МАРСЕЛЬ НИЗАМУТДИНОВ

bhv

ТАКТИКА ЗАЩИТЫ И НАПАДЕНИЯ НА WEB-ПРИЛОЖЕНИЯ

- ОБЗОР УЯЗВИМОСТЕЙ В СКРИПТАХ
- ЗАЩИТА БАЗ ДАННЫХ ОТ SQL-ИНЪЕКЦИЙ
- БЕЗОПАСНАЯ АВТОРИЗАЦИЯ И АУТИФИКАЦИЯ
- XSS И ПОХИЩЕННЫЕ COOKIE
- БЕЗОПАСНОСТЬ ПРИ РАЗМЕЩЕНИИ САЙТА НА СЕРВЕРЕ ХОСТИНГОВОЙ КОМПАНИИ
- ПРАКТИЧЕСКИЕ РЕКОМЕНДАЦИИ И ПРИМЕРЫ

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Марсель Низамутдинов

ТАКТИКА ЗАЩИТЫ И НАПАДЕНИЯ НА WEB-ПРИЛОЖЕНИЯ

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.202
Н61

Низамутдинов М. Ф.

Н61 Тактика защиты и нападения на Web-приложения. — СПб.: БХВ-Петербург, 2005. — 432 с.: ил.

ISBN 5-94157-599-8

Рассмотрены вопросы обнаружения, исследования, эксплуатации и устранения уязвимостей в программном коде Web-приложений. Описаны наиболее часто встречаемые уязвимости и основные принципы написания защищенного кода. Большое внимание уделено методам защиты баз данных от SQL-инъекций. Приведены различные способы построения безопасной системы авторизации и аутификации. Рассмотрен межсайтовый скриптинг (XSS) с точки зрения построения безопасного кода при создании чатов, форумов, систем доступа к электронной почте через Web-интерфейс и др. Уделено внимание вопросам безопасности и защиты систем при размещении сайта на сервере хостинговой компании. Приведено описание вируса, размножающегося исключительно через уязвимости в Web-приложениях. Материал книги сопровождается многочисленными практическими примерами и рекомендациями.

На CD представлены примеры скриптов, описанных в книге, и необходимое программное обеспечение для их запуска, а также примеры уязвимых «тестовых сайтов», проникнуть в которые будет предложено читателю при прочтении книги.

Для Web-разработчиков

УДК 681.3.06
ББК 32.973.202

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Татьяна Лапина*
Компьютерная верстка	*Екатерины Трубниковой*
Корректор	*Татьяна Кошелева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 12.05.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 34,83.
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 5-94157-599-8

Содержание

Обращение к читателям

Уважаемые читатели!

Хотелось бы отдать вам должное за проявленный интерес к книге. Мне есть что сказать вам и есть чем поделиться. Информация, изложенная здесь, на мой взгляд, покажется любопытной не только начинающим Web-программистам, но и специалистам в этой области. Книга вами еще не прочитана, и мне бы хотелось сказать несколько слов по поводу некоторых глав.

Как вы уже поняли, ключевыми понятиями книги являются защита и нападение на Web-приложение.

Рассчитывая на вашу компетентность, очевидно, не стоит говорить о том, что вопрос безопасности Web-приложений не может решаться без детального анализа действий атакующего лица. Не зная тактики нападающего, мы рискуем быть побежденными.

Идеей моей книги является не призыв к атаке, а умение использовать ее в защите. Исходя из этих соображений, моя цель — помочь вам рассмотреть вопросы безопасности Web-приложений как с точки зрения защищающегося, так и атакующего. Достойно защищать системы мы научимся только тогда, когда узнаем "врага" в лицо, именно поэтому на каждую описанную в книге проблему дан взгляд с обеих сторон.

Прошу обратить особое внимание на главу о концептуальном вирусе. Его создание не влечет за собой никаких последствий, так как программа неспособна к размножению, а интерес к нему чисто теоретический. Моя убедительная просьба — не искать связи с вредоносной программой, а постараться извлечь максимально полезную информацию по вопросу о Web-безопасности.

Спасибо за внимание к моей книге.

Марсель Низамутдинов

Введение

Эта книга, в первую очередь, об уязвимых местах в Web-приложениях — в скриптах и программах, выполняющихся на сервере и доступных по протоколу HTTP. В книге я постарался дать подробную информацию о наиболее частых ошибках, которые допускают несведущие Web-программисты. Эти ошибки могут использоваться хакером для получения доступа к системе или для повышения своих привилегий.

Однако тема безопасности в Интернете настолько обширна и разнообразна, что рассмотреть ее в одной книге не представляется возможным. Эта книга исключительно о Web-приложениях, она не охватывает такие аспекты безопасности, как безопасность установки и конфигурирования серверного программного обеспечения, использование файрволов и антивирусов, уязвимости в исполняемых файлах, дающие взломщику системы полномочия на сервере без прохождения аутентификации. Таким образом, книга ориентирована, прежде всего, на Web-программиста, а не на системного администратора, на котором лежит ответственность за сервер в целом.

Одновременно я старался показать, что в случае неверного программирования, именно ошибки в Web-приложениях могут стать слабым звеном в цепочке выстроенной защиты сервера. Бреши в этом звене могут позволить хакеру обойти сложнейшую защиту и дать минимальные полномочия на сервере; этих полномочий будет достаточно для изучения сервера изнутри.

Сразу оговорюсь, что под словом "защита" стоит подразумевать два типа защиты:

- защита от изменения информации;
- защита от несанкционированного доступа к информации.

Представим себе небольшой Web-сайт, целиком составленный на статическом контенте. В этом случае можно сказать, что создателям сайта скрывать нечего. HTML-файлы не содержат ни паролей, ни мандатов для доступа к базе данных. Более того, они, согласно протоколу HTTP, передаются сервером пользователю как есть, безо всякой обработки или изменения.

В таких условиях утечка информации об этих файлах с сайта или сервера будет некритична, более того, например, получение доступа к этим файлам по протоколу FTP, а не HTTP, ничего не даст нападающему. Здесь гораздо опасней изменение информации, нежели получение несанкционированного доступа к ней, так как, по сути, никакой конфиденциальной информации в этом случае на сервере храниться не может. Исключение могут составить только каталоги, закрытые паролем с помощью Web-сервера.

Теперь рассмотрим более сложную систему, например, интернет-магазин. Скрипты, работающие на стороне сервера, активно взаимодействуют с базой данных. В базе данных хранится конфиденциальная информация. Это может быть информация о клиентах, поставщиках или любая другая информация, имеющая коммерческую тайну.

Но в базе данных может храниться гораздо более чувствительная к разглашению и хранению информация, например, данные о счетах пользователей.

Кроме того, сами исходные тексты скриптов будут весьма чувствительны к раскрытию. Вероятно, в этих исходных текстах будут данные, достаточные для получения доступа к базе данных, — имя пользователя и пароль. Даже если они не лежат в открытом виде, скорее всего, получить их не составит труда.

Имея на руках тексты скриптов можно анализировать вероятные уязвимые места в них с целью дальнейшего повышения привилегий или получения доступа к серверу.

Таким образом, в такой системе информация чувствительна к разглашению гораздо больше, чем на статическом сайте. Можно сказать, что хакер, найдя дыру в такой системе, вероятно, не будет производить никаких изменений, дабы остаться незамеченным, а лишь будет периодически получать данные, составляющие коммерческую тайну, для того чтобы затем использовать их с выгодой для себя.

Нападающему следует сразу определить, что он хочет сделать: изменить информацию на сервере (произвести дефейс, пополнить свой личный счет, уничтожить базу данных) или собрать максимум внутренней информации (сделать дамп базы данных, скопировать служебные файлы и т. п.).

В любом случае действия атакующего будут направлены в одну сторону — сбор максимума данных о внутреннем и внешнем устройстве сервера, получение привилегий на нем.

Web-программисту же стоит уяснить, от какого типа атак ему следует защищаться в первую очередь. В большинстве случаев следует уделить особое внимание как защите от кражи конфиденциальной информации, так и защите от изменения информации (дефейса).

Следует также иметь в виду, что хакер, используя дыры в Web-приложениях, сможет получить минимальный доступ на сервер для того, чтобы

изучать структуру сервера изнутри. Таким образом, не стоит пренебрегать защитой, даже если ценность информации на сервере минимальна и стоимость ее утраты или компрометации пренебрежимо мала. В этом случае хакер может захватить сервер исключительно для использования его вычислительных мощностей в своих целях.

Так, например, сервер может использоваться в качестве релея для отправки спама, сканирования на предмет уязвимых мест других серверов, подбора паролей к хешам и т. п.

Итак, мы выяснили основной принцип — писать максимально защищенные Web-приложения нужно всегда. Тем более, обеспечить защиту так просто! Надеюсь, прочитав эту книгу, вы поймете, как писать защищенные приложения и как использовать уязвимые места в Web-приложениях в свою пользу.

Глава 1

Интернет — "враждебная" среда

Рассмотрим некоторую систему или скрипт. Их поведение, как и поведение любого объекта в этом мире, зависит от внешних и внутренних условий. К внутренним условиям функционирования стоит отнести такие факторы, как настройки сервера, тип сервера, тип базы данных, с которой объект взаимодействует, содержание переменных окружения, содержание информации на жестком диске сервера, а также содержание самой базы данных.

К внешним же условиям следует отнести данные, отправляемые по протоколу HTTP на сервер. Такими данными являются параметры `GET`, `POST` и cookies. Кроме того, к таким данным относятся некоторые заголовки, отправляемые на сервер клиентом в рамках протокола HTTP. Эти заголовки задаются и могут быть изменены клиентом, а скрипту они передаются как переменные окружения.

К счастью, внешний пользователь, посетитель Web-сайта, не может повлиять на внутренние условия функционирования. Но он может менять внешние условия.

1.1. Динамика — прародительница всех дыр

Если рассматривать более сложную систему, то, как и всякая система, она состоит из большого количества взаимосвязанных частей. Для Web-системы, вероятно, это будут чат, система новостей, форум и т. п. Причем, стоит принять по умолчанию, что система или сайт состоят из динамического контента.

Определение

Контент — содержание чего-либо, наполнение. В данном случае — содержание HTML-страницы.

Динамический контент есть не что иное, как реакция системы на изменение ее внешних условий. Такая реакция может быть как *документированной*, т. е явно описанной или однозначно предполагаемой со стороны системы, так и являться результатом побочных явлений, происходящих в системе. Эти побочные явления могут носить непредсказуемый характер, и именно их принято называть *уязвимостями*, или дырами.

Именно динамический контент несет в себе потенциальную угрозу. Сайт, построенный целиком на статическом контенте, состоящий только из статических HTML-страниц, будет неуязвим к атакам на скрипты хотя бы потому, что скриптов там нет. По определению, статическая система никак не реагирует на изменение внешних условий, поэтому можно предположить, что и недокументированная реакция отсутствует.

Не следует думать, что статический сайт будет неуязвим абсолютно при всех типах нападений. Например, будет возможна атака через другие сервисы — через уязвимости на сторонних Web-сайтах, физически расположенных на том же сервере, однако являющихся частью другой системы. Кроме того, будут возможны атаки на сам Web-сервер.

В этой книге я рассматриваю только атаки через Web-приложения или скрипты, доступные по протоколу HTTP.

Итак, прародителем всех дыр в Web-приложениях является динамический контент. Казалось бы, стоит отказаться от динамики в Интернете, как тут же исчезает эта проблема. Однако без динамики не будет такого Интернета, каким мы его видим в настоящее время. Не будет форумов, гостевых книг, систем новостей и т. п. Таким образом, необходимо писать безопасные Web-приложения, скрипы, устойчивые системы.

1.2. Устойчивые системы

Определение

Устойчивая система — система документированно реагирующая на любое изменение внешних условий.

Как оказалось, ключом к написанию безопасных Web-приложений является это определение, "открытое" еще на младших курсах института. Система может прекрасно работать в нормальных условиях функционирования. Сообщения могут добавляться в форум, поиск — работать по всем новостям в базе данных и т. п. Более того, система может даже пройти все тесты на работоспособность в нормальных условиях. В условиях, когда пользователь не вмешивается во взаимодействие между браузером и сервером, т. е. его взаимодействие ограничивается переходами по ссылкам, отправлением форм,

содержащих адекватные данные, и т. п. И при таких условиях система может нормально и адекватно работать.

Мы видим, что взаимодействие пользователя с системой или, другими словами, изменение внешних условий функционирования системы, может быть двух типов.

Логически верные HTTP-запросы, которые могут быть объяснены здравым смыслом. Например, в качестве целочисленного идентификатора логично вставить число 0, 1, 99 и т. п. А в качестве запроса на поиск в базе данных имен корректно с точки зрения логики вписать слова, содержащие буквы русского или английского алфавита. Например "иван", "петров" и т. п.

Определение

HTTP-запрос — набор данных, посылаемых клиентом на Web-сервер согласно протоколу HTTP. Данные содержат адрес запрашиваемого скрипта, имя сервера, а также, возможно, `GET`, `POST` и cookie-параметры. Кроме того, в качестве полей заголовка клиентом могут быть отправлены некоторые второстепенные данные.

Однако в любом случае стоит протестировать систему, посмотреть, как она ведет себя, когда пользователь активно изучает отданный Web-сервером HTML-код, когда он по-своему формирует запросы, когда он в поля форм вводит символы, которые по смыслу не могут быть отправлены из этого поля.

Приведем несколько примеров.

Скрипт **HTTP://localhost/book/1/1.php** выводит имя человека из тестовой базы данных, соответствующего некоторому идентификатору. Идентификатор передается в качестве `GET`-параметра с именем `id`.

Как видим, система нормально реагирует на правильные значения идентификатора:

- **HTTP://localhost/book/1/1.php?id=1**
- **HTTP://localhost/book/1/1.php?id=2**
- **HTTP://localhost/book/1/1.php?id=100**

Таким образом, можно сделать вывод, что система правильно работает. Правильно работает в том смысле, что на корректные запросы система выдает корректные ответы.

В нашем случае видим, что на запрос без параметров выводится поле, куда можно ввести идентификатор человека. После ввода идентификатора, представляющего собой целое число, нам выводится либо имя человека, соответствующее этому идентификатору, либо сообщение о том, что запись не найдена.

Таким образом, реализован простой механизм получения информации из простой базы данных, из таблицы с двумя полями: целое число `id` — идентификатор человека и строка `name` — имя человека.

А как будет вести себя этот скрипт в несколько других условиях? Что, если поменять формат идентификатора на другой, не являющийся целым числом? В описании не сказано, как на это должна реагировать система, однако по смыслу понятно, что система должна распознать этот идентификатор как неверный и выдать ошибку. Но какая реакция последует на самом деле?

Пробуем **HTTP://localhost/1/1.php?id=a**, как мы видим, выводится ошибка:

```
Warning: mysql_fetch_object(): supplied argument is not a valid MySQL re-
sult resource in x:\localhost\1\1.php on line 15
записи не найдены
```

Что она означает и что может дать нападающему, как стоит защищаться от подобных ошибок, будет описано в следующих главах.

Вывод системной ошибки позволяет судить о том, что система некорректно реагирует на идентификатор, не являющийся целым числом.

Приведем еще один пример.

Скрипт **HTTP://localhost/1/2.php** делает то же самое, но только имя человека ищется не в базе данных, а в файле. При этом имя файла составлено из идентификатора и расширения txt.

Протестируем этот скрипт.

Делаем следующие запросы:

- **HTTP://localhost/1/2.php?id=1**
- **HTTP://localhost/1/2.php?id=2**
- **HTTP://localhost/1/2.php?id=3**

Видим, что скрипт нормально реагирует на нормальные ситуации, когда запрошен идентификатор человека, файл которого присутствует на диске.

Проверим, как ведет себя система в экстремальных ситуациях:

- **HTTP://localhost/1/2.php?id=9999**
- **HTTP://localhost/1/2.php?id=a**

В этих примерах можем видеть, что подобные запросы вызывают примерно следующие ошибки:

```
Warning: fopen(data/5.txt): failed to open stream: No such file or direc-
tory in x:\localhost\1\2.php on line 12
Warning: fread(): supplied argument is not a valid stream resource in
x:\localhost\1\2.php on line 13
Warning: fclose(): supplied argument is not a valid stream resource in
x:\localhost\1\2.php on line 15
```

Мы видим, что если идентификатор не представляет собой целое число, или даже если данному идентификатору не соответствует ни одна запись, система реагирует некорректно.

Что может дать нападающему информация, содержащаяся в тексте ошибок, будет описано в дальнейшем.

Оба приведенных примера являются примерами неустойчивой системы. Поведение скриптов предсказуемо и объяснимо, если заглянуть в текст скриптов, однако совершенно очевидно, что данное поведение не может являться документированной реакцией.

Документированной реакцией в данном случае может быть вывод скриптом сообщения об ошибке, но не сообщение интерпретатора, что в скрипте встретилась ошибка.

Множество таких примеров можно встретить и в реальной жизни. Такова особенность человеческой психики — уделять максимум внимания работе системы при нормальных внешних условиях, и почти не обращать внимания на то, что внешние условия могут быть не совсем такими или совсем не такими, как подсказывает логика.

Ключевым понятием к написанию устойчивых систем является фильтрация.

1.3. Фильтрация

Термин *фильтрация* довольно часто можно встретить в описаниях уязвимостей.

Определение

Под *фильтрацией* понимается изменение содержания некоторого параметра таким образом, чтобы избежать недокументированной реакции со стороны скрипта.

Иногда фильтрацию проводит сам скрипт до того, как этот параметр будет использован, иногда фильтрацию параметров производят дополнительные модули.

Один и тот же символ или последовательность символов, принятых от пользователя, можно отфильтровать различным образом. Например, перед использованием строки в SQL-запросе, кавычки, которые могут повлиять на ход SQL-запроса и привести к ошибке синтаксиса, можно просто вырезать из строки. Второй, более качественный, на мой взгляд, вариант — это добавить перед каждой кавычкой символ обратного слэша \. В этом случае сервер базы данных не посчитает кавычку завершающей строкой, а символ обратной косой черты не будет воспринят как часть строки.

Для демонстрации того, как SQL реагирует на символ обратной черты, можно выполнить несколько SQL-запросов:

```
mysql> select 'test - \'tested\' ';
+-----------------+
| test - 'tested' |
+-----------------+
| test - 'tested' |
+-----------------+
1 row in set (0.00 sec)
mysql>
```

Как видим, в этом примере экранированные обратным слэшем кавычки нормально вывелись в запросе. В отличие от следующего запроса, который завершился ошибкой:

```
mysql> select 'test - 'tested' ';
ERROR 1064: You have an error in your SQL syntax. Check the manual that
corresponds to your MySQL server version for the right syntax to use near
'' '' at line 1
mysql>
```

Очевидно, что различные параметры могут и должны фильтроваться поразному. В одном случае критическим может стать наличие незакрытой обратной косой кавычки в строке. В другом — привести к системной ошибке может то, что тип параметра не будет являться ожидаемым, что и произошло в запросе **HTTP://localhost/1/1.php?id=a**. В третьем случае к ошибке приводит выход параметра за некоторые допустимые пределы, точнее, то, что параметр не принадлежит ни одному из допустимых для него значений.

По сути, фильтрация может быть двух типов:

- с блокированием подозрительных значений параметров;
- с приведением к безопасным значениям параметра.

Фильтрация с блокированием сводится к тому, что если в параметре обнаружены подозрительные элементы (например, кавычка, символ < или >, которые ограничивают HTML-теги), то выполнение скрипта блокируется, а пользователю выводится сообщение об ошибке. У этого типа фильтрации есть свои недостатки. Защита с блокированием подозрительных параметров может отреагировать и на вполне адекватные значения. Например, в сообщении на форуме, в тексте сообщения может встретиться символ одиночной кавычки. Если предположить, что сообщения хранятся в базе данных, то защита обязана ответить на этот символ. В этом случае, на форуме невозможно будет отправить сообщение, содержащее одинарную кавычку.

Такое поведение защиты, хотя и вполне нормально, если вспомнить, что наличие кавычки в тексте привело бы к тому, что SQL-запрос к базе данных перестал бы быть корректным, но совершенно не оправдывается с точки зрения здравого смысла.

Фильтрация приведением к безопасному виду — оптимальный, на мой взгляд, тип фильтрации. Однако, в этом случае, все подозрительные значения будут приводиться к безопасному виду – таким образом в некоторых случаях сами значения параметров меняются. Так, например, из строковых значений параметров могут быть вырезаны одинарные и двойные кавычки, которые могут привести к возникновению синтаксической ошибки в SQL-запросе при использования таких значений без изменения.

1.4. Когда фильтрации недостаточно

Казалось бы, что полная фильтрация — это ключ к решению проблемы безопасности Web-приложений. Однако это не всегда так.

Приведем пример: **HTTP://localhost/1/3.php**. Техническое задание к написанию этого скрипта могло бы быть таким: *"Разработать скрипт, который бы выводил имя человека, соответствующего веденному идентификатору. Данные хранятся в файлах, с именем, равным этому идентификатору, и расширением txt. Например, данные человека с идентификатором 3, хранятся в файле 3.txt."*

Если данные с таким идентификатором не найдены, то об этом должно быть выведено сообщение.

Идентификатор передается методом HTTP GET. Если никакой идентификатор не найден, то скрипт должен вывести форму с предложением ввести идентификатор человека.

Рассмотрим текст скрипта:

```
<?
 if(empty($id))
 {
 echo "
 <form>
 введите id человека (целое число)<input type=text name=id>
 <input type=submit>
 </form>
 ";
 exit;
 };
 if(file_exists("data/$id.txt"))
 {
 $f=fopen("data/$id.txt", "r");
 $s=fread($f, 1024);
 echo $s;
 fclose($f);
 }
```

```
 else
 echo "записи не найдены";
?>
```

Соответствует ли скрипт спецификации, определенной в техническом задании? На этот вопрос можно ответить положительно. В техническом задании (ТЗ) весьма подробно была расписана работа скрипта, а также его реакция на возможные ошибочные ситуации: если идентификатор не найден, другими словами, не найден файл с соответствующим именем, то должно выводиться сообщение о несуществующей записи.

Одновременно в техническом задании не описано, должен ли быть идентификатор целым числом или нет.

Скрипт целиком и полностью реализует семантику поведения, заданную в ТЗ. Так, если идентификатор не передан, то выводится соответствующая форма. В случае, если идентификатор передан, то проверяется, имеется ли файл с соответствующим именем.

В случае ненайденного файла, выводится сообщение о том, что запись не обнаружена, и далее не делается никакой попытки извлечь данные из файла.

В ситуации, когда файл на диске существует, его содержимое выводится в браузер.

Казалось бы такое поведение неуязвимо. Невозможно представить себе ситуации, когда возникла бы ошибка. В любом случае, если файл не существует или имя имеет неправильный формат, то в браузер выведется сообщение об ошибке. Причем не системное сообщение интерпретатора, а сообщение, генерируемое самим скриптом.

Протестируем, так ли это. Запрашиваем:

- **HTTP://localhost/1/3.php?id=1**
- **HTTP://localhost/1/3.php?id=2**
- **HTTP://localhost/1/3.php?id=3**

Результат — выводится соответствующая запись. И даже в случае не целого, но существующего идентификатора: **HTTP://localhost/1/3.php?id=abc** вывод оправдывает ожидание — выводится соответствующая запись.

Пробуем задавать идентификаторы, отсутствующие в базе данных или содержащие символы, которые не могут присутствовать в файле (для файловой системы FAT).

Пробуем следующие запросы:

- **HTTP://localhost/1/3.php?id=999**
- **HTTP://localhost/1/3.php?id=abcde**

- **HTTP://localhost/1/3.php?id=%3F**
- **HTTP://localhost/1/3.php?id=%3C**
- **HTTP://localhost/1/3.php?id=%7C**

При этом стоит отметить, что последовательность `%3F`, `%3C`, `%7C` кодирует, соответственно, символы `?`, `<` и `|`, что соответствует передаче этих символов в качестве идентификаторов.

Как видим, даже в этом случае система нормально реагирует, выдавая программную ошибку, говорящую, что запись, соответствующая этому идентификатору, не найдена.

Однако при столь устойчивом поведении скрипта в нем существует одна ошибка, которая связана скорее с особенностью построения файловых систем.

Вспомним, какие специальные символы и последовательности используются для смены каталога, и в принципе никто не запрещает их применять в именах файлов. При этом эти последовательности будут иметь семантику смены (или обхода) каталога внутри имени файла. Последовательность `../` означает переход на уровень вверх. Посмотрим, как система реагирует на эту последовательность.

Предположим, что мы знаем, что в каталоге на уровень выше имеется файл test.txt, к которому нет доступа посредством протокола HTTP и содержание которого очень хотелось бы получить. В данном случае в качестве идентификатора человека будет последовательность `../test` . При этом `../` как раз и обозначает переход вверх по каталогу. Посмотрим по коду, какое имя файла будет проверяться на присутствие в системе и содержимое какого файла будет отправляться пользователю. Это файл data/../test.txt или, учитывая переход вверх по каталогу, это файл test.txt — то, что нам надо.

Проверим, работает ли этот фокус: **HTTP://localhost/1/3.php?id=../test**. Как видим, нам без ошибок в браузер вывелось содержание секретного файла. Почему же не сработала защита, и не вывелось сообщение, что файл в каталоге с данными не найден? Дело в том, что этот файл **действительно присутствует** в системе. Более того, имя файла нормально воспринимается такими функциями, как функция на проверку существования файла (`file_exests()`), функция открытия файла, в том числе для чтения (`fopen()`).

Очевидно это уязвимость. Причем критическая. Разберемся в причинах ее возникновения. Казалось бы, система абсолютно устойчивая, и все ситуации, которые могли бы привести к возникновению ошибок интерпретатора, исключены, однако в системе имеется несомненная дыра.

В данном случае проблема — в некорректно составленном техническом задании. Корректно составленное техническое задание должно было бы выглядеть следующим образом.

Разработать скрипт, который бы выводил имя человека, соответствующего веденному идентификатору. Данные хранятся в файлах с именем, равным этому идентификатору, и расширением txt. Например, данные человека с идентификатором 3 хранятся в файле 3.txt. Идентификатор человека является последовательностью из цифр, больших и малых символов латинского алфавита, знаков подчеркивания, минус, точка. Если переданный идентификатор не является правильным, то выводится сообщение об ошибке.

В качестве дополнительных знаков можно было бы указать и другие разрешенные файловой системой символы.

Очевидно, целиком соответствующий спецификации скрипт, описанный в таком техническом задании, будет неуязвим к подобной атаке.

1.5. Основные принципы безопасного программирования

Обозначим основные принципы написания безопасного кода и основные причины возникновения уязвимостей.

Основная причина возникновения уязвимостей в том, что пользователь может вмешиваться во взаимодействие между браузером и сервером, может отправлять на сервер не только обоснованные с логической точки зрения значения параметров.

Основной принцип звучит кратко и лаконично — *не доверяйте принятым извне данным*.

Техническое задание к написанию каждого скрипта системы должно быть лаконичным и одновременно учитывающим все опасные ситуации. Скрипт, написанный в соответствии с правильным техническим заданием, будет неуязвим для атак через Интеренет.

Даже если идея к написанию скрипта или системы рождается в голове самого программиста, если задание к написанию системы ставится человеком, не разбирающимся в безопасности, или ставится в терминологии заказчика, а не программиста, все равно стоит для себя сформировать детальное лаконичное техническое задание, учитывающее все аспекты безопасности.

Из указанного вытекает следующий принцип — *безопасность Web-приложения должна быть продумана одновременно с написанием технического задания еще до того, как написана первая строчка кода*.

Одновременно человек, составляющий техническое задание, должен разбираться в безопасности Web-приложений. Он должен совершенно ясно представлять себе, какие данные и как необходимо фильтровать, и, более того, понимать все механизмы и причины необходимости именно этой фильтрации.

Какие данные считать безопасными? В каких случаях достаточно приводить данные к правильному виду, а в каких необходимо блокировать выполнение скрипта? Как видоизменять данные, чтобы реализовать скрытую в скрипте функциональность? Как использовать в своих целях ошибки и недочеты при программировании? Обо всем этом написано в следующих главах этой книги.

Однако существует еще несколько причин возникновения уязвимостей, вина за которые целиком ложится на плечи исполняющей стороны — программистов.

Эти причины не могут быть описаны в техническом задании в большей степени потому, что они связаны с особенностью конкретного языка программирования. Как правило, любой язык программирования имеет узкие моменты, функции, возможности, использовать которые надо с большой осторожностью. Обход этих узких моментов — дело не специалиста, составляющего ТЗ, а программиста.

Например, для языка C, C++ одним из таких узких моментов является использование функций типа `printf()`, `strcpy()` и т. п. Эти функции копируют участки памяти, не проверяя, что скопированные данные могут выходить за выделенное адресное пространство. Однако это уже выходит за рамки задач, рассматриваемых в данной книге.

Для популярного языка PHP, на котором пишется солидная часть Web-приложений, одной из проблем является автоматическое определение (регистрация) глобальных переменных, на основе данных принятых как `GET`, `POST` и cookie-параметры.

Как использовать уязвимости этого типа, как устранять их и писать безопасный код, более подробно будет рассказано в следующих главах.

Глава 2

Уязвимости в скриптах

Определение

Скрипт — небольшая программа, написанная на интерпретируемом языке, например, на языке "Ы+". Скрипт может исполняться как на стороне сервера, так и на стороне клиента.

2.1. Ошибки при различных методах передачи данных

Как было сказано ранее, основой всех дыр и уязвимостей в Web-приложениях является динамический контент, то есть различная реакция скриптов на внешние условия. Если под внешними условиями функционирования понять данные, посылаемые клиентом или браузером по протоколу HTTP, то в первую очередь следует изучить, каким образом данные могут быть посланы клиентом. Как в рамках протокола HTTP передаются данные серверу. В чем разница различных типов передачи данных.

2.1.1. HTTP *GET*

HTTP `GET` является одним из самых распространенных и простых типов передачи данных от клиента серверу.

Определение

`GET` — метод передачи данных по протоколу HTTP, при котором данные передаются через адрес запрашиваемой страницы, через символ `?`.

Таким образом, `GET`-параметры можно редактировать, редактируя адрес запрашиваемой страницы. При посылке клиентом `GET`-параметры отделяются друг от друга знаком `&`. Имя параметра от значения отделяется знаком =.

Например, в адресе: **HTTP:/localhost/2/1.php?test=hello&id=2** скрипту **HTTP://localhost/2/1.php** переданы два GET-параметра. Test со значением hello и id = 2.

Метод GET имеет ограничение на максимальный размер данных. Методом GET нельзя передавать файлы.

Приведем несколько примеров отправки данных в качестве GET-параметров.

Самый простой метод — это сформировать запрос на скрипт со строкой GET-параметров внутри HTML-тега <a>.

Примеры:

1. `<a href=HTTP://localhost/2/1.php?id=21&test=hello>Test1 </a>`
2. `<a href=1.php?id=21&test=hello>Test1 </a>`
3. `<a href=/2/1.php?id=21&test=hello>Test1 </a>`

В первом примере два параметра: id = 21 и test = hello передаются методом HTTP GET скрипту **HTTP://localhost/2/1.php**.

Во втором примере эти же параметры передаются скрипту с именем 1.php в том же каталоге на том же сервере, что и текущий скрипт.

В третьем примере параметры передаются скрипту /2/1.php на том же сервере. Указан абсолютный путь от корня сервера.

Еще один пример — отправка данных из формы:

```
<form action=HTTP://localhost/2/1.php method=GET>
id: <input type=text name=id>
test: <input type=text name=test>
<input type=submit>
</form>
```

Если в заголовке формы не указан параметр action, то считается, что данные будут отправлены на текущий скрипт. Если не указан параметр method, то по умолчанию данные будут отправлены методом GET.

В этом примере можно видеть, что два HTTP GET-параметра отправляются на скрипт **HTTP://localhost/2/1.php**.

Рассмотрим данные, которые посылает клиент Web-серверу при передаче GET-параметров. Вот реальный заголовок, отправленный клиентом, — браузером Mozilla 1.7.1 в операционной системе Windows 2000.

```
GET /2/1.php?id=21&test=hello HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/20040707
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
```

```
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
```

Самой интересной тут является первая строчка. Первое слово в ней означает метод, которым были отправлены данные. Далее идет адрес скрипта относительно корня сервера, далее протокол, который был использован при передаче. В данном случае, это протокол HTTP/1.1.

Второй строчкой определяется имя сервера, откуда запрашивается скрипт.

В третьей строке браузер идентифицирует себя. В данном случае отправился тип браузера, версия операционной системы и версия браузера.

В следующих строках браузер указывает, какие типы документов он может принимать, на каком языке, какой кодировке следует отдать предпочтение, а также возможные типы сжатия при передаче документов.

Две последних строки указывают, что серверу не следует разрывать соединение после передачи запрошенного документа, и в течение какого времени следует соединение удерживать.

Столь подробное знание всех полей заголовков, которые передает сервер при `GET`-запросе, необходимо для имитации HTTP-сеансов, создании программ, которые могут запрашивать HTTP-документ на сервере, и при этом определяться на сервере как обычный браузер.

2.1.2. HTTP *POST*

Еще один метод передачи данных по протоколу HTTP, это метод `POST`.

Определение

`POST` — метод передачи по протоколу HTTP, при котором данные посылаются после того, как все заголовки будут отправлены клиентом серверу.

Отправить данные методом `POST` из HTML-страницы можно только через форму. Синтаксис формы идентичен форме для HTTP `GET`-запроса за исключением того, что метод отправки указывается `POST`.

Пример:

```
<form action=HTTP://localhost/2/1.php method=POST>
id: <input type=text name=id><br>
test: <input type=text name=test><br>
<input type=submit>
</form>
```

В данном примере два параметра `id и test` будут отправлены на скрипт методом HTTP `POST`.

Если ключевое слово `action` не указано в заголовке формы, то данные будут отправлены на текущий скрипт. Также допустимо сокращение внутренней части `action`.

Если не указан сервер, то данные будут отправлены на текущий сервер.

Если не указан сервер и путь, данные будут отправлены на скрипт, находящийся на том же сервере в том же каталоге.

Рассмотрим, какие данные посылает браузер при передаче данных методом HTTP `POST`.

Вот пример реально переданных данных браузером Mozilla:

```
POST /2/1.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/20040707
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
Referer: HTTP://localhost/2/1.php?id=21&test=hello
Content-Type: /x-www-form-urlencoded
Content-Length: 16
<пустая строка>
id=53&test=hello
```

Как видим, отличия от HTTP `GET`-запроса следующие.

В первой строке на первом месте стоит слово `POST`, обозначающее, что серверу следует ожидать `POST`-параметров.

Далее, как обычно, идет адрес запрашиваемого скрипта и протокол — HTTP/1.1, сервер, идентификация браузера, тип принимаемых страниц и т. д.

Как в `GET`- так и `POST`-запросе дополнительно может быть передано поле — `referer`. Содержание этого поля — адрес страницы, на которой клиент был до этого.

Далее идут два поля `Content-Type` и `Content-Length`. Если `GET`-запрос к серверу имел только заголовок и не имел тела (контента), то в `POST`-запросе в качестве контента выступают передаваемые по методу `POST` данные. Соответственно, необходима передача еще двух полей заголовка.

`Content-Type` — это тип данных, передаваемых в теле. В нашем случае значение этого заголовка `application/x-www-form-urlencoded` означает, что в теле следует ожидать URL-кодированных данных из www-формы.

`Content-Length` — это длина передаваемых данных. Передача этого параметра обязательна для того, чтобы сервер мог опознать, когда данные приняты полностью.

Далее идет пустая строка, обозначающая, что заголовок закончился, затем — собственно контент — значение `POST`-параметров.

URL-кодирование необходимо для того, чтобы избежать коллизий при передаче некоторых символов в данных. Например, необходимо отправить методом `POST` переменную `text` со следующим содержанием: `"help&x=y"`. Посмотрим, что произойдет, если данные отправятся без кодирования: `text=help&x=y`.

Совершенно очевидно, что скрипт разберет эту последовательность, как две переменных: `test=help` и `x=y`, в то время как была послана только одна переменная `test` со значением `"help&x=y"`.

Чтобы избавиться от подобных коллизий как при передаче данных методом `POST`, так и при передаче методом `GET`, некоторые символы кодируются специальным образом. Символ заменятся на последовательность `%XX`, где `XX` — два символа — шестнадцатеричный код заменяемого символа.

Так, например, символ `&` кодируется как `%26`, символ = как `%3D`, сам символ `%` кодируется как `%25` и т. д. В принципе, никто не запрещает кодировать при передаче все символы, однако в большинстве случаев кодируют только необходимые, так как эта операция увеличивает размер передаваемых данных.

Итак, наша строка закодируется следующим образом: `help%26x%3Dy`, и на сервер уйдет запрос `text=help%26x%3Dy`, который будет без труда правильно понят сервером.

Методом `POST` также можно передавать и файлы.

2.1.3. *GET & POST*

Теперь, когда мы ознакомились с форматом запросов, которые идут на сервер при передаче `GET`- и `POST`-параметров, можно задать вопрос. А что если скомбинировать эти методы?

Что получится, если послать на сервер следующий запрос:

```
POST /2/1.php?id=88&test=tested HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.0; en-US; rv:1.7.1) Gecko/20040707
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 3000
Connection: keep-alive
```

```
Referer: HTTP://localhost/2/1.php?id=21&test=hello
Content-Type: application/x-www-form-urlencoded
Content-Length: 16
<пустая строка>
id=53&test=hello
```

То есть, с одной стороны, указать тип запроса `POST`, а с другой стороны, передать некоторые параметры скрипту так, как были бы переданы `GET`-параметры.

Как оказалось — PHP и многие другие серверные интерпретаторы скриптов нормально реагируют на подобный запрос.

При этом параметры, переданные таким же образом, будут регистрироваться как `GET`-параметры. Например, в PHP доступ к этим параметрам можно было бы получить, обратившись к глобальной переменной `$_GET`.

Данные, переданные нами обычным образом как `POST`, регистрируются, как и следовало ожидать, как обычные `POST`-параметры. В PHP доступ к ним мог бы быть организован при помощи глобальной переменной `$_POST`.

Этот запрос к серверу составлен нами специально. А вот можно ли обычный браузер заставить сделать такой запрос к серверу. Как мы помним, в форме следует указывать только один метод передачи данных.

Теперь поступим так, как подсказывает логика, и составим следующую форму:

```
<form action=HTTP://localhost/2/1.php?id=88&test=tested method=POST>
id: <input type=text name=id><br>
test: <input type=text name=test><br>
<input type=submit>
</form>
```

Эта форма реализована в примере **HTTP://localhost/2/1.php**.

Как нетрудно убедиться, эта форма нормально воспримется и отправится браузером, и, более того, браузер составит запрос, практически идентичный приведенному.

"А для чего все это надо?" — спросите вы. И будете правы, пока не увидите следующий пример на PHP:

HTTP://localhost/2/2.php

```
<?
if(empty($_GET["id"]) || (string)(int)$_GET["id"] <> $_GET["id"] )
{
 echo "
 <form method=GET action=2.php>
```

```
 введите id человека: <input type=text name=id>
 <input type=submit>
 </form>
";
exit;
}
mysql_connect("localhost", "root", "");
mysql_select_db("book1");
$q=mysql_query("select * from test1 where id=$id");
if($r=mysql_fetch_object($q))
  echo $r->name;
else echo "записи не найдены";
?>
```

Как видим, это несколько модифицированный пример скрипта из *главы 1*. А именно, идентификатор пользователя ожидается как GET-параметр, и в самом начале скрипта происходит проверка того, что GET id является целым числом.

Однако далее используется автоматически зарегистрированная переменная id. Допустим, что PHP в данном случае сконфигурирован так, что POST-параметры для него имеют более высокий приоритет. Если это GET-запрос, то скрипт работает обычным образом, и его поведение устойчиво.

А теперь представим случай, что кроме GET- переданы и POST-параметры. Например, отправим следующую форму **HTTP://localhost/2/form1.html**:

```
<form method=POST action=2.php?id=2>
id: <input type=text name=id>
<input type=submit>
</form>
```

В данном случае формируется HTTP POST-запрос на файл 2.php с GET-параметром id равным целому числу. POST-параметр id запрашивается у пользователя.

Как видно из текста скрипта 2.php, GET-параметр id нормально пройдет фильтрацию, и управление передастся части, выполняющей запрос к базе данных.

Эта часть уже использует автоматически зарегистрированное значение параметра id, которое, согласно указанной конфигурации, берется из POST-значений. Как видим, фильтрацию проходят GET-параметры, в то время как в запрос вставляются POST-параметры. Таким образом мы обходим фильтрацию и сможем вставить в запрос произвольные данные.