база данных под прицелом

крис касперски ака мыщъх no-email

данные это основа всего. это и номера кредитных карт, и личная информация пользователей, и сведениях об угнанных машинах. содержимое чатов и форумов тоже хранится в БД. проникновение в корпоративную (военную, правительственную) базу данных – самое худшее, что только может случиться с компанией. поразительно, но даже критические сервера зачастую оказываются никак не защищены и взламываются даже 12-летними любителями командной строки без особых усилий.

введение

Сервера баз данных относятся к наиболее критичным информационным ресурсам и потому они должны размещаться на выделенном сервере, расположенным во внутренней корпоративной сети, огражденной маршрутизатором или брандмауэром. Взаимодействие с базами данных обычно осуществляется через WEB-сервер, находящийся внутри DMZ-зоны (см. статью о брандмауэрах).

Размещать сервер базы данных на одном узле с WEB-сервером категорически недопустимо не только по техническим, но и юридическим соображениям (законодательства многих стран диктуют свою политику обращения с конфиденциальными данными, особенно если эти данные хранят информацию о клиентах компании). Тем не менее, совмещение сервера БД с WEB-сервером – обычное дело, которым сегодня никого не удивишь. Экономия… мать ее так! Захватив управление WEB-сервером (а практически ни одному WEB-серверу не удалось избежать ошибок переполнения буфера и прочих дыр), атакующий получит доступ ко всем данным, хранящимся в базе!

Сервер БД, как и любой другой сервер, подвержен ошибкам проектирования среди которых доминируют переполняющиеся буфера, многие из которых позволяют атакующему захватывать управление удаленной машиной с наследованием администраторских привилегий. Яркий пример тому – уязвимость, обнаруженная в сервере MS SQL и ставшая причиной крупной вирусной эпидемии. Не избежал этой участи и MySQL. Версия 3.23.31 падала на запросах типа `select a.AAAAAAA…AAAAAA.b`, а на соответствующим образом подготовленных строках – передавала управление на shell-код, причем атаку можно было осуществить и через браузер, передав в URL что-то типа: `script.php?index=a.(shell-code).b`.

Однако, даже защищенный брандмауэром SQL-сервер, может быть атакован через уязвимый скрипт или нестойкий механизм аутентификации. Разумеется, мы не можем рассказать обо всех существующих атаках, но продемонстрировать пару-тройку излюбленных хакерских приемов – вполне в наших силах.

нестойкость шифрования паролей

Пароли, регламентирующие доступ к базе данных, ни при каких обстоятельствах не должны передаваться открытым текстом по сети. Вместо пароля, передается его хэш, зашифрованный случайно сгенерированной последовательностью байт, и называемый проверочной строкой (check-string). Короче говоря, реализуется классическая схема аутентификации, устойчивая к перехвату информации и при этом не допускающая ни подбора пароля, ни его декодирования (во всяком случае в теории).

На практике же во многих серверах БД обнаруживаются жестокие ошибки проектирования. Взять хотя бы MySQL версии 3.x. Хэш-функция, используемая для "сворачивания" пароля возвращает 64-разрядную кодированную последовательность, в то время как длина случайно генерируемой строки (random string) составляет всего лишь 40 бит. Как следствие, шифрование не полностью удаляет всю избыточную информацию и анализ большого количества перехваченных check-string/random-string позволяет восстановить исходный хэш (пароль восстанавливать не требуется, т. к. для аутентификации он на фиг не нужен).

В несколько упрощенном виде процедура шифрования выглядит так:

```
// P1/P2  – 4 левых/правый байт парольного хэша соответственно
// C1/C2 – 4 левых/правый байт random-string соответственно
```

```
seed1 = P1 ^ C1;
seed2 = P2 ^ C2 ;
for(i = 1; i <= 8; i++)
{
        seed1 = seed1 + (3*seed2);
        seed2 = seed1 + seed2 + 33;
        r[i] = floor((seed1/n)*31) + 64;
}

seed1 = seed1+(3*seed2);
seed2 = seed1+seed2+33;
r[9] = floor((seed1/n)*31);

checksum =(r[1]^r[9] || r[2]^r[9] || r[7]^r[9] || r[8]^r[9]);
```

Листинг 1 шифрование парольного хеша случайной строкой

Нестойкие механизмы аутентификации встречались и в других серверах, однако, к настоящему моменту практически все они давно ликвидированы.

перехват пароля

Для авторизации на сайте в подавляющем большинстве случаев используются нестойкие механизмы аутентификации, разработанные непосредственно самим WEB-мастером, и передающие пароль в открытом виде. Как следствие – он может быть легко перехвачен злоумышленником, забросившим на одну из машин внутренней сети и/или DMZ-зоны сниффер, или создавшим точную копию атакуемого WEB-сервера, для заманивания доверчивых пользователей – тогда логин и пароль они введут сами.

Многие сервера хранят информацию об авторизации в куках (cookie), находящихся на машинах удаленных пользователей, и вместо того, чтобы ломиться на хорошо защищенный корпоративный сервер, хакер может атаковать никем не охраняемые клиентские узлы. Главная трудность заключается в том, что их сетевые координаты наперед неизвестны и атакующему приходится тыкаться вслепую, действуя наугад. Обычно эта проблема решается массированной рассылкой почтовой корреспонденции с троянизированным вложением внутри по многим адресам – если нам повезет, то среди пользователей, доверчиво запустивших трояна, окажется хотя бы один корпоративный клиент. Ну а извлечь кук – уже дело техники.

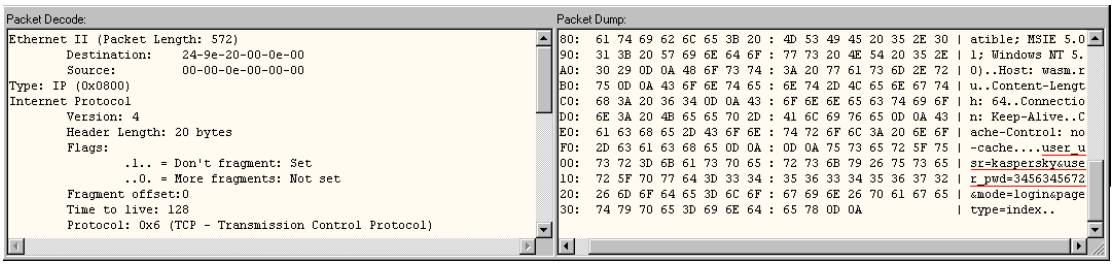

Рисунок 1 пароль к БД, выловленный сниффером

Некоторые сервера баз данных (и в частности ранние версии MS SQL), автоматически устанавливают пароль по умолчанию, предоставляющий полный доступ к базе и позволяющий делать с ней все, что угодно (у MS SQL'я этот пароль "sa").

>>>> врезка вскрытие скрипта

Нормально работающий WEB-сервер никогда не выдает исходный код скрипта, а только результат его работы. Между тем, вездесущие ошибки реализации приводят к тому, что код скрипта в некоторых случаях все-таки становится доступным, причем виновником может быть как сервер, так и обрабатываемый им скрипт. Естественно, в скриптах ошибки встречаются намного чаще, поскольку их пишут все кому не лень, порой не имея никакого представления о безопасности. Сервера же проходят более или менее тщательное тестирование и основные дыры обнаруживаются еще на стадии бета-тестирования.

Подробнее об этом можно прочитать в моей статье "Безопасное программирование на языке Perl" (kpnc.opennet.ru/safe.perl.zip), здесь же мы сосредоточимся непосредственно на взломе самой БД. Исследуя тело скрипта, можно нарыть немало интересного. Например, имена полей, названия таблиц, мастер-пароли, хранящиеся открытым текстом и т. д.

```
…
if ($filename eq "passwd")     #проверка имени на корректность
…
```

Листинг 2 мастер-пароль к БД, хранящийся в теле скрпита открытым текстом

навязывание запроса или SQL-injecting

Типичный сценарий взаимодействия с базой данных выглядит так: пользователь вводит некоторую информацию в поля запроса. Оттуда ее извлекает специальный скрипт и преобразует в строку запроса к базе данных, передавая серверу ее на выполнение:

```
$result = mysql_db_query("database", "select * from userTable
                        where login = '$userLogin' and password = '$userPassword' ");
```

Листинг 3 типичная схема формирования запроса к БД

Здесь: `$userlogin` – переменная, содержащая имя пользователя, а `$userPassword` – его пароль. Обратите внимание, что обе переменные размещены внутри текстовой строки, окаймленной кавычками. Это необычно для Си, но типично для интерпретируемых языков наподобие Perl'a и PHP. Подобный механизм называется интерполяцией строк и позволяет автоматически подставлять вместо переменной ее фактическое значение.

Допустим, пользователь введет KPNC/passwd, тогда строка запроса будет выглядеть так: "`select * from userTable where login = '`**`KPNC`**`' and password = '`**`passwd`**`'`". (пользовательский ввод выделен полужирным шрифтом). Если такой логиг/пароль действительно присутствуют в базе, функция сообщает идентификатор результата, в противном случае возвращается FALSE.

Хотите войти в систему под именем другого пользователя, зная его логин, но не зная пароль? Воспользуется тем, что механизм интерполяции позволяет атакующему воздействовать на строку запроса, видоизменяя ее по своему усмотрению. Посмотрим, что произойдет, если вместо пароля ввести последовательность "`fuck' or '1'= '1`" (естественно без кавычек): "`select * from userTable where login = '`**`KPNC`**`' and password = '`**`fuck' or '1' = '1`**`'`". Смотрите, кавычка, стоящая после fuck'a, замкнула пользовательский пароль, а весь последующий ввод попал в логическое выражение, навязанное базе данных атакующим. Поскольку один всегда равен одному, запрос будет считаться выполненным при любом введенном пароле и SQL-сервер возвратит все-все-все записи из таблицы (в том числе и не относящиеся к логину KPNC)!

Рассмотрим другой пример: "`SELECT * FROM userTable WHERE msg='$msg' AND ID=669`". Здесь: msg – номер сообщения, извлекаемого из базы, а ID – идентификатор пользователя, автоматически подставляемый скриптом в строку запроса и непосредственно не связанный с пользовательским вводом (константная переменная использована по соображениям наглядности, в конечном скрипте будет скорее всего использована конструкция типа: ID='$userID'). Чтобы получить доступ к остальным полям базы (а не только тем, чей ID равен 669) необходимо отсечь последнее логическое условие. Это можно сделать внедрив в строку пользовательского ввода символов комментария ("--" и "/*" для MS SQL и MySQL соответственно). Текст, расположенный левее символов комментария, игнорируется. Если вместо номера сообщения ввести "1' AND ID=666 --", строка запроса примет следующий вид: "`SELECT * FROM userTable WHERE msg='1' and ID= 666 --' AND ID=669`" (блеклым цветом выделен текст комментария). Как следствие, атакующий получит возможность самостоятельно формировать ID, читая сообщения, предназначенные совсем для других пользователей.

Причем, одним лишь видоизменением полей SELECT'a дело не огранивается и существует угроза прорыва за его пределы. Некоторые SQL-сервера поддерживают возможность задания нескольких команд в одной строке, разделяя их из знаком ";", что позволяет атакующему выполнить любые SQL-команды, какие ему только заблагорассудится. Например, последовательность "`'; DROP TABLE 'userTable' --`", введенная в качестве имени пользователя или пароля, удаляет всю userTable на хрен!

Еще атакующий может сохранять часть таблицы в файл, подсовывая базе данных запрос типа "`SELECT * FROM userTable INTO OUTFILE 'FileName'`". Соответствующий ему URL уязвимого скрипта может выглядеть например так:

`www.victim.com/admin.php?op=login&pwd=123&aid=Admin'%20INTO%20OUTFILE%20'/path_to_file/pwd.txt`, где path_to_file – путь к файлу pwd.txt, в который будет записан администрский пароль. Удобное средство для похищения данных, не так ли? Главное – размесить файл в таком месте откуда его потом будет можно беспрепятственно утянуть, например, в одном из публичных WWW-каталогах. Тогда полный путь к файлу должен выглядеть приблизительно так: "`../../../../WWW/myfile.txt`". (точная форма запроса зависит от конфигурации сервера). Но это еще только цветочки! Возможность создания файлов на сервере, позволяет засылать на атакуемую машину собственные скрипты (например, скрипт, дающий удаленный shell – "`<? passthru($cmd) ?>`"). Естественно, максимальный размер скрипта ограничен предельно допустимой длиной формы пользовательского ввода, но это ограничение зачастую удается обойти ручным формированием запроса в URL, или использованием SQL-команды INSERT INTO, добавляющей новые записи в таблицу.

Скорректированный URL-запрос может выглядеть например так: `http://www.victim.com/index.php?id=12'` или так: `http://www.victim.com/index.php?id=12+union+select+null,null,null+from+table1 /*`. Последний запрос работает только на MySQL версии 4.x и выше, поддерживающий union (объединение нескольких запросов в одной строке). Здесь table1 – имя таблицы, содержимое которой необходимо вывести на экран.

Атаки подобного типа называются ***SQL-инъекциями*** (***SQL-injection***) и являются частным случаем более общих атак, основанных на ошибках фильтрации и интерполяции строк. Мы словно "впрыскиваем" в форму запроса к базе данных собственную команду, прокалывая хакерской иглой тело уязвимого скрипта (отсюда и "инъекции"). Это не ошибка SQL-сервера (как часто принято считать). Это – ошибка разработчиков скрипта. Грамотно спроектированный скрпит должен проверять пользовательский ввод на предмет присутствия потенциально опасных символов (как-то: одиночная кавычка, точка с запятой, двойное тире, а для MySQL еще и символ звездочки), включая и их шестнадцатеричные эквиваленты, задаваемые через префикс "%", а именно: %27, %2A и %3B. (Код символа двойного тире фильтровать не нужно, т. к. он не входит в число метасимволов, поддерживаемых браузером). Если хотя бы одно из условий фильтрации не проверяется или проверяется не везде (например, остаются не отфильтрованными строки URL или cookie) в скрипте образуется дыра, через которую его можно атаковать.

Впрочем, сделать это будет не так уж и просто. Необходимо иметь опыт программирования на Perl/PHP и знать как может выглядеть та или иная форма запроса и как чаще всего именуются поля таблицы, в противном случае интерполяция ни к чему не приведет. Непосредственной возможности определения имен полей и таблиц у хакера нет и ему приходится действовать методом слепого перебора (blinding).

К счастью для атакующего, большинство администраторов и WEB-мастеров слишком ленивы, чтобы разрабатывать все необходимые им скрипты самостоятельно и чаще они используют готовые решения, исходные тексты которых свободно доступны в сети. Причем, большинство этих скриптов дырявы как ведро без дна. Взять к примеру тот же PHP Nuke, в котором постоянно обнаруживаются все новые и новые уязвимости.

Приблизительная стратегия поиска дыр выглядит так. Скачиваем исходные тексты PHP Nuk'a (или любой другой портальной системы), устанавливаем их на свой локальный компьютер, проходимся глобальным поиском по всем файлам, откладывая в сторонку все те, что обращаются к базе данных (вызов типа **mysql_query**/**mysql_db_query** или типа того). Прокручиваем курсор верх и смотрим – где-то поблизости должна быть расположена строка запроса к базе (например: `$query = "SELECT user_email, user_id FROM ${prefix}_users WHERE user_id = '$cookie[0]'"`). Определяем имена переменных, подставляемых в базу, находим код, ответственный за передачу параметров пользовательского ввода и анализируем условия фильтрации.

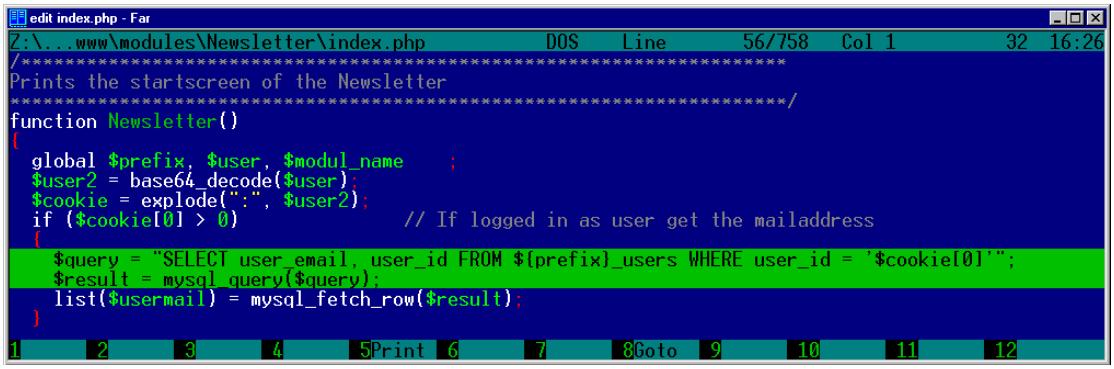

Рисунок 2 фрагмент PHP Nuke, ответственный за формирование запроса к базе

В качестве наглядного примера рассмотрим одну из уязвимостей PHP Nuke 7.3, связанную с обработкой новостей. Соответствующий ей URL выглядит так: `modules.php?name=News&file=categories&op=newindex&catid=1`. По его внешнему виду можно предположить, что значение `catid` передается непосредственно в строке запроса к БД и если разработчик скрипта забыл о фильтрации, у нас появляется возможность модифицировать запрос по своему усмотрению. Для проверки этого предположения заменим catid с 1 на, допустим, 669. Сервер немедленного отобразит в ответ пустой экран. Теперь добавим к нашему URL следующую конструкцию `"'or'1'1='1"` (полностью он будет выглядеть так: `modules.php?name=News&file=categories&op=newindex&catid=`**`669'or'1'='1`**). Сервер послушно отобразит все новостные сообщения раздела, подтверждая, что SQL-инъекция сработала!

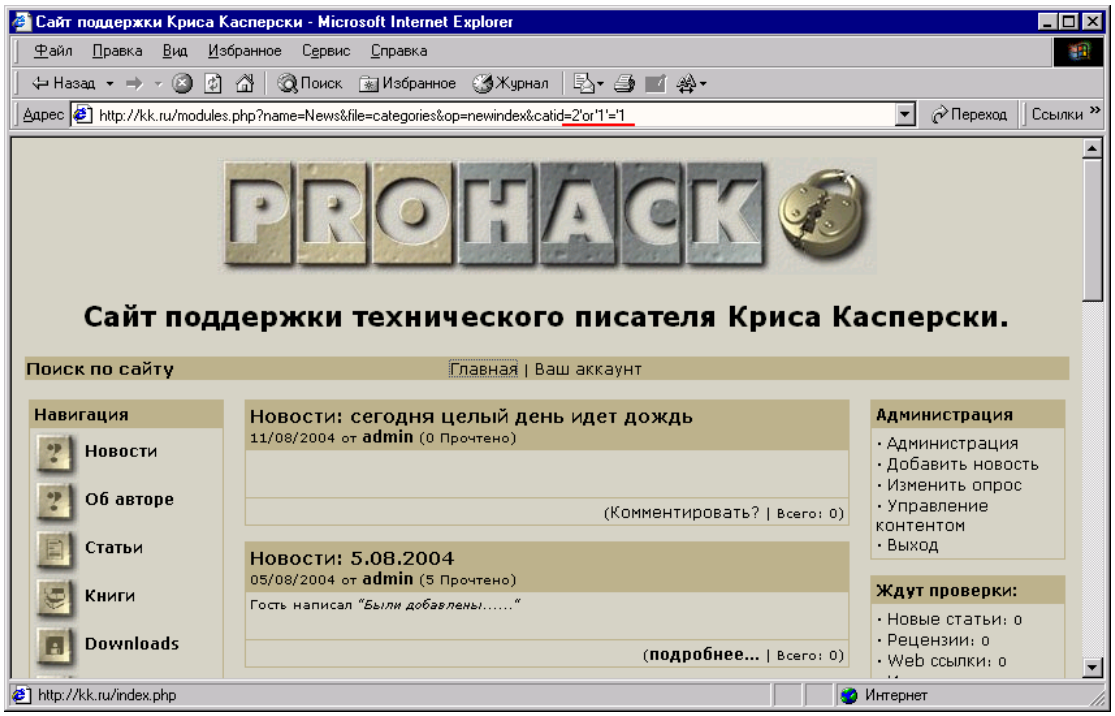

Рисунок 3 SQL-инъекции через строку URL

Еще можно попытаться вызвать ошибку SQL, подсунув ей заведомо неправильный запрос (например, символ одиночной кавычки) и тогда она может сообщить много интересного. Отсутствие ошибок еще не означает, что скрипт фильтрует пользовательский ввод – быть может он просто перехватывает сообщения об ошибках, что является нормальной практикой сетевого программирования. Также возможна ситуация, когда при возникновении ошибки возвращается код ответа 500, или происходит переадресация на главную страницу. Подобная двусмысленность ситуации существенно затрудняет поиск уязвимых серверов, но отнюдь не делает его невозможным!

Анализ показывает, что ошибки фильтрации встречаются в большом количестве скриптов (включая коммерческие), зачастую оставаясь неисправленными годами. Естественно, дыры в основных полях ввода давным-давно заткнуты и потому рассчитывать на быстрый успех уже не приходится. Запросы, передаваемые методом POST, протестированы значительно хуже, поскольку, передаются скрыто от пользователя и не могут быть модифицированы непосредственно из браузера, отсекая армаду начинающих "хакеров". Между тем взаимодействовать с WEB-сервером можно и посредством netcat'a (telnet'a), формируя POST-запросы вручную.

команда	назначение
CREATE TABLE	создание новой таблицы
DROP TABLE	удаление существующей таблицы
INSERT INTO	добавление в таблицу поля с заданным значением
DELETE FROM …WHERE	удаление из таблицы всех записей, отвечающих условию WHERE
SELECT * FROM … WHERE	выборка из базы всех записей, отвечающих условию WHERE
UPDATE ... SET ... WHERE	обновление всех полей базы, отвечающих условию WHERE

Таблица 1 основные команды SQL

>>>> определить наличие SQL

Прежде чем начинать атаку на SQL-сервер, неплохо бы определить его присутствие, а в идеале – еще и распознать тип. Если сервер расположен внутри DMZ (где ему находиться ни в коем случае нельзя), то атакующему достаточно просканировать порты (см. таблицу 2).

порт	сервер
1433	Microsoft-SQL-Server
1434	Microsoft-SQL-Monitor
1498	Watcom-SQL
1525	ORACLE
1527	ORACLE
1571	Oracle Remote Data Base
3306	MySQL

Таблица 2 порты, прослушиваемые различными серверами БД

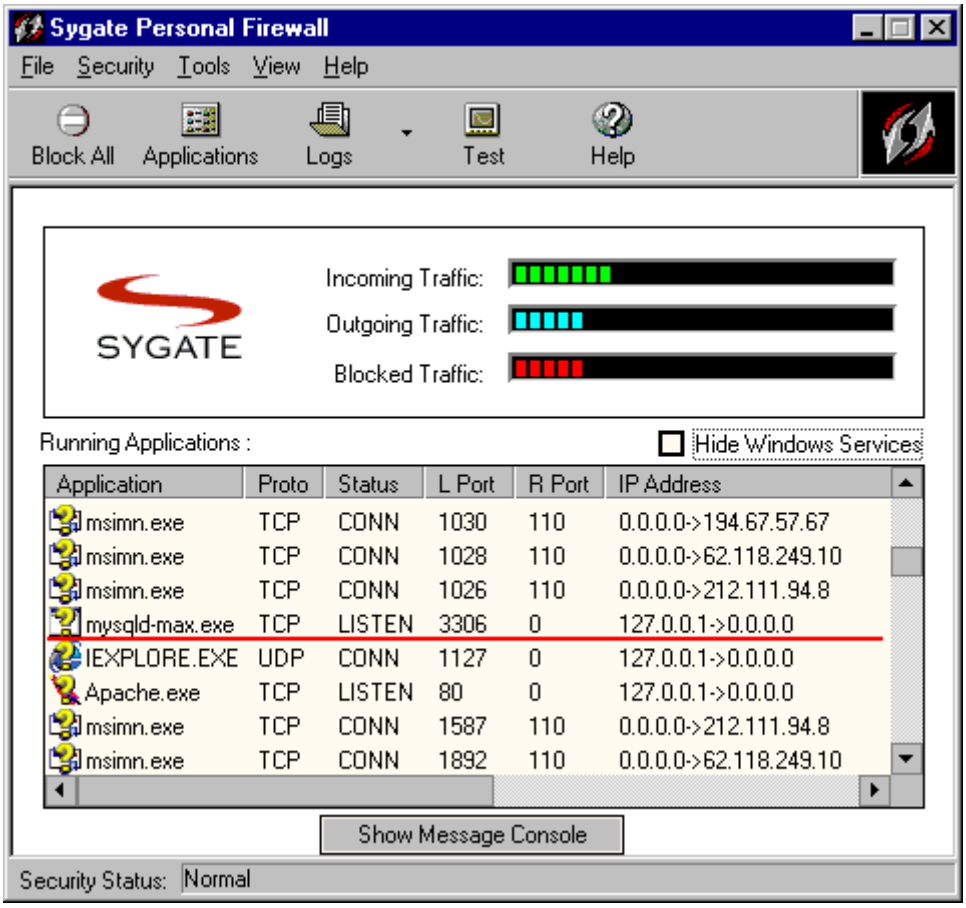

Рисунок 4 сервер MySQL, прослушивающий 3306-порт

>>>> врезка противодействие вторжению

Когда ручной поиск дыр надоедает, хакеры, в сердцах обложив всех WEB-программистов смачным матом, запускают свое любое средства автоматического поиска уязвимостей и идут на перекур.

Одним из таких средств является Security Scanner, разработанный компанией Application Security и официально предназначенный для тестирования MySQL на стойкость к взлому. Ну, хакерам официоз не грозит. Как и всякое оружие, Security Scanner может использоваться и во вред, и во благо.

Он позволяет искать дыры как в самом сервере БД, так и в web-скриптах. При этом, БД проверятся на предмет уязвимости к атакам типа Denial of Service, наличия слабых паролей, неверно сконфигурированных прав доступа и т.д. В скриптах сканер позволяет обнаружить ошибки фильтрации ввода, позволяющие осуществлять SQL-инъекции, что значительно упрощает атаку.

заключение

SQL-инъекции в очередной раз продемонстрировали миру, что программ без ошибок не бывает. Однако, не стоит переоценивать их значимость. Мавр сделал свое дело и может уходить. Администраторы и девелоперы об опасности и количество уязвимых сайтов неуклонно тает с каждым днем. Реальную власть нас системой дают лишь принципиально новые методики атак, неизвестные широкой общественности. Найти их наша с тобой задача. Освободи свой разум, перешагни грань неведомого, и зайди на сервер с той стороны, с которой на него еще никто не заходил.