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 С, 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

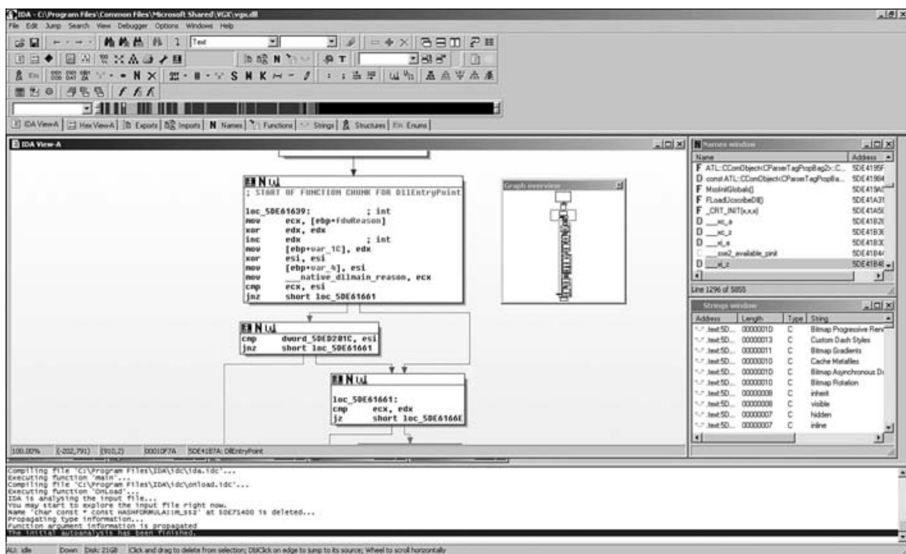

Рис. 1.2. *DataRescue IDA Pro*

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

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

к ней и отслеживая ее исполнение. Дебаггер может отражать содержимое реестра компьютера и состояние памяти во время исполнения программы. Популярными дебаггерами для платформы Win32 являются OllyDbg[1], скриншот которого можно увидеть на рис. 1.3, и Microsoft WinDbg (произносится «wind bag», дословно – болтун, пустозвон).[2] WinDbg – это элемент пакета Debugging Tools for Windows[3], и его можно бесплатно скачать с сайта Microsoft. OllyDbg – платный дебаггер, разработанный Олегом Ющуком и отличающийся несколько большим дружелюбием к пользователю. Оба дебаггера позволяют создавать стандартные расширения, а различные плагины третьего уровня могут расширить функциональность OllyDbg.[4] Для среды UNIX также существует множество дебаггеров, самый удобный и популярный из которых – GNU Project Debugger[5] (GDB). GDB работает из командной строки и включен во многие сборки UNIX/Linux.

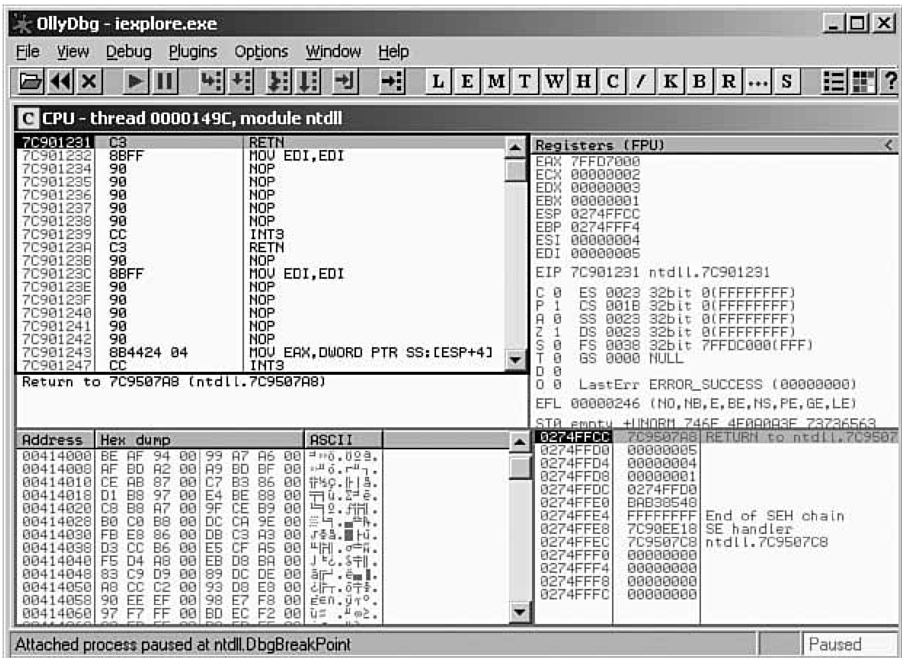

Рис. 1.3. OllyDbg

1 *http://www.ollydbg.de/*

2 *http://www.openrce.org/forums/posts/4*

3 *http://www.microsoft.com/whdc/devtools/debugging/default.mspx*

4 *http://www.openrce.org/downloads/browse/OllyDbg_Plugins*

5 *http://www.gnu.org/software/gdb/gdb.html*

Автоматическая бинарная проверка

Существует несколько инструментов, призванных автоматизировать процесс RCE и выявить возможные изъяны в безопасности двоичных приложений. Основные приложения, как коммерческие, так и бесплатные, являются либо плагинами к IDA Pro, либо самостоятельными программами. В табл. 1.2 перечислены некоторые из них.

Таблица 1.2. Инструменты автоматической бинарной проверки

Имя	**Распространитель**	**Лицензия**	**Примечания**
LogiScan	LogicLibrary	Платная	LogicLibrary приобрела BugScan в сентябре 2004 года, переименовала этот инструмент и внедрила его в решение Logidex SDA
BugScam	Halvar Flake	Бесплатная	BugScam – это коллекция скриптов IDC для IDA Pro, которые подсчитывают запросы к функциям в бинарном коде, чтобы выявить потенциально небезопасное использование различных обращений к библиотеке. Приложение было написано в основном как пародия на BugScan
Inspector	HB Gary	Платная	Inspector – это система управления RCE, которая унифицирует выходные данные с различных инструментов RCE, например IDA Pro и OllyDbg
Security-Review	Veracode	Платная	Продукт VeraCode внедряет функцию бинарного анализа непосредственно в среду разработки. Примерно так же работают инструменты анализа исходного кода, например Coverity. Анализ на двоичном уровне помогает VeraCode избежать некоторых проблем типа «Ты видишь не то, что выполняешь»
BinAudit	SABRE Security	Платная	На момент публикации этой книги BinAudit еще не вышел. Однако согласно информации веб-сайта SABRE Security это плагин для IDA Pro, который создан для нахождения изъянов в безопасности наподобие доступа вне рамок таблицы, повреждений при двойном освобождении памяти, утечек в памяти и т. д.

За и против

Как указывалось ранее, метод серого ящика – это своеобразный гибрид, который объединяет традиционное тестирование методом черного ящика с преимуществами RCE. Как и другие методы, этот имеет и достоинства, и недостатки. Среди его преимуществ такие:

- *Доступность.* Если речь идет не об удаленных веб-сервисах и приложениях, бинарные версии программ всегда доступны.
- *Охват.* Информация, полученная посредством анализа методом серого ящика, может быть применена для помощи в улучшении чистого метода черного ящика в процессе фаззинга.

Из недостатков метода серого ящика отметим:

- *Сложность.* RCE – это набор очень специальных навыков, и поэтому не всегда оказывается достаточно возможностей для его применения.

Резюме

На самом верхнем уровне методы обнаружения уязвимостей разделяются на методы белого, черного и серого ящиков. Различие между ними определяется ресурсами, которые доступны тестеру. Метод белого ящика использует все доступные ресурсы, в том числе исходный код, в то время как при методе черного ящика имеется доступ только к вводимым данным и получаемым результатам. Нечто среднее представляет собой метод серого ящика, в котором помимо информации, полученной методом черного ящика, используются и итоги анализа доступного двоичного кода с помощью RCE.

Метод белого ящика использует различные подходы к анализу исходного кода. Тестирование может проводиться вручную или с помощью автоматических средств – средств проверки на этапе компиляции, броузеров исходного кода и автоматических средств проверки исходного кода.

При использовании метода черного ящика исходный код недоступен. Фаззинг, проводимый в ходе этого метода, считается слепым. При этом вводятся данные и отслеживается реакция системы, однако неизвестны детали, касающиеся внутреннего состояния объекта. Фаззинг посредством метода серого ящика – это тот же фаззинг с помощью черного ящика плюс данные, полученные с помощью средств RCE. При фаззинге пытаются многократно вводить в приложение неожиданные данные и одновременно отслеживают исключительные ситуации, которые могут произойти в процессе получения результатов. Дальнейший материал книги посвящен фаззингу как подходу к выявлению уязвимостей в безопасности.