

# Тема 7

## «Тестирование безопасности»

# Виды и источники угроз безопасности веб-ориентированных приложений

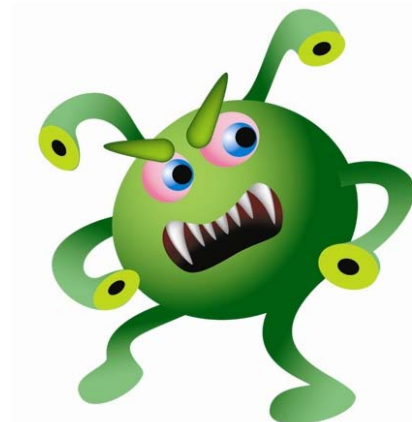
# Классификация угроз безопасности

Организация **The Web Application Security Consortium** непрерывно ведёт систематизацию и классификацию угроз безопасности веб-приложений.

Самую свежую версию списка типов угроз можно найти здесь:

<http://projects.webappsec.org/w/page/13246978/Threat-Classification>

Мы же рассмотрим самые главные направления...



# Классификация угроз безопасности

Распространённые уязвимости веб-приложений организованы в структурированный **список**, состоящий из шести классов:

- **аутентификация** (authentication);
- **авторизация** (authorisation);
- **атаки на клиентов** (client-side attacks);
- **выполнение кода** (command execution);
- **разглашение информации** (information disclosure);
- **логические атаки** (logical attacks).



# Три важных термина

Прежде чем мы приступим к рассмотрению угроз безопасности, следует определиться с **тремя важными терминами**.



**Идентификация** — это предоставление системе некоторым лицом информации для «опознавания». (*«Я – Вася Пупкин»*)



**Аутентификация** — это установление соответствия лица названному им идентификатору. (*«Я – Вася Пупкин, вот мой пароль, отпечатки пальцев... Ну Вася я! Точно!»*)



**Авторизация** — предоставление некоторому лицу возможностей в соответствии с положенными ему правами или проверка наличия прав при попытке выполнить какое-либо действие. (*«Я – Вася Пупкин, хочу посмотреть баланс своего счёта»*)

# Угрозы: аутентификация

Итак, как следует из только что рассмотренных терминов, угрозы аутентификации заключаются в том, что одно лицо (будь то человек, программа, сервис и т.п.) может выдавать себя за другое лицо.



Эта проблема проистекает из недостаточно продуманных вопросов идентификации и приводит, в свою очередь, к проблемам авторизации.

Основной вопрос, который следует задавать себе при тестировании системы на проблемы с аутентификацией: «Как я могу выполнить действие от чужого имени?»

# Угрозы: авторизация

В случае, если у системы есть проблемы с аутентификацией, ситуация становится почти неразрешимой (до устранения этих проблем).

Однако, даже при корректно работающей аутентификации **пользователь**, порой, **может выполнить действия, запрещённые для него бизнес-правилами**. Печально то, что в некоторых случаях такая проверка прав вообще не была реализована.

**Основной вопрос**, который следует задавать себе при тестировании системы на проблемы с авторизацией: **«Как я могу выполнить действие, которое не имею права выполнять?»**



# Угрозы: атаки на клиентов

Поскольку крупные информационные системы, как правило, являются хорошо защищёнными, **ограниченным в средствах злоумышленникам выгоднее организовать атаку на клиентов таких систем.**

Для этого можно использовать средства социальной инженерии (об этом – дальше в нашей теме) или уязвимости в клиентском ПО.

**Основной вопрос,** который следует задавать себе при тестировании системы на уязвимость к подобным атакам: **«Какую пользу сможет извлечь злоумышленник, если обдурит полного идиота-пользователя со старым и дырявым софтом? Как помешать злоумышленнику?»**





# Угрозы: выполнение кода

Эта атака направлена на получение злоумышленником возможности выполнить на стороне сервера вредоносный код. Последствия успешной атаки такого типа могут быть катастрофическими.

Реализация подобных атак и противодействие им зависит от конкретных применяемых технологий. Однако, чаще всего подобные проблемы связаны с недостаточной фильтрацией входных данных.

Основной вопрос, который следует задавать себе при тестировании системы на уязвимость к подобным атакам: «Есть ли хоть где-то в серверной части ПО обработка данных, которые до этого не были тщательно проверены?»



# Угрозы: разглашение информации

Суть этой атаки заключается в **сборе** злоумышленником информации, которую сообщают о себе, своих настройках и выполняемых действиях серверные приложения.

Эту информацию можно **использовать** для дальнейших атак — в т.ч. атак вида «выполнение кода».

**Основной вопрос**, который следует задавать себе при тестировании системы на уязвимость к подобным атакам: «Что наш софт рассказывает о себе? Как это проверить и устранить?»

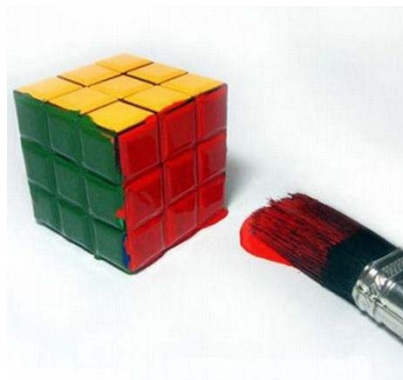


# Угрозы: логические атаки

В данном случае злоумышленник исследует систему **на предмет нарушения заданной последовательности действий** с целью получения неожиданного результата.

Уязвимость к логическим атакам может быть следствием недостаточной фильтрации данных, проблем с авторизацией и просто недостаточно продуманной архитектуры системы.

**Основной вопрос**, который следует задавать себе при тестировании системы на уязвимость к подобным атакам: **«Как можно заставить систему повести себя неожиданным образом?»** (Здесь хорошо помогает исследовательское тестирование.)



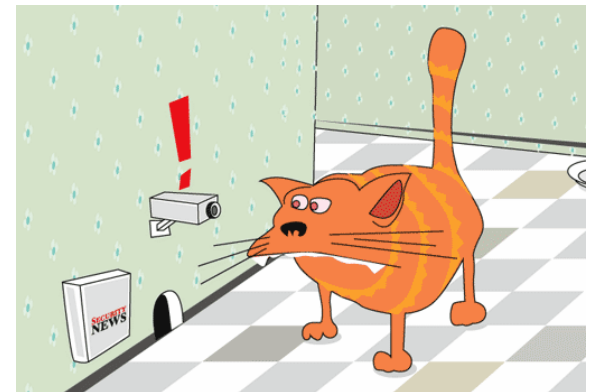
# Промежуточный итог

Тестирование безопасности требует **глубоких знаний** в области применяемых для построения и эксплуатации веб-ориентированных приложений **технологий**.

В каждом конкретном случае **глубокое исследование** вопросов безопасности **носит совершенно уникальный характер**.

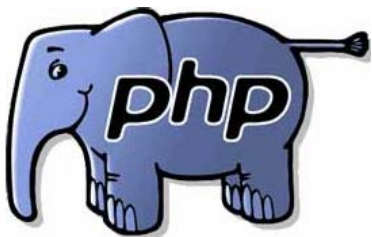
Однако, мы постараемся рассмотреть **несколько идей**, **которые являются достаточно универсальными** для большинства приложений.

Итак...



# Универсальные подходы к обеспечению и тестированию безопасности веб-ориентированных приложений

# Краткое предисловие



Рассмотренные в данной части темы советы и идеи относятся, по большей части, к тестированию приложений, **написанных на PHP**.

Эти идеи в равной мере **подходят разработчикам** по принципу «что делать» **и тестировщикам** по принципу «что проверять».



# Всё контролируем сами!

Законы Мерфи гласят, что **если что-то может выйти из строя, – оно выйдет из строя**. Это справедливо для любой разработки и любого тестирования вообще, не говоря уже о вопросах безопасности.

Как только какая-то операция выполняется в надежде на то, что «что-то будет так-то и так-то» – это **потенциальная проблема**, потенциальная дыра в безопасности.

Поэтому – **проверяем всё, что только можно**. Даже излишние на первый взгляд проверки иногда по факту оказываются очень полезными. К этому же пункту относится мысль о том, что нет ничего лучше своего кода, написанного «от и до своими руками» (руками коллег), в отличие от готовых библиотек сомнительного происхождения.



*Из того, что ты – параноик, не следует, что за тобой не следят*

# Одна точка входа



Охранять одну дверь проще, чем сто дверей.  
Так заложите все, кроме одной, кирпичом 😊

Т.е. **оставьте одну точку входа на сайт**, перехватывая обращения к ЛЮБОМУ URL'у, а затем обрабатывая запрос «своими силами». Затратно? Никак не затратнее устранения последствий нарушения безопасности.

И на производительность влияет не так сильно, как могло бы казаться.



# Одна точка входа



Как это сделать:

а) В .htaccess

RewriteEngine On

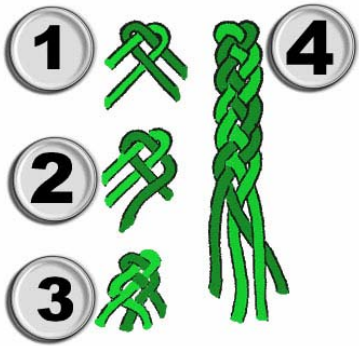
RewriteBase /

RewriteRule .\* index.php?url=\$0 [QSA,L]

б) В скрипте, анализирующем запрошенный URL, фильтруем его, т.е. приводим в нижний регистр и убираем последовательности вида %hh; идущие подряд две точки (".."); всё, что не является буквой английского алфавита, цифрой или знаками "\_./".

```
$url = preg_replace("/%{2}\\.\\.|^[^a-z\d_\.V]/", "",  
strtolower($url)); if (preg_match("/%{2}\\.\\.|^[^a-  
z\d_\.V]/", $url)==1) { // нашёлся кто-то слишком  
умный, посылаем подальше }
```

# Одна точка входа



Вторая проверка нужна против гениальных идей с комбинациями удаляемых значений, когда удаление части текста снова приводит к появлению недопустимой комбинации.

Цикл лучше не делать, т.к. (да-да, мы должны быть параноиками) цикл может стать вечным.

в) Если запрошенный URL не заканчивается на слеш, получаем полное имя запрошенного файла в виде «путь к приложению»+URL.

Проверяем, можно ли с нашего сайта запрашивать такие файлы (по расширению).

Если да, то проверяем, есть ли такой файл в файловой системе. Если да, то выставляем правильный MIME-тип в заголовке «Content-type» и отдаём файл функцией `fpasssthru()`, после чего завершаем скрипт.

# Одна точка входа



Если такого файла нет, **возможно, посетитель запросил не файл, а страницу** сайта (проверим в дальнейшей работе движка).

Если файлы с таким расширением нельзя запрашивать, **логируем ошибку и прекращаем работу.**

**ВНИМАНИЕ!** Злоумышленник может передать переменную url через GET-запрос **в обход mod\_rewrite**. Поэтому можно брать информацию о запрошенном ресурсе из `$_SERVER['REQUEST_URI']` и анализировать.

Однако, переданное злоумышленником всё равно попадёт в `$_GET['url']` и будет проанализировано, как показано выше.

# Одна точка входа



**ВНИМАНИЕ!** Помните, что злоумышленник может специально передать некий вредоносный код (например, на JavaScript) в надежде, что он где-то запротоколируется, а потом выполнится при просмотре логов.

Также может быть передан, например, PHP-код в надежде про-include'ить его потом и выполнить каким-то нетривиальным способом.

**Вывод:** данные, попадающие в лог, тоже должны подвергаться тщательной фильтрации и обработке.

Если по какой-то причине создать одну точку входа на сайт нельзя или нежелательно, **тщательно проверяйте во всех скриптах, действительно ли они вызваны должным образом** (вошёл ли пользователь в систему, достаточно ли у него прав для выполнения требуемой операции, не пытается ли кто-то вызвать скрипт в обход системы авторизации или иного механизма обеспечения безопасности).

# Одна точка входа

В контексте этого примечания хочется дать совет «очень начинающим»: если вы не уверены, что механизм защиты системы администрирования вашего сайта достаточно надёжен, используйте HTTP-авторизацию средствами `.htaccess` на доступ ко всему каталогу, в котором лежат скрипты системы администрирования.



# Скрываем уязвимые данные

В подавляющем большинстве случаев приложение будет установлено в некоторый подкаталог домашнего каталога пользователя (т.е., например, в /users/mycoolsiteuser/public\_html/).

Это значит, что у вас есть как минимум один уровень иерархии файловой системы, доступный вашим скриптам, но заведомо недоступный по HTTP.

Имеет смысл разместить файлы с паролями к БД и прочими важными данными в этом «защищённом от доступа по HTTP» каталоге.



# Скрываем уязвимые данные

К этому же пункту относится мысль о том, что думать следует не только о том, как ведут себя ваши скрипты, но и **о среде, в которой они работают.**

Да, операционных систем очень много, и специфика везде своя, но **проверить права доступа на те или иные каталоги и файлы можно (и нужно!) везде.**



# Скрываем уязвимые данные

Несмотря на небольшую распространённость, также существует угроза прочтения злоумышленником файлов сессий с сервера.

Потому – стараемся даже в сессиях не хранить никаких паролей, номеров кредитных карт и прочих полезных для злоумышленника вещей. Само собой разумеется, что такие данные не имеют права появляться в куки.





# Скрываем уязвимые данные

И ещё одно чудесное правило: **НИКАКИХ ОТНОСИТЕЛЬНЫХ ПУТЕЙ** в файловой системе.

Ваше приложение при инсталляции должно прописать себе в конфигурационный файл пути ко всем своим каталогам (главному, с шаблонами, с обработчиками и т.п.) и **ОБЯЗАТЕЛЬНО** добавлять их **в начало путей к соответствующим файлам**, а потом ещё и обрабатывать полученное функцией `realpath()`.



# Проверяем данные



Любой текст (будь то часть URL'а, поле формы, куки или что-то иное) проверяем на наличие недопустимых символов, тегов и т.п.

```
if (preg_match("/ПЛОХИЕСИМВОЛЫ/", $text)==1)
{
    // посылаем подальше
}
```



# Проверяем данные



Любые данные проверяем на длину (как правило, проблемы возникают, когда длина равна нулю или превышает некоторое допустимое значение).

Помним, что для мультибайтовых кодировок длина текста в символах может не совпадать с длиной текста в байтах, т.е.:

```
mb_strlen('Ляляля', 'UTF-8?') != strlen('Ляляля')
```



# Проверяем данные



Числа проверяем на разрядность. Если число извлекается из строки, сначала проверяем его длину в количестве символов, чтобы нам не передали «10-мегабайтное число», которое гарантированно не поместится ни в какой числовой тип данных.

```
if (strlen($num)>8)
{
    echo "Ну сказано же: от 0 до 99999999";
}
```



# Проверяем данные



Числа также проверяем на неравенство нулю и неотрицательность (если эти два свойства важны в смысловом контексте числа).



Если мы собираемся передать **данные в БД**, то **проверяем их особенно тщательно**: недопустимые символы, длины, кодировки, формат и т.п. Об этом - чуть ниже.



# Проверяем данные



Если мы получили файл, имеет смысл проверить его **размер**, **имя** (только латиница, нижний регистр, буквы, цифры, знак подчёркивания, одна точка, обязательное расширение из списка допустимых).

**Если имя не соответствует требуемому – его можно привести к таковому.** А некоторые разработчики рекомендуют не извращаться, а генерировать случайное имя: исходное имя в такой ситуации, при необходимости, можно хранить отдельно, например в БД, но проверьте сначала, чтобы файл не назывался "delete from admins" ;).

В особо опасных ситуациях **имеет смысл проверить не только имя, но и содержимое файла.**

# Проверяем данные

Данные, которые будут отображены на страницах сайта, фильтруем на предмет HTML/JS/CSS – короче, на предмет любых тегов и всего того, что может выполняться на стороне клиента. Иначе банальное

```
<script language="JavaScript">  
  window.location='http://www.saitstroyanamiivirusami.com'  
</script>
```

в сообщении форума или комментарии блога приведёт к очень печальным последствиям для ваших пользователей.

Если же JavaScript выполнится в системе администрирования, ему ничто не мешает произвести с вашим приложением любые действия от имени вошедшего в систему администратора.



# Проверяем данные

Во избежание атак на средства отправки почты (всевозможные формы обратной связи) проверяем на наличие (и удаляем) символы "\r" и "\n" в любых полях, полученных из формы (кроме текста сообщения).

Иными словами, форма обратной связи должна позволять выбрать, кому пишется письмо, ввести заголовок и текст сообщения, которые подвергаются фильтрации.

На всякий случай напомним простую истину: список «кому» должен содержать в своих значениях идентификаторы адресатов, а не непосредственно e-mail'ы.





# Помним о производительности

Чтобы сделать гадость, сайт не обязательно ломать инъекциями кода или чем-то подобным. Иногда его достаточно просто нагрузить (забавно, что это может произойти и просто из-за наплыва посетителей, не имеющих никакого злого умысла).

Поэтому ещё в процессе разработки добавляем в код отладочный механизм, показывающий нам, сколько времени генерировалась страница и выполнялись запросы к БД, сколько при этом было использовано памяти и т.п.

Любые подозрительно большие значения в будущем выльются в проблему. На стадии реальной эксплуатации сайта эти же параметры можно использовать для предсказания того, что «конец близок», и принятия мер: использования дополнительного кэширования, оповещения администратора и т.п.



# Помним о производительности

Повысить производительность помогает **грамотная архитектура проекта**, позволяющая собирать и выполнять только те скрипты, которые необходимы для генерации данной конкретной страницы или выполнения данной конкретной операции (т.е. **НЕ НАДО** собирать для любого чиха мухи **ВСЕ** скрипты в один большущий мега-скрипт `include'ами`).

Да, само собой, **следует** провести **полноценное нагрузочное тестирование** (мы рассматривали его в прошлой теме).



# Адаптация к изменениям



Иногда угроза приходит оттуда, откуда не ждали. Действия администраторов сервера или просто накапливающиеся за время работы служебные данные могут стать причиной падения приложения.



Периодически проверяем количество доступного дискового пространства. Логи, закаченные посетителями файлы, незабранная почта – всё это съедает доступное место очень быстро.

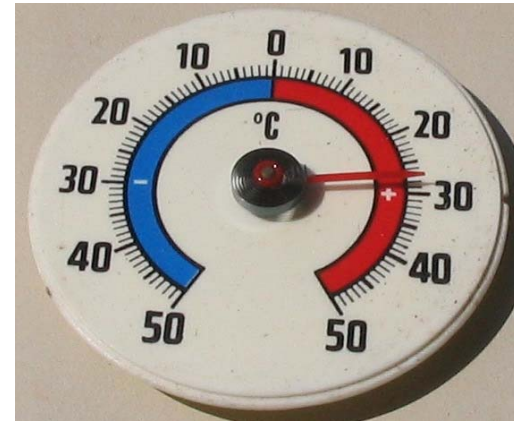
Как только его количество опустилось ниже некоторого критического значения – оповещаем администратора, а если ситуация совсем плачевная – автоматически удаляем то, чем можно пожертвовать (старые логи, почта в спам-боксах и т.п.)

# Адаптация к изменениям

Приложение должно адаптироваться к настройкам среды или хотя бы проверять их.

Можно при инсталляции собрать информацию о критичных для работы приложения параметрах среды, а затем по крону (раз в сутки, например) **проверять, не изменились ли эти параметры**; если изменились — уведомлять администратора.

Получить настройки PHP можно с помощью функций `ini_get()` или `ini_get_all()`.



# Следим за настройками

Критические для безопасности и вообще работы настройки изменяем на нужные нам (везде, где это возможно), используя функцию `ini_set()`:

**`register_globals`** – этой проблеме уже много лет: для обратной совместимости некоторые хостеры включают эту опцию, что позволяет злоумышленникам нарушить логику работы некачественных скриптов.

**`error_reporting` и `error_reporting(0)`** – есть такая опция, есть такая функция: помним, что на стадии реальной эксплуатации приложения **никакие сообщения от интерпретатора PHP не должны быть показаны посетителям.**



# Следим за настройками

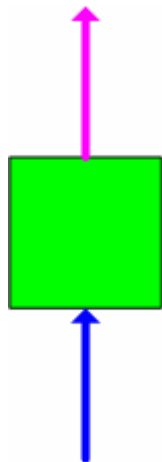
**display\_errors** — дополнение к предыдущему пункту: ТОЖЕ МОЖНО ВЫКЛЮЧИТЬ.

**allow\_url\_fopen** и **allow\_url\_include** — выключаем; помним, что в любые include'ы или require'ы можно передавать **ТОЛЬКО строковые константы** или переменные, полученные из надёжного источника (например, настроек сайта в БД).

**short\_open\_tag** — тоже может создать вам проблему, если вы используете открывающий тег "<?" вместо рекомендуемого "<?php". Так что надо привыкать использовать то, что **работает всегда по умолчанию**: "<?php".



# Следим за настройками



**output\_buffering** — при разработке обязательно отключить, чтобы полнее проверить работу всех функций, модифицирующих заголовки HTTP-ответа.

**max\_execution\_time, max\_input\_time, default\_socket\_timeout** — истечение любого из этих таймаутов приводит к **неприятным последствиям, порой фатальным**.

Потому определяем значения этих настроек и учитываем при выполнении любой достаточно долгой операции.

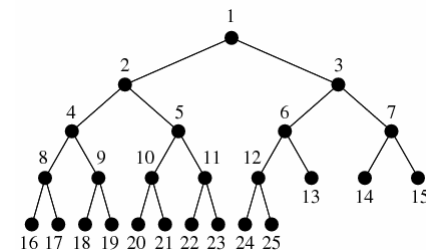


# Следим за настройками

**memory\_limit** – скрипты на PHP не очень прожорливы в плане используемой памяти: до тех пор, пока кому-то не придёт в голову «чудесная» идея дёрнуть из БД или с диска в память многосотмегабайтный запрос или файл.

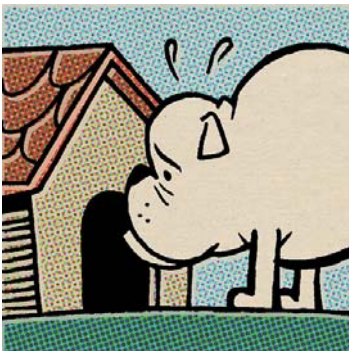
И всё. Интерпретатор не простит вашему скрипту такую вольность. Поэтому – перед «прожорливой» операцией **имеет смысл проверить, хватит ли нам памяти** для её выполнения.

**max\_input\_nesting\_level** – не давайте злоумышленнику передать вам миллиардомерный массив 😊



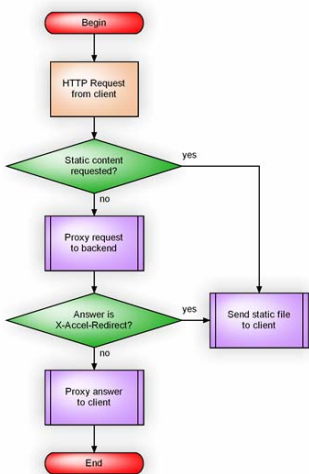


# Следим за настройками



**post\_max\_size, upload\_max\_filesize** – если вы хотите, чтобы ваши пользователи могли закладывать файлы, не натываясь на каждом шагу на проблемы, **подправьте эти настройки** так, чтобы их значения соответствовали вашим ожиданиям.

**session.use\_trans\_sid, session.use\_only\_cookies, session.bug\_compat\_42, session.bug\_compat\_warn** – неверные значения этих параметров порой приводят к **частичной или полной неработоспособности сессий**.



# Не облегчайте взлом!

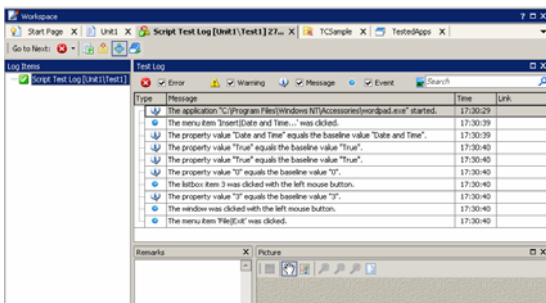
Иногда для быстроты работы с проектом разработчики заводят некие «магические значения»: пользователей с простыми паролями и полными правами, ссылки для входа в систему без аутентификации, жёстко прописывают в код «всегда подходящие пароли» и используют прочие «заглушки».

Если забыть удалить что-то подобное на стадии эксплуатации, ничего хорошего не предвидится.

А лучше – изначально этого не делать.



# Всё протоколируем!



Любые подозрительные ситуации имеет смысл **протоколировать**.

Обращения к несуществующим объектам файловой системы (или таким, доступ к которым закрыт), ошибки аутентификации, обращения к несуществующим страницам сайта, ошибки отправки файлов, форм и т.п.

**Всё это может быть попыткой взлома.**

Изучая подобные логи, **можно найти слабые места в безопасности** вашего приложения. Про проблему с логами уже дважды сказано выше, не буду повторяться.



# SQL-инъекции



Источниками SQL-инъекций могут являться **любые внешние данные**, а также (в худшем случае) **неверное поведение самого приложения** (вызванное внешними воздействиями или внутренними сбоями).

Что **проверяем**:

- а) Запрошенный **URL** (сам URL и переменные GET).
- б) **Формы**: содержимое полей, наличие необходимых и отсутствие лишних полей.
- в) **Куки**.

# SQL-инъекции



Как **защищаемся**:

Прежде, чем передать что-то в СУБД, вы должны гарантированно **убедиться, что в этих данных нет ничего опасного**. Применяем **фильтрацию данных** везде, где это возможно, по самой полной программе. Очень хорошо в этом плане работают **регулярные выражения**.

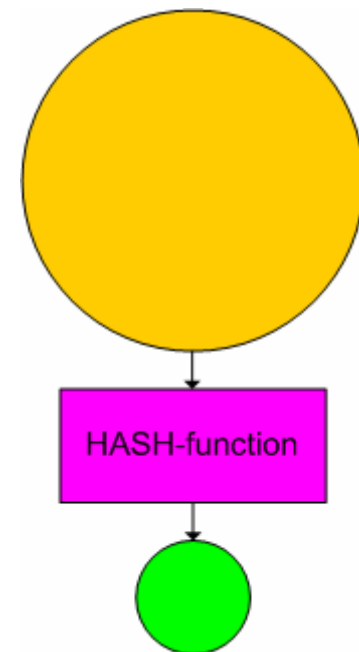
Применяем функцию **addslashes()** (при этом помним о параметре `magic_quotes_gpc`: если он включён, применение этой функции даст «двойное экранирование»).

Применяем функцию **`mysql_real_escape_string()`**.

# SQL-инъекции

Прекрасный способ защититься от получения каких бы то ни было опасных для СУБД данных — **использование хэширования**. Например, строка с sha1-хэшем уже явно не будет выполнена как вредоносный запрос.

Иногда оправданной является **модификация алгоритмов работы приложения с целью обработки данных на стороне бизнес-логики**, а не на стороне данных: т.е. некий константный запрос проводит только выборку, а затем PHP-код проводит анализ. Альтернатива этого — **использование хранимых подпрограмм** (процедур и функций).



# SQL-инъекции



Для всего, где идёт **ТОЛЬКО** выборка, имеет смысл создавать т.н. «вьюшки» (представление, view), для которых **заводить отдельного пользователя БД с минимальными правами.**



Перед выполнением запроса следует оценить (чаще всего – это возможно), **сколько памяти и времени нам понадобится для его выполнения.** Проверяем это и не даём выполнять «долгоиграющие тяжёлые запросы».

# Закрываем обходные пути

Итак, вы купили хостинг, написали **более-менее защищённое** приложение и разместили его в сети.

Сделайте это **СО СВОЕГО компьютера**, чистого от вирусов и троянов, перехватывающих пароли.

Сделайте это **в сети**, администратор которой **не анализирует трафик** с целью перехватить чужие логины-пароли.

Создайте **отдельную учётную запись для работы по FTP**, логин и пароль которой не совпадают с логином и паролем учётной записи у хостера.

То же справедливо для **любых других учётных записей**: в СУБД, в почте и т.п.





# Тест для проверки изученного

1. Какие основные классы угроз безопасности веб-приложений вы знаете?
2. В чём разница между «идентификацией», «аутентификацией» и «авторизацией»?
3. Что такое SQL-инъекция? Какие способы противостояния SQL-инъекциям вы знаете?
4. Чем опасно отсутствие фильтрации **выходных** данных приложения?
5. Какие преимущества в контексте безопасности позволяет получить организация одной точки входа в приложение?
6. Какую пользу в контексте безопасности даёт протоколирование действий пользователя и приложения?
7. Ухудшение каких показателей качества может привести к проблемам с безопасностью?
8. Какие уязвимости появляются в приложении в случае отсутствия фильтрации данных, вводимых пользователем, а затем отображаемых в приложении (на форумах, блогах и т.п.)?
9. Какие атаки могут эксплуатировать уязвимости приложения, связанные с файловой системой?
10. Как приложение может противостоять атакам на внутреннюю логику?

# Планирование тестов безопасности

# Что следует учесть

В самом начале данной темы мы рассмотрели основные **виды и источники угроз безопасности** веб-ориентированных приложений.

Следует проанализировать, **какие из этих угроз актуальны** для нашего приложения, и выяснить **степень их опасности**.

После этого **основными направлениями исследования** будут являться:

- **Интерфейсы** приложения.
- Внутренняя **логика** приложения.
- **Среда** исполнения приложения.



# Что следует учесть: интерфейсы

Под интерфейсами в данном контексте мы понимаем **ВСЕ способы взаимодействия** нашего приложения с внешним миром: пользовательский интерфейс, программные интерфейсы, работа с сетью и т.п.

Каждый интерфейс рассматривается с точки зрения того, какую информацию злоумышленник может передать **В приложение** и извлечь **ИЗ приложения** с его помощью.

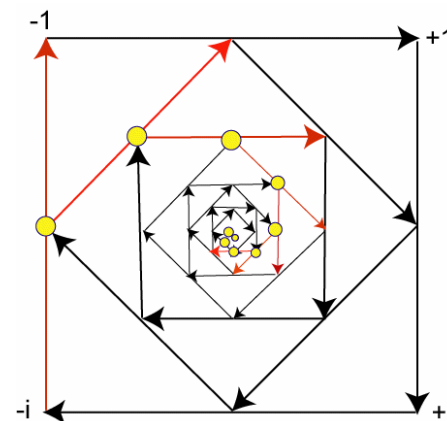
**Защита должна работать в обоих направлениях.**



# Что следует учесть: логика

Это **самый сложный этап** тестирования безопасности. К сожалению, здесь нет универсальных решений, кроме общей рекомендации **по одновременному применению методов белого и чёрного ящика**.

В приложении, к безопасности которого предъявляются повышенные требования, **вопросы защищённости должны рассматриваться на стадии формирования архитектуры**, что значительно упростит дальнейшее тестирование безопасности в контексте атак на внутреннюю логику.



# Что следует учесть: среда

Среда исполнения (в частности, операционная система, сервер приложений, СУБД, веб-сервер и т.п.) может стать причиной успешных атак, если она:

- изначально небезопасна;
- неверно сконфигурирована;
- устарела или, наоборот, только выпущена и недостаточно исследована.

Во многом здесь тестирование безопасности пересекается с вопросами грамотного сетевого администрирования, однако следует помнить, что качественное приложение способно само провести диагностику среды исполнения и выдать сообщение о найденных проблемах.



# Пример планирования...



Сейчас мы рассмотрим несколько подходов к планированию тестовых испытаний на примере тестирования безопасности функций сайта «Авторизация пользователя» и «Поиск по сайту».

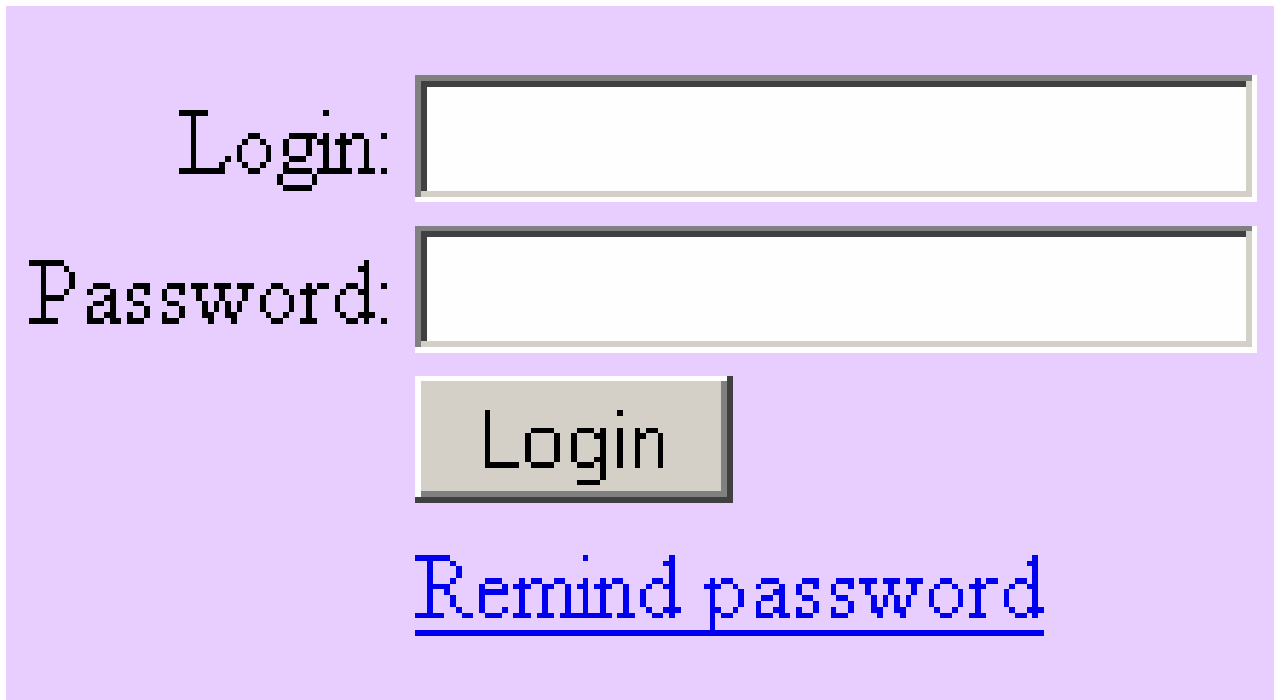
Пример тестирования безопасности на основе функций сайта «Авторизация пользователя» и «Поиск по сайту»



# «Авторизация пользователя»

Итак, перед нами форма, с помощью которой пользователь может **авторизоваться на сайте**.

**Что мы проверим**, чтобы убедиться в безопасности этой части приложения?



Login:

Password:

[Remind password](#)

# «Авторизация пользователя»

В первую очередь следует провести **несколько простых позитивных и негативных тестов**, чтобы убедиться, что:

- Система **пускает пользователей** с верной парой «логин-пароль».
- Система **НЕ пускает пользователей** с неверным логином и/или паролем.
- Система **фиксирует количество попыток входа** для каждого логина, каждого ip (уникального посетителя; сразу возникает вопрос, как она определяет, что выполнить вход под другим именем пытается тот же пользователь, у которого не прошли попытки с другим логином). По факту достижения порогового значения попыток входа данная **операция должна стать недоступной** данному пользователю (на некоторое время или до вмешательства администратора).

# «Авторизация пользователя»

С точки зрения анализа интерфейсов следует выяснить:

- Какие сообщения и в каких ситуациях выдаёт система. Могут ли эти сообщения помочь злоумышленнику.
- Можно ли создать ситуацию, когда система выдаст сообщения от сервера приложения, веб-сервера, СУБД.
- Способна ли система противостоять попыткам автоматизированного взлома (наличие CAPTCHA; привязка к сессии и т.п.)
- Где хранится информация о том, что пользователь успешно авторизован. Можно ли эту информацию украсть или изменить.

Суровая правда жизни с [bash.org.ru](http://bash.org.ru): «Извините, введенный вами пароль нельзя использовать. Такой пароль уже использует пользователь vasya99».

# «Авторизация пользователя»

С точки зрения **внутренней логики**:

- Система должна проверять, переданы ли ей из формы **И логин, И пароль**.
- Пароль должен **хэшироваться**.
- Перед передачей в БД данные должны **проверяться** на:
  - наличие;
  - длину;
  - отсутствие недопустимых символов и их последовательностей.
- В системе **не должно быть никаких «заглушек»** в виде стандартных коротких логинов/паролей, используемых разработчиками для ускорения работы.

# «Авторизация пользователя»

С точки зрения среды:

- Легко ли перехватить данные, передаваемые по сети.
- Можно ли обойти авторизацию или выполнить какие-то действия в системе от имени авторизованного пользователя, не выполняя авторизацию.
- Можно ли добавить в систему пользователя с необходимыми злоумышленнику реквизитами.
- Какие уязвимости известны в используемом серверном ПО.



# «Авторизация пользователя»

Итак, некоторые идеи мы рассмотрели. Теперь **ваша очередь предлагать тесты**. Поехали...

И помним:

- интерфейсы;
- логика;
- среда.



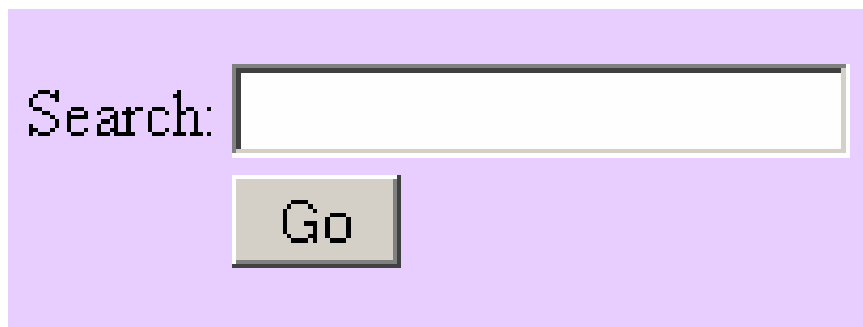
Login:

Password:

[Remind password](#)

# «Поиск по сайту»

С авторизацией немного разобрались. Теперь поговорим о **поиске по сайту**.



A search form on a light purple background. It consists of a text input field with a thin black border and a small 'x' icon in the top right corner. To the left of the input field is the label 'Search:'. Below the input field is a grey button with a black border and the text 'Go' in a monospaced font.

# «Поиск по сайту»

С точки зрения **интерфейса**:

- Какие **сообщения** и в каких **ситуациях** выдаёт система. Могут ли эти сообщения помочь злоумышленнику.
- Можно ли создать **ситуацию, когда система выдаст сообщения от сервера приложения, веб-сервера, СУБД.**

*Как вы могли заметить, эти пункты полностью идентичны ситуации с «Авторизацией пользователя».*

- Можно ли, выполняя различные запросы, **определить реализацию поискового механизма и его слабые стороны.**



# «Поиск по сайту»

С точки зрения **внутренней логики**:

- Как система противостоит **SQL-инъекциям**.
- Как система борется с **DOS-атаками** (насколько, вообще, процедура поиска **загружает аппаратные ресурсы**).
- Противостоит ли система **поиску по шаблонам, возвращающим чрезмерно большое количество совпадений**.
- Исключены ли из результатов поиска **страницы и данные**, не предназначенные для пользователей, не являющихся администраторами.

# «Поиск по сайту»

С точки зрения среды:

- Можно ли, выполняя какие бы то ни было поисковые запросы, **нанести вред среде выполнения приложения**.
- Может ли злоумышленник **изменить процесс поиска** нужным ему образом, воздействуя на среду.
- Какие **уязвимости известны в используемом серверном ПО** (*да, совпадает с «Авторизацией пользователя»*).

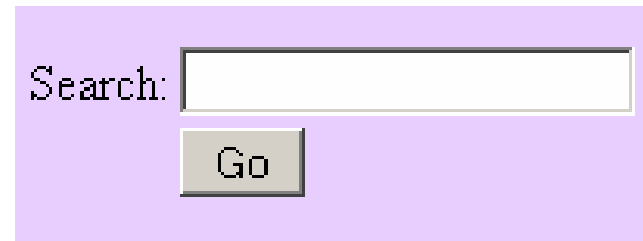


# «Поиск по сайту»

Итак, некоторые идеи мы рассмотрели. Теперь **ваша очередь предлагать тесты**. Поехали...

И помним:

- интерфейсы;
- логика;
- среда.

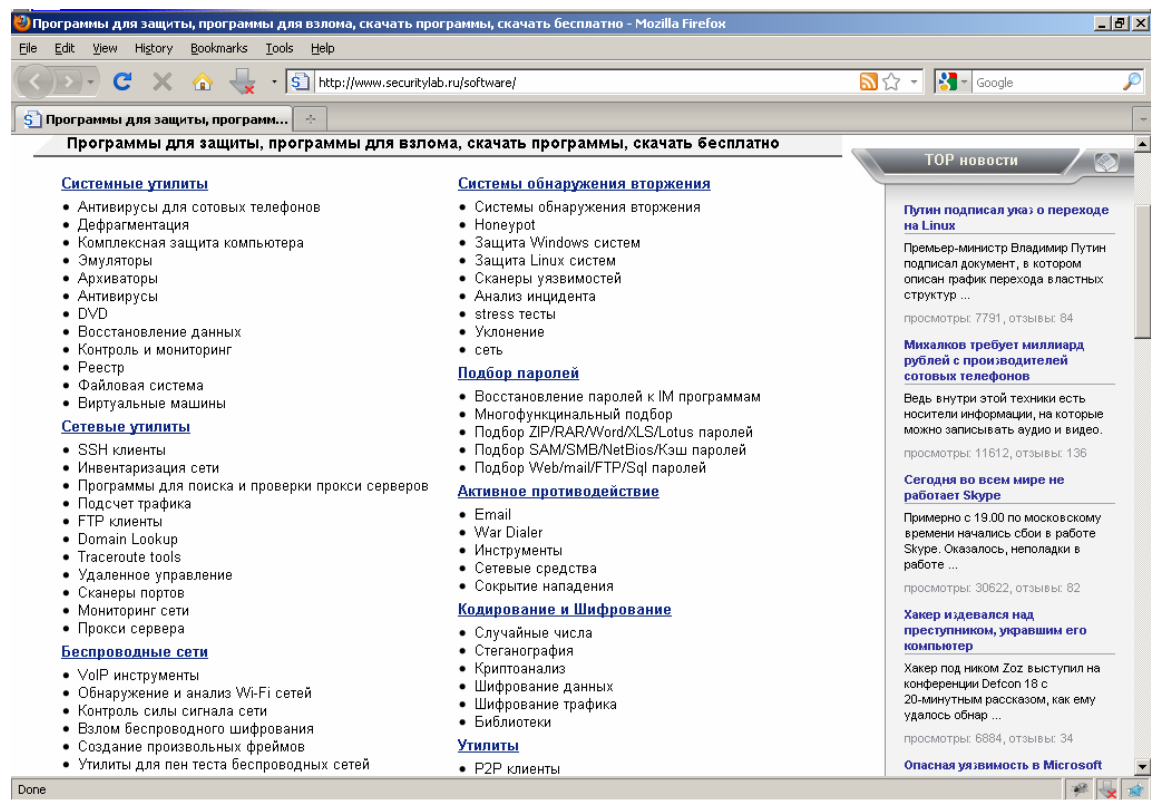


Search:

# Использование готовых инструментальных средств тестирования безопасности

# Где взять...

Одна из лучших подборок **готовых средств** тестирования **безопасности** представлена на сайте **securitylab.ru** в разделе «Софт».



# Что может понадобиться

Ответ на вопрос о том, каким ПО вам придётся воспользоваться, очень сильно зависит от того, **что и как вы собираетесь тестировать**.

В любом случае вам могут пригодиться:

- всевозможные **сетевые утилиты и снифферы** — позволят изучить работу приложения с сетью;
- системы **обнаружения вторжений** (с точки зрения «что должно быть в нашем приложении»;
- ПО для **подбора паролей**;
- **сканеры уязвимостей** (сети, операционной системы и т.п.)

# Что может понадобиться

Помните, что для взлома и защиты от взлома путём тестирования применяются одни и те же средства. В процессе тестирования безопасности вы должны быть «вооружены» и информированы как профессиональный взломщик.

Ваша задача – сломать на стадии тестирования то, что кто-то хотел бы сломать на стадии эксплуатации.

Успешный взлом при тестировании – это неудача злоумышленника в реальной жизни.

- Меня как-то папа попросил сервер генпрокуратуры взломать, дело его посмотреть.
- Получилось ?
- Да. Теперь там и моё дело есть.

(C) [bash.org.ru](http://bash.org.ru)



# Противостояние социальной инженерии



# Определение

**Социальная инженерия** — метод управления действиями человека без использования технических средств. Метод основан на **использовании слабостей человеческого фактора** и считается очень разрушительным.

Злоумышленник получает информацию, например, путём сбора информации о служащих объекта атаки, с помощью обычного телефонного звонка или путем проникновения в организацию под видом её служащего.

Злоумышленник может позвонить работнику компании (под видом технической службы) и выведать пароль, сославшись на необходимость решения небольшой проблемы в компьютерной системе.

Используя реальные имена в разговоре со службой технической поддержки, злоумышленник рассказывает придуманную историю, что не может попасть на важное совещание на сайте со своей учетной записью удаленного доступа.

Другим подспорьем в данном методе являются исследование мусора организаций, виртуальных мусорных корзин, кража портативного компьютера или носителей информации.



# Определение

**Итак, по сути:** с применением методов социальной инженерии злоумышленник может получить то, что не может получить техническими средствами.

Пароль **тяжело взломать**? Но если это – дата рождения, телефон или имя родственника жертвы – его **можно угадать**.

Не угадывается? Можно прислать уведомление о получении открытки, заманить жертву на сайт-ловушку и **обманом заставить там ввести свои учётные данные**.

И т.д. – примеров много.

**Главный вывод:** проблема в недостаточной образованности и недостаточной внимательности людей. Значит – нужно что-то с этим делать.

- Какой самый страшный компьютерный вирус можно подхватить?
- Руки. Они у тебя уже есть. (C) [bash.org.ru](http://bash.org.ru)



# Защита

Для **защиты пользователей** от социальной инженерии можно применять как **технические**, так и антропогенные (**человеческие**) средства.

Работа с людьми реализуется в виде:

- **Привлечении внимания к вопросам безопасности.** Приложение может давать порой очень навязчивые «подсказки», постоянно выводить напоминания в виде «мы никогда ни под каким предлогом не спросим ваш пароль» и т.п.
- **Повышении ответственности пользователей через осознание ими всей серьёзности проблемы.** «Поставили у себя на компьютере запрещённый софт — остались в этом месяце без зарплаты».
- **Постоянном повышении квалификации сотрудников.** Рекомендуется проводить краткие «микротренинги» (на полчаса-час), где рассказывать о существующей опасности и способах защиты.

*Узнай свой IQ! Отправь sms с текстом "Мой IQ" на быстрый номер XXXX и получи результат. (C) bash.org.ru*

# Защита

К **технической защите** можно отнести средства, **мешающие** **заполучить информацию** и средства, **мешающие воспользоваться полученной информацией**.

Так, например (коль скоро большую распространённость получили атаки при помощи электронных писем): помешать злоумышленнику получить запрашиваемую информацию можно, **анализируя** как **текст входящих** писем (предположительно, злоумышленника), так **и исходящих** (предположительно, цели атаки) по ключевым словам.

**CENSORED**

# Защита

Средства, мешающие воспользоваться полученной информацией, можно разделить на те, которые:

- полностью блокируют использование данных, где бы то ни было, кроме рабочего места пользователя (привязка аутентификационных данных к серийным номерам и электронным подписям комплектующих компьютера, ip и физическому адресам);
- делают невозможным автоматическое использование полученных ресурсов (например, авторизация по системе CAPTCHA, когда в качестве пароля нужно выбрать указанное ранее изображение или часть изображения, но в сильно искажённом виде).



# Что должно быть в приложении

Итак, что мы можем реализовать (и проверить!) в нашем приложении.

- все пароли должны быть сложными и НЕ должны содержать в себе в каком бы то ни было виде каких бы то ни было персональных данных человека;
- если в системе используется несколько паролей – они обязательно должны различаться;
- система должна заставлять пользователей менять пароли с некоторой периодичностью;
- система должна максимально информировать пользователя о последствиях выполнения «небезопасных» действий;
- безопасность системы должна быть спроектирована так, чтобы не возникало необходимости спрашивать мнение пользователя, часто технически неграмотного.

# Тест для проверки изученного

1. Какие основные направления исследования учитываются при планировании тестов безопасности?
2. Что понимается под «интерфейсами приложения» в контексте тестирования безопасности?
3. Как приложение может противостоять атакам через среду исполнения?
4. Какие наиболее показательные тесты безопасности на примере функций «Авторизация пользователя» и «Поиск по сайту» вы запомнили?
5. Какие инструментальные средства могут пригодиться для тестирования безопасности (в т.ч. автоматизированного)?
6. Что такое «социальная инженерия» (в контексте безопасности веб-ориентированных приложений)?
7. Какие примеры атак с использованием социальной инженерии вы знаете?
8. Какие антропогенные способы защиты от атак с использованием социальной инженерии вы знаете?
9. Какие средства противостояния социальной инженерии относятся к техническим?
10. Что можно порекомендовать в ситуации, когда пользователи не могут запомнить сложные пароли и стремятся их где-то записать?