них не является единственно правильным, и ни один не способен раскрыть все возможные варианты реакции на заданный стимул. На высшем уровне выделяются три основных подхода к выявлению недостатков системы: тестирование методами белого ящика, черного ящика и серого ящика. Различия в этих подходах заключаются в тех средствах, которыми вы как тестер располагаете. Метод белого ящика представляет собой одну крайность и требует полного доступа ко всем ресурсам. Необходим доступ к исходному коду, знание особенностей дизайна и порой даже знакомство непосредственно с программистами. Другая крайность – метод черного ящика, при котором не требуется практически никакого знания внешних особенностей; он весьма близок к слепому тестированию. Пен-тестирование удаленного веб-приложения без доступа к исходному коду как раз и является хорошим примером тестирования методом черного ящика. Между ними находится метод серого ящика, определение которого варьируется, кого о нем ни спроси. Для наших целей метод серого ящика требует по крайней мере доступа к компилированным кодам и, возможно, к части основной документации.

В этой главе мы исследуем различные подходы к определению уязвимости системы как высокого, так и низкого уровня и начнем с тестирования методом белого ящика, о котором вы, возможно, слышали также как о методе чистого, стеклянного или прозрачного ящика. Затем мы дадим определения методов черного и серого ящиков, которые включают в себя фаззинг. Мы рассмотрим преимущества и недостатки этих подходов, и это даст нам изначальные знания для того, чтобы на протяжении всей остальной книги сконцентрироваться на фаззинге. Фаззинг – всего лишь один из подходов к нахождению уязвимости системы, и поэтому важно бывает понять, не будут ли в конкретной ситуации более полезными другие подходы.

Метод белого ящика

Фаззинг как методика тестирования в основном относится к областям серого и черного ящиков. Тем не менее, начнем мы с определения популярного альтернативного варианта тестирования чувствительности системы, который разработчики программного обеспечения именуют методом белого ящика.

Просмотр исходного кода

Просмотр исходного кода можно выполнить вручную или с помощью каких-либо автоматических средств. Учитывая, что компьютерные программы обычно состоят из десятков, сотен а то и тысяч строк кода, чисто ручной визуальный просмотр обычно нереален. И здесь автоматические средства становятся неоценимой помощью, благодаря которой утомительное отслеживание каждой строчки кода сводится к определению только *потенциально* чувствительных или подозрительных сегментов кода. Человек приступает к анализу, когда нужно определить, верны или неверны подозрительные строки.

Чтобы достичь полезных результатов, программам анализа исходного кода требуется преодолеть множество препятствий. Раскрытие всего комплекса этих задач лежит за пределами данной книги, однако давайте рассмотрим образец кода на языке C, в котором слово *test* просто копируется в 10-байтный символьный массив:

```
#include <string.h>

int main (int argc, char **argv)
{
   char buffer[10];
   strcpy(buffer, "test");
}
```

Затем изменим этот фрагмент кода таким образом, чтобы пользователь мог ввести его в матрицу цифр:

```
#include <string.h>
int main (int argc, char **argv)
```

```
{
   char buffer[10];
   strcpy(buffer, argv[1]);
}
```

Утечка информации об исходном коде Microsoft

Чтобы подкрепить наше утверждение о том, что анализ исходного кода не обязательно превосходит метод черного ящика, рассмотрим то, что произошло в феврале 2004 года. По Интернету начали циркулировать архивы кодов, которые, по слухам, являлись выдержками из исходных кодов операционных систем Microsoft Windows NT 4.0 и Windows 2000. Microsoft позднее подтвердила, что архивы действительно были подлинными. Многие корпорации опасались, что эта утечка скоро может привести к обнаружению множества изъянов в этих двух популярных операционных системах. Однако этого не случилось. До сего дня в просочившейся части кода обнаружена лишь горсточка изъянов. В этом кратком списке есть, например, CVE-2004-0566, который вызывает переполнение при рендеринге файлов .bmp.[1] Интересно, что Microsoft оспаривала это открытие, заявляя, что компания самостоятельно выявила данный недостаток при внутреннем аудите.[2] Почему же не обнаружилась масса изъянов? Разве доступ к исходному коду не должен был помочь выявить их все? Дело в том, что анализ исходного кода, хотя он и является очень важным компонентом проверки безопасности приложения или операционной системы, трудно бывает провести из-за больших объемов и сложности кода. Более того, метод разбиения на части может выявить те же самые недостатки. Возьмем, например, проекты TinyKRNL[3] или ReactOS[4], целью которых является обеспечение совместимости ядра и операционной системы Microsoft Windows. Разработчики этих проектов не имели доступа к исходному коду ядра Microsoft, однако сумели создать проекты, которые до какой-то степени способны обеспечить совместимую с Windows среду. При проверке операционной системы Windows вам вряд ли обеспечат доступ к ее исходному коду, однако исходный код этих проектов может быть использован как руководство при анализе ошибок Windows.

---

1 *http://archives.neohapsis.com/archives/fulldisclosure/2004-02/0806.html*

2 *http://news.zdnet.com/2100-1009_22-5160566.html*

3 *http://www.tinykrnl.org/*

4 *http://www.reactos.org/*

Оба сегмента кода используют функцию `strcpy()`, чтобы копировать данные в основанный на стеке буфер. Использование `strcpy()` обычно не рекомендуется в программировании на C/C++, так как это ограничивает возможности проверки того, какие данные скопированы в буфер. В результате, если программист не позаботится о том, чтобы выполнить проверку самостоятельно, буфер может переполниться и данные окажутся за границами заданного поля. В первом примере переполнения буфера не случится, потому что длина строки «test» (включая нулевой разделитель) есть и всегда будет равна 5, а значит, меньше 10-байтного буфера, в который она копируется. Во втором сценарии переполнение буфера может произойти или не произойти – в зависимости от значения, которое пользователь введет в командную строку. Решающим здесь будет то, контролирует ли пользователь ввод по отношению к уязвимой функции. Рудиментарная проверка кода в обоих образцах отметит строку `strcpy()` как потенциально уязвимую. Однако при отслеживании значений кода нужно понять, действительно ли существует используемое условие. Нельзя сказать, что проверки исходного кода не могут быть полезными при исследовании безопасности. Их следует проводить, когда код доступен. Однако в зависимости от вашей роли и перспектив зачастую бывает, что на этом уровне у вас нет доступа.

Часто неправильно считают, что метод белого ящика более эффективен, чем метод черного ящика. Что может быть лучше или полнее, чем доступ к исходному коду? Но помните: то, что вы видите, – это совсем не обязательно то, что вы выполняете, когда доходит до исходного кода. Процесс построения программы может внести серьезные изменения в код сборки при переработке исходного кода. И это, помимо остальных причин, уже объясняет, почему невозможно утверждать, что один подход к тестированию обязательно лучше, чем другой. Это просто различные подходы, которые обычно выявляют различные типы уязвимости. Таким образом, для всестороннего исследования, необходимо сочетать различные методы.

Инструменты и автоматизация

Инструменты анализа исходного кода обычно делятся на три категории: средства проверки на этапе компиляции, броузеры исходного кода и автоматические инструменты проверки исходного кода. Средства проверки на этапе компиляции (compile time checker) ищут изъяны уже после создания кода. Они обычно встроены в компиляторы, но их подход к проблемам безопасности отличается от подхода к функциональности приложения. Опция `/analyze` в Microsoft Visual C++ – как раз такой пример.[1] Microsoft также предлагает PREfast for Drivers[2],

[1] *http://msdn2.microsoft.com/en-us/library/k3a3hzw7.aspx*

[2] *http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx*

который способен определить различные типы уязвимостей при разработке драйверов, которые не всегда обнаруживаются компилятором.

Броузеры исходного кода – это инструменты, которые созданы для помощи при анализе исходного кода вручную. Этот тип инструментов позволяет пользователю применять улучшенный тип поиска, а также устанавливать между строками кода кросс-ссылки и двигаться по ним. Например, такой инструмент можно использовать для того, чтобы выявить все места, где встречается `strcpy()`, чтобы определить потенциально уязвимые для переполнения участки. Cscope[1] и Linux Cross-Reference[2] – популярные типы броузеров исходного кода.

Автоматические инструменты проверки исходного кода призваны просмотреть исходный код и автоматически определить зоны опасности. Как большинство инструментов проверки, они доступны как бесплатно, так и на платной основе. Кроме того, эти инструменты обычно ориентированы на определенные языки программирования, так что если ваш продукт создан с помощью разных языков, может потребоваться несколько таких программ. На коммерческой основе эти продукты предоставляются такими лабораториями, как Fortify[3], Coverity[4], KlocWork[5], GrammaTech[6] и др. В табл. 1.1 представлен список некоторых популярных бесплатных инструментов, языков, с которыми они работают, и платформ, которые они поддерживают.

Таблица 1.1. Бесплатные автоматические инструменты проверки исходного кода

Название	Языки	Платформа	Где скачать
RATS (Rough Auditing Tool for Security)	C, C++, Perl, PHP, Python	UNIX,Win32	*http://www.fortifysoftware.com/security-resources/rats.jsp*
ITS4	C, C++	UNIX,Win32	*http://www.cigital.com/its4/*
Splint	C	UNIX,Win32	*http://lclint.cs.virginia.edu/*
Flawfinder	C, C++	UNIX	*http://www.dwheeler.com/flawfinder/*
Jlint	Java	UNIX,Win32	*http://jlint.sourceforge.net/*
CodeSpy	Java	Java	*http://www.owasp.org/software/labs/codespy.htm*

[1] *http://cscope.sourceforge.net/*

[2] *http://lxr.linux.no/*

[3] *http://www.fortifysoftware.com/*

[4] *http://www.coverity.com/*

[5] *http://www.klocwork.com/*

[6] *http://www.grammatech.com/*

Важно помнить, что ни один автоматический инструмент никогда не заменит опытного тестера. Это просто средства реализации тяжелейшей задачи исследования тысяч строк исходного кода. Они помогают сберечь время и не впасть в отчаяние. Отчеты, которые создаются этими инструментами, все равно должны проверять опытный аналитик, который определит неверные результаты, и разработчики, которые, собственно, и устранят ошибку. Возьмем, например, образец отчета, который создан Rough Auditing Tool for Security (RATS) после работы с уязвимым образцом кода, приведенным ранее. RATS указывает на две возможных проблемы безопасности: использование фиксированного буфера и потенциальные опасности использования `strcpy()`. Однако это не окончательное утверждение об уязвимости. Так можно только обратить внимание пользователя на место вероятных проблем в коде, и только сам пользователь может решить, действительно ли этот код небезопасен.

```
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing userinput.c
userinput.c:4: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that
are allocated on the stack are used safely. They are prime targets
for buffer overflow attacks.

userinput.c:5: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 7
Total time 0.000131 seconds
53435 lines per second
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing userinput.c
userinput.c:4: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that
are allocated on the stack are used safely. They are prime targets
for buffer overflow attacks.

userinput.c:5: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 7
Total time 0.000794 seconds
8816 lines per second
```

За и против

Как говорилось ранее, не существует единственно верного подхода к определению изъянов в безопасности. Как же выбрать правильный метод? Что ж, иногда решение уже принято за нас. Например, метод белого ящика применить невозможно, если у нас нет доступа к исходному коду объекта. С этим чаще всего имеют дело тестеры и программисты, особенно когда они работают в среде Microsoft Windows с коммерческими программами. В чем же преимущества метода белого ящика?

- *Охват.* Поскольку весь исходный код известен, его проверка обеспечивает практически полный охват. Все пути кода могут быть проверены на возможную уязвимость. Но это, конечно, может привести и к неверным результатам, если некоторые пути кода недостижимы во время исполнения кода.

Анализ кода не всегда возможен. Даже когда его можно провести, он должен сочетаться с другими средствами установления уязвимости. У анализа исходного кода есть следующие недостатки:

- *Сложность.* Средства анализа исходного кода несовершенны и порой выдают неверные результаты. Таким образом, отчеты об ошибках, которые выдают эти инструменты, – только начало пути. Их должны просмотреть опытные программисты и определить, в каких случаях речь действительно идет об изъянах в коде. Поскольку важные программные проекты обычно содержат сотни тысяч строк кода, отчеты могут быть длинными и требовать значительного времени на просмотр.
- *Доступность.* Исходный код не всегда доступен. Хотя многие проекты UNIX поставляются с открытым кодом, который можно просмотреть, такая ситуация редко встречается в среде Win32, особенно если дело касается коммерческого продукта. А без доступа к исходному коду исключается сама возможность использования метода белого ящика.

Метод черного ящика

Метод черного ящика предполагает, что вы знаете только то, что можете наблюдать воочию. Вы как конечный пользователь контролируете то, что вводится в черный ящик, и можете наблюдать результат, получающийся на выходе, однако внутренних процессов видеть не можете. Эта ситуация чаще всего встречается при работе с удаленными веб-приложениями и веб-сервисами. Вводить данные можно в форме запросов HTML или XML, а работать на выходе с созданной веб-страницей или значением результата соответственно, но все равно вы не будете знать, что происходит внутри.

Приведем еще один пример: когда вы приобретаете приложение вроде Microsoft Office, вы обычно получаете уже сконструированное двоич-

ное приложение, а не исходный код, с помощью которого оно было построено. Ваши цели в этой ситуации определяют, какого оттенка цвета ящик нужно использовать для тестирования. Если вы не собираетесь применять технику декодирования, то эффективно исследование программы методом черного ящика. Также можно воспользоваться методом серого ящика, речь о котором впереди.

Тестирование вручную

Допустим, мы работаем с веб-приложениями. В этом случае при ручном тестировании может использоваться стандартный веб-броузер. С его помощью можно установить иерархию веб-сайтов и долго и нудно вводить потенциально опасные данные в те поля, которые нас интересуют. На ранних стадиях проверки этот способ можно использовать нерегулярно – например, добавлять одиночные кавычки к различным параметрам в ожидании того, что выявится уязвимость типа SQL-инъекции.

Тестирование приложений вручную, без помощи автоматических инструментов, обычно плохое решение (если только ваша фирма не наймет целую толпу тестеров). Единственный сценарий, при котором оно имеет смысл, – это свипинг (sweeping), поиск сходных изъянов в различных приложениях. При свипинге исходят из того, что зачастую разные программисты делают в разных программах одну и ту же ошибку. Например, если переполнение буфера обнаружено на одном сервере LDAP, то тестирование на предмет той же ошибки других серверов LDAP выявит, что они также уязвимы. Учитывая, что у программ часто бывает общий код, а программисты работают над различными проектами, такие случаи встречаются нередко.

Свипинг

`CreateProcess()` – это функция, которая используется в программном интерфейсе приложения Microsoft Windows (API). Как видно из ее названия, `CreateProcess()` начинает новый процесс и его первичный поток.[1] Прототип функции показан ниже:

```
BOOL CreateProcess(
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
```

[1] *http://msdn2.microsoft.com/en-us/library/ms682425.aspx*

```
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Известно и документально подтверждено, что если параметр `lpApplicationName` определен как `NULL`, то процесс, который будет запущен, – это первое из отделенных пробелом значений параметра `lpCommandLine`. Возьмем, к примеру, такой запрос к `CreateProcess()`:

```
CreateProcess(
    NULL,
    "c:\program files\sub dir\program.exe",
    ...
);
```

В этом случае `CreateProcess()` будет итеративно пытаться запустить каждое из следующих значений, разделенных пробелом:

```
c:\program.exe
c:\program files\sub.exe
c:\program files\sub dir\program.exe
```

Так будет продолжаться до тех пор, пока наконец не будет обнаружен исполняемый файл или не будут исчерпаны все прочие возможности. Таким образом, если файл program.exe размещается в каталоге c:\, приложения с небезопасными запросами к `CreateProcess()` вышеупомянутой структуры выполнят program.exe. Это обеспечит хакерам возможность доступа к исполнению файла, даже если формально доступ к нему отсутствует.

В ноябре 2005 года вышел бюллетень по безопасности[1], в котором упоминалось несколько популярных приложений с небезопасными запросами к `CreateProcess()`. Эти исследования были результатом успешного и очень несложного упражнения по свипингу. Если вы хотите найти сходные уязвимости, скопируйте и переименуйте простое приложение (например, notepad.exe) и поместите его в каталог c:\. Теперь пользуйтесь компьютером как обычно. Если скопированное приложение внезапно запускается, вы, судя по всему, обнаружили небезопасный запрос к `CreateProcess()`.

---

[1] *http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=340*

Автоматическое тестирование, или фаззинг

Фаззинг – это, говоря коротко, метод грубой силы, и хотя он не отличается элегантностью, но восполняет этот недостаток простотой и эффективностью. В главе 2 «Что такое фаззинг?» мы дадим определение этого термина и детально раскроем его значение. По сути, фаззинг состоит из процессов вброса в объект всего, что ни попадется под руку (кроме разве что кухонной раковины), и исследования результатов. Большинство программ создано для работы с данными особого вида, но должны быть достаточно устойчивы для того, чтобы успешно справляться с ситуациями неверного ввода данных. Рассмотрим простую веб-форму, изображенную на рис. 1.1.

Рис. 1.1. *Простая веб-форма*

Пользуются ли фаззингом в Microsoft?

Правильный ответ – да. Опубликованный компанией в марте 2005 года документ «The Trustworthy Computing Security Development Lifecycle document» (SDL)[1] показывает, что Microsoft считает фаззинг важнейшим инструментом для обнаружения изъянов в безопасности перед выпуском программы. SDL стал документом, инициирующим внедрение безопасности в жизненный цикл разработки программы, для того чтобы ответственность за безопасность касалась всех, кто так или иначе принимает участие в процессе разработки. О фаззинге в SDL говорится как о типе инструментов для проверки безопасности, которым необходимо пользоваться на стадии реализации проекта. В документе утверждается, что «особое внимание к тестированию методом фаззинга – сравнительно новое добавление к SDL, но пока результаты весьма обнадеживают».

---

[1] *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/sdl.asp*

Справедливо предположение, что поле Name (Имя) должно получить буквенную последовательность, а поле Age (Возраст) – целое число. Что случится, если пользователь случайно перепутает поля ввода и наберет слово в поле возраста? Будет ли строка букв автоматически конвертирована в число в соответствии со значениями ASCII? Появится ли сообщение об ошибке? Или приложение вообще зависнет? Фаззинг позволяет ответить на эти вопросы с помощью автоматизированного процесса. Тестеру не требуется никаких знаний о внутренних процессах приложения – таким образом, это использование метода черного ящика. Вы стоите и швыряете камни в цель, ожидая, что окно разобьется. В этом смысле фаззинг подпадает под определение черного ящика. Однако в этой книге мы покажем, как управлять грубой силой фаззинга, чтобы убедиться, что камень летит по прямой и точно в цель каждый раз и в этом фаззинг имеет нечто общее с методом серого ящика.

За и против

Метод черного ящика, хотя и не всегда является наилучшим, всегда возможен. Из преимуществ этого метода назовем следующие:

- *Доступность*. Тестирование методом черного ящика применимо всегда, и даже в ситуациях, когда доступен исходный код, метод черного ящика может дать важные результаты.
- *Воспроизводимость*. Поскольку для метода черного ящика не требуются предположения насчет объекта, тест, примененный, например, к одному FTP-серверу, можно легко перенести на любой другой FTP-сервер.
- *Простота*. Хотя такие подходы, как восстановление кода (reverse code engineering, RCE), требуют серьезных навыков, обычный метод черного ящика на самом простом уровне может работать без глубокого знания внутренних процессов приложения. Однако на деле, хотя основные изъяны можно без труда найти с помощью автоматизированных средств, обычно требуются специальные познания для того, чтобы решить, может ли разовый срыв программы развиться во что-то более интересное, например, в исполнение кода.

Несмотря на доступность метода черного ящика у него есть и некоторые недостатки. Например, следующие:

- *Охват*. Одна из главных проблем при использовании черного ящика – решить, когда закончить тестирование, и понять, насколько эффективным оно оказалось. Этот вопрос более подробно рассматривается в главе 23 «Фаззинговый трекинг».
- *Разумность*. Метод черного ящика хорош, когда применяется к сценариям, при которых изъян обусловлен разовой ошибкой ввода. Комплексная атака, однако, может развиваться по различным направлениям; некоторые из них ставят под удар испытываемое приложение, а некоторые переключают его эксплуатацию. Подобные

атаки требуют глубокого понимания внутренней логики приложения, и обычно раскрыть их можно только ручной проверкой кода или посредством RCE.

Метод серого ящика

Балансирующий между белым и черным ящиком серый ящик, по нашему определению, – это метод черного ящика в сочетании со взглядом на объект с помощью восстановления кода (reverse code engineering – RCE). Исходный код – это неоценимый ресурс, который относительно несложно прочесть и который дает четкое представление о специфической функциональности. К тому же он сообщает о данных, ввода которых ожидает функция, и о выходных данных, создания которых можно от этой функции ожидать. Но если этого важнейшего ресурса нет, не все еще потеряно. Анализ скомпилированной сборки может дать похожую картину, хотя это значительно труднее. Оценку безопасности сборки по отношению к уровню исходного кода принято называть бинарной проверкой.

Бинарная проверка

RCE часто считают аналогом бинарной проверки, но здесь мы будем понимать под RCE субкатегорию, чтобы отличить ее от полностью автоматических методов. Конечная цель RCE – определить внутреннюю функциональность созданного бинарного приложения. Хотя невозможно перевести двоичный файл обратно в исходный код, но можно обратно преобразовать последовательности инструкций сборки в формат, который лежит между исходным кодом и машинным кодом, представляющим бинарный файл(ы). Обычно эта «средняя форма» – это комбинация кода на ассемблере и графического представления исполнения кода в приложении.

Когда двоичный файл переведен в доступную для человека форму, код можно исследовать на предмет участков, которые могут содержать потенциальные изъяны, притом почти тем же способом, что и при анализе исходного кода. Так же, как и в случае с исходным кодом, обнаружение потенциально уязвимого кода – это не конец игры. Необходимо вдобавок определить, может ли пользователь воздействовать на уязвимый участок. Следуя этой логике, бинарная проверка – это техника выворота наизнанку. Сначала тестер находит интересующую его строку в дизассемблированном коде, а затем смотрит, выполняется ли этот изъян.

Восстановление кода – это хирургическая техника, которая использует такие инструменты, как дизассемблеры, декомпиляторы и дебаггеры (отладчики). Дизассемблеры преобразуют нечитаемый машинный код в ассемблерный код, так что его может просмотреть человек. Доступны различные бесплатные дизассемблеры, но для серьезной работы вам, скорее всего, понадобится потратиться на DataRescue's Interac-

tive Disassembler (IDA) Pro[1], который можно видеть на рис. 1.2. IDA – это платный дизассемблер, который работает на платформах Windows, UNIX и MacOS и способен расчленять бинарные коды множества различных архитектур.

Подобно дисассемблеру декомпилятор статически анализирует и конвертирует двоичный код в формат, понятный человеку. Вместо того чтобы напрямую переводить код в ассемблер, декомпилятор пытается создать языковые конструкции более высокого уровня – условия и циклы. Декомпиляторы не способны воспроизвести исходный код, поскольку информация, которая в нем содержалась, – комментарии, различные названия, имена функций и даже базовая структура – не сохраняются, когда исходный код скомпилирован. Декомпиляторы для языков, пользующихся машинным кодом (например, C и C++), обычно имеют значительные ограничения и по природе своей в основном экспериментальны. Пример такого декомпилятора – Boomerang.[2] Декомпиляторы чаще используются для языков, которые компилируют код в промежуточную форму байтового кода (например, C#), поскольку в скомпилированном коде остается больше деталей, а декомпиляция оттого становится более успешной.

В отличие от дисассемблеров и декомпиляторов, дебаггеры применяют динамический анализ, запуская программу-объект или присоединяясь

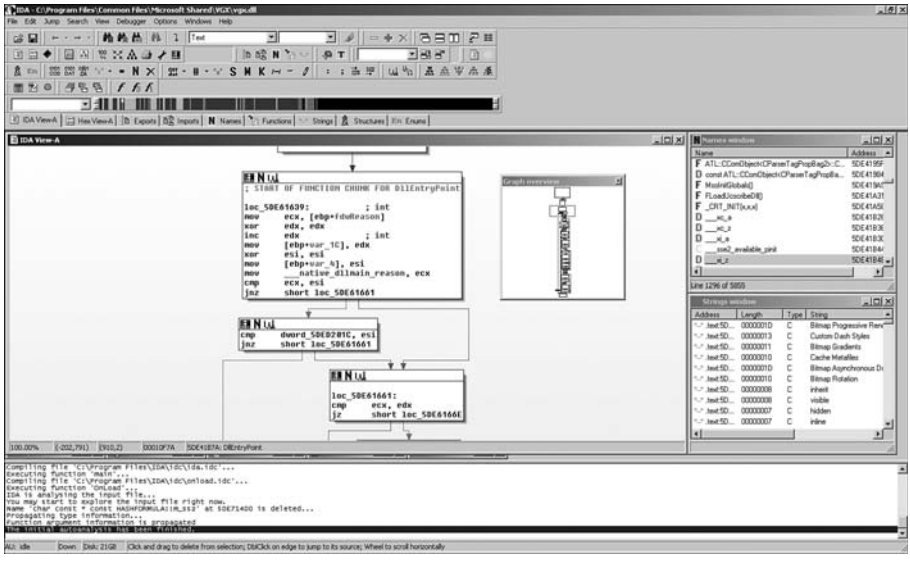

Рис. 1.2. *DataRescue IDA Pro*

[1] *http://www.datarescue.com*

[2] *http://www.boomerang.sourceforge.net/*