Крис Касперски

БЕЗОПАСНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PERL

kk@sendmail.ru

Крис Касперски
kk@sendmail.ru

Безопасное программирование на языке Perl

Статья посвящена проблемам безопасности cgi-приложений и рассказывает о типовых ошибках разработчиков, предлагая возможные пути их устранения. Подаваемый материал большей частью ориентирован на Perl, но во многом применим и к другим языкам.

Статья рассчитана на WEB-мастеров, программистов средней квалификации и новичков, имеющих общее представление о языке Perl и устройстве операционной системы.

Введение

Для организации интерактивного взаимодействия с пользователем возможностей "голого" HTML оказывается недостаточно и приходится прибегать к вызову внешних программ, взаимодействующих с WEB-сервером через CGI-интерфейс. Такие программы могут быть написаны на любом языке, хоть на Бейсике, но исторически сложилось так, что в подавляющем большинстве случаев для их разработки используется Perl.

Это лаконичный, немногословный, мощный и в то же время легкий в освоении язык, реализованный практически на всех существующих платформах и операционных системах. Он выгодно отличается от Си тем, что не имеет характерных для последнего проблем переполнения буферов, но, несмотря на все свои достоинства, с точки зрения безопасности Perl – едва ли не самый худший выбор.

Изначально Perl предназначался для разработки средств управления и мониторинга многоуровневых сетей – ***локальных*** приложений, нетребовательных к защищенности. Удобству эксплуатации отдавалось предпочтение в ущерб безопасности, например, вызов *open* может не только открывать файл, но и ***запускать*** его, если в имени присутствует символ конвейера. Излишняя гибкость и самостоятельность языка создают проблемы при написании ***серверных*** приложений, обязанных не допустить выполнение любых действий, явно не санкционированных разработчиком.

Никогда нельзя быть уверенным, что все клиенты будут использовать программу "***как нужно***", а не "***как можно***". Помимо непреднамеренных ошибок пользователей большую угрозу представляют злоумышленники, пытающиеся найти такие запросы, обработка которых приносила бы тот или иной ущерб серверу.

Большинство скриптов разрабатываются непрофессионалами, порой только-только осваивающих Perl в процессе программирования. Неудивительно, что при этом практически всегда допускаются ошибки, приводящие к уязвимости скрипта. На сегодняшний день качественный скрипт скорее исключение, чем правило. Ошибки нередко обнаруживаются и в профессиональных (вернее, претендующих на это звание) продуктах. Все скрипты Мэта Райта, приведенные в его книге "CGI/Perl", и получившие в результате этого большую распространенность, некорректно фильтруют ввод пользователя, допуская тем самым возможность атаки на сервер.

Создание безопасных серверных приложений представляет серьезную проблему, решению которой и посвящена настоящая статья.

Источник угрозы

Можно выделить три основных угрозы, связанных с использованием любых cgi-скриптов, не обязательно написанных именно на Perl:

- *несанкционированное получение прав пользователя (суперпользователя) на удаленной машине*;
- *несанкционированная модификация динамически генерируемой страницы включением собственных тегов*;
- *перегрузка сервера интенсивной работой cgi-скриптов, вплоть до ее полной неработоспособности.*

Большинство WEB-серверов, работающих под управлением UNIX-подобных операционных систем, обладают привилегиями суперпользователя. В противном случае сервер не сможет открыть необходимый ему восьмидесятый порт. Некоторые WEB-сервера обслуживают клиентов через нестандартный 8000 или 8080 порт и, в принципе, могут довольствоваться правами непривилегированного пользователя. Однако, такая схема ввиду связанных с ней неудобств не получила массового распространения.

ОС семейства Windows не накладывают никаких ограничений на обработку входящих TCP-соединений и позволяют открывать восьмидесятый порт даже прикладному коду, запущенному с гостевыми правами. Тем не менее, распространенные WEB-серверы под Windows NT\2000 требуют для своей работы наивысших привилегий.

По соображениям безопасности многие (но не все!) серверы допускают возможность понижения привилегий потока, обслуживающего web-клиента, до обычного непривилегированного пользователя (как правило *nobody* или *www*), а программа *cgiwarp* (автор Nathan Neulinger) позволяет запускать cgi-скрипты с правами их владельца (по умолчанию скрипты исполняются от имени и с привилегиями WEB-сервера).

Ошибка в скрипте может привести к несанкционированному получению прав пользователя на удаленной машине, а в некоторых (между прочим, достаточно частых) случаях и привилегий администратора!

Обладая правами пользователя, злоумышленник может модифицировать WEB-страницы по своему усмотрению, получать доступ к секретной информации (например, номерам кредитных карт покупателей, паролям посетителей сайта, файлам протоколов и т. д.), использовать сервер в качестве полигона для атаки на другой узел и делать еще много других нехороших дел. А с привилегиями администратора он и вовсе может сотворить с системой все, что ему заблагорассудится!

Вторая категория ошибок связна с динамической генерацией интерактивных WEB-страниц. С свою очереди их можно разделить на две подкатегории: ***страницы, доступные только тому пользователю для которого была сгенерирована страница*** и ***страницы, доступные множеству пользователей***. Примером первых может служить результат работы поисковых машин, а вторых – чаты и гостевые книги.

В большинстве случаев страница, динамически сгенерированная в ответ на запрос пользователя, содержит и сам запрос. А это открывает возможность для использования HTML-тегов, наибольшую опасность из которых представляют директивы SSI (Server Side Include), поскольку они позволяют включать в страницу содержимое другого файла, значение переменной окружения, результат работы вызванной программы и т. д.

Страницы, доступные всем пользователям, помимо вышеупомянутой, подвержены и другой угрозе – созданию фальшивых полей ввода (как правило требующих указания своего пароля или номера кредитной карты), передающих свое содержимое злоумышленнику. Так же злоумышленник может разместить вредоносный Java (Perl и VisualBasic) скрипт, зацикливающий браузер посетителя или открывающий у него множество окон размером миллион на миллион пикселей. Еще можно рассмотреть возможность автоматического перенаправления клиентов на свою станицу или сайт баннерного спонсора, похищение у них локальных файлов через "дыры" в Internet Explorer и Netscape Навигатор и т. д.

Наконец, любой, даже в высшей степени корректно написанный скрипт, требует для своей работы весьма значительных процессорных ресурсов. Шквал запросов, инициируемый злоумышленником, представляет собой эффективную атаку "отказа в обслуживании", особенно если скрипт интенсивно работает с файловой системой, да еще в добавок неэкономно расходует память. Увеличить выносливость сервера можно переходом на компилируемые языки, установкой более быстрого процессора и отказом от использования скриптов везде, где это возможно.

Таким образом, размещение скрипта на сервере всегда таит в себе некоторую опасность и создает угрозу для его благополучия.

Проблемы администрирования

Чем рискует администратор, разрешая клиентам выполнение собственных cgi-скриптов? Если WEB-сервер исполняет их не с правами root, то ничем, за исключением предоставления прав пользователя владельцем скриптов (и потенциального предоставления этого права злоумышленникам, обнаружившим уязвимость скрипта, если, конечно, она есть).

Пользователь же даже с хорошо защищенной системой может сотворить много нехорошего. В лучшем случае использовать ее для массовой рассылки корреспонденции (в просторечии спама) или для атак на другие узлы. Особенно это актуально для служб бесплатного хостинга, не имеющих возможности проверить подлинность данных, сообщенных клиентом при регистрации. Это позволяет злоумышленнику не только оставаться полностью анонимным, но и противостоять закрытию его аккаунта, т. к. ничего не стоит повторно зарегистрироваться под другим именем.

Для предотвращения подобных атак администраторам бесплатных ресурсов настоятельно рекомендуется установить межсетевые экраны, запрещающие установку исходящих соединений. Однако, такое решение ограничит и легальных клиентов, вызывая их отток к провайдеру, не оснащенному подобной защитой.

В UNIX-системах любой, даже непривилегированный, пользователь имеет доступ ко многим секретным файлам сервера, манипуляции с которыми потенциально способны нанести ущерб системе. Например, файл "/etc/passwd" доступен для чтения всем пользователям этого сервера, и, если администратор забыл "затенить" пароли, злоумышленник сможет в относительно короткое время их подобрать. ОС Windows NT\2000 в этом отношении защищена значительно лучше, но все же имеет ряд слабых мест, например, допускает просмотр профилей безопасности до ввода пароля (*профили же среди прочей информации хранят историю паролей, с целью предотвращения их повторного использования; да, это старые, уже недействительные пароли, но они раскрывают стратегию выбора пароля – используются ли случайные комбинации символов, словарные слова, клички любимых хомячков, чем невероятно облегчают проникновение в систему*).

Проблем при предоставлении WEB – хостинга без права выполнения собственных скриптов намного меньше: в этом случае сервер обслуживает пользователей самостоятельно, а операционная система о их существовании даже и не подозревает. Чтобы отличить "нормальных" пользователей от клиентов, обслуживаемых сервером, последних часто называют ***псевдопользователями***.

Псевдопользователи не имеют никаких прав и полномочий доступа к файлам, кроме явно разрешенных сервером. Они могут исполнять уже находящиеся на сервере скрипты, но создавать свои не в состоянии. Большинство администраторов предоставляет в распоряжение псевдопользователей некоторое количество типовых скриптов, как то счетчики, гостевые книги, доски объявлений и т. д.

С точки зрения безопасности это довольно рискованный ход – ошибка скрипта может запросто дать WEB-клиенту права непривилегированного пользователя, а то и самого администратора, со всеми вытекающими отсюда последствиями.

Практика показывает – ошибки в широко распространенных скриптах скорее закономерность, чем непредвиденная случайность. Любой скрипт, даже полученный от профессиональных программистов, а уж тем более созданный любителем и свободно распространяемый по сети (а именно такие и пользуются наибольшей популярностью), должен быть тщательнейшим образом проверен перед его помещением на сервер.

Эта статья в равной мере ориентирована как на самих разработчиков, так и на основных потребителей их продукции – администраторов и WEB-мастеров.

Типовые ошибки и способы их устранения

Несмотря на свое многообразие ошибки разработчиков легко разделить на четные категории:

- *передача ввода пользователя внешним программам или штатным функциям без надлежащей фильтрации;*
- *оформление компонентов приложения в виде самостоятельного скрипта, допускающего непосредственный вызов удаленным пользователем;*
- *помещение секретных данных в файл, доступный псевдопользователям;*
- *использование переменных окружения для ответственных целей.*

Все эти пункты подробно рассмотрены ниже.

Отсутствие фильтрации. Ошибки фильтрации пользовательского ввода (или полное отсутствие таковой) наиболее распространены и особенно характерны для Perl, штатные функции которого слишком вольно интерпретируют переданные им аргументы и допускают множество умолчаний. К примеру, программист хочет открыть и вывести на экран запрошенный пользователем файл и создает код наподобие следующего:

```
open(f,$filename);
while(<f>)
{
        print;
}
```

Допущенные ошибки очевидны: во-первых, злоумышленник может получить содержимое любого файла системы, доступного скрипту, передав запрос наподобие "/etc/passwd", а, во-вторых, указав в имени символ конвейера ("|") он сможет запустить любое доступное скрипту приложение и увидеть в браузере результат его работы (например, "echo "+ +" >/.rhosts" позволит подключиться по протоколу rlogin без ввода пароля).

Но не всякая уязвимость настолько очевидна! Попробуйте найти ошибку в следующей реализации того же примера, усиленного принудительным добавлением расширения ".html" к имени открываемого файла:

```
open(f,$filename.".html");
while(<f>)
{
        print;
}
```

На первый взгляд, злоумышленник не сможет ни открыть, ни запустить никакие другие файлы, кроме HTML. Но это не так! Дело в том, что Perl не трактует символ нуля как конец строки и обрабатывает его точно так, как и все остальные символы. В то же время, компоненты, написанные на Си, интерпретируют ноль как завершитель строки! Таким образом, передав строку, содержащую на конце "\0" (например, "/etc/passwsd\0") злоумышленник обойдет защиту скрипта! Помимо этого, функция *open* допускает возможность одновременного запуска множества файлов – передача строки "|calc.exe|sol.exe|freecell.exe|" приведет к запуску приложений "Калькулятор", "Пасьянс Косынка" и "Пасьянс Свободная ячейка", независимо от того будет ли добавлено в конце расширение ".html" или нет.

Даже "*open(f, "/home/www/pages/".$filename."html")*" не уберегает от использования нескольких символов конвейера, и тем более не предотвращает обращения к вышележащим каталогам, хотя на первый взгляд такая защита может показаться совершенно неприступной.

Решение проблемы заключается в ***фильтрации данных*** – удалении из ввода пользователя всех потенциально опасных символов или выдачи сообщения об ошибке при их обнаружении. Таких символов очень много и все они (что очень неприятно) специфичны для каждой функции. Например, у *open* опасны следующие символы и их комбинации:

- *">","">>" и "+>" – открытие файла для записи, дозаписи и перезаписи соответственно;*
- "+<" – *открытие файла для записи и чтения;*
- "|" *и* "`" – *запуск программы;*
- "-" – *чтение со стандартного ввода;*
- "&" – *обращение к файловому манипулятору;*
- ".." *и "/" – обращение к вышележащим каталогам;*
- "\0" – *задание конца строки*.

О возможности обращения к файлу по его манипулятору следует сказать особо. Пусть существует некоторый секретный файл (например, файл паролей или номеров кредитных карт), который открывается в начале работы программы, а затем на экран выводится содержимое файла, запрошенного пользователем, до закрытия секретного файла. Если злоумышленнику доступен исходный тест скрипта или хотя бы приблизительно известны манеры его разработчика, он сможет прочитать секретный файл с помощью самой программы, передав вместо имени его манипулятор! Для чего достаточно воспользоваться клонированием "x&filehandle" или созданием псевдонимов "x&=filehandle", где "x" обозначает режим доступа – "<" для чтения и ">" для записи. Следующий пример как раз и демонстрирует эту уязвимость.

```
open (psw,"passwd") || die;      #открытие файла паролей
#...некоторый код...
print "введите имя файла:"       #запрос имени отображаемого файла
$filename=<>; chop $filename;
if ($filename eq "passwd")       #проверка имени на корректность
         {print "Hello,Hacker!\n";die;}
open(f,$filename) || die;        #вывод файла на экран
while(<f>)
{
         print;
}
```

Если злоумышленник введет "<&=psw" или "<&psw", то на экран выдастся содержимое файла паролей!

Аналогичным путем можно ознакомится и содержимым лексемы DATA, доступной через одноименный манипулятор и очень часто содержащее информацию, не для посторонних глаз.

> ***Замечание**: не все реализации Perl позволяют клонировать манипулятор DATA, и в общем-то не должны этого делать, но пренебрегать такой угрозой не стоит.*

Много трудностей и непонимания вызывает ***интерполяция строк***, заключенных в двойные кавычки. Язык Perl может автоматически подставлять вместо имени переменной ее содержимое, а вместо имени функции – возвращенный ею результат. Последняя возможность считается особо опасной, т.к. на первый взгляд позволяет злоумышленнику вызывать любые команды Perl и даже выполнять внешние программы с помощью функций *exec*, *eval* и многих других. Практически все руководства по программированию скриптов настоятельно рекомендуют фильтровать символы "@", "$", "[]", "{}", "()" и разработчики (даже опытные!) в большинстве своем послушно следуют этому требованию!

На самом деле никакой опасности нет – интерполяция строк выполняется только в ***текстах программ*** и никогда в ***значениях переменных***. Наглядно продемонстрировать это утверждение позволяет следующий пример (предполагается, что во втором случае с клавиатуры вводится : "${\(print '>Hello')}"; наклонным синим шрифтом выделен вывод программы на экран):

```
$filename="${\(print '>Hello')}";
print "$filename";
>Hello1
```

```
$filename=<>;
print "$filename";
${\(print '>Hello')}
```

Замена имени функции на результат ее работы в пользовательском вводе не была выполнена! Независимо от того, заключена ли введенная строка в двойные кавычки или нет, она всегда отображается на экране такой, какая есть, без каких бы то ни было преобразований. Фильтровать символы интерполяции совершенно не нужно – их использование не возымеет никакого эффекта! Тем более, что "собака" является неотъемлемой частью адреса электронной почты и отказ от нее просто невозможен.

Точно так напрасны опасения относительно обратной кавычки – "`". В документации по языку Perl сказано, что строка, заключенная в обратные кавычки интерпретируется как команды операционной системы, которой они и передаются на выполнение. Это действительно так, но только по отношению к строкам текста

программы, а не содержимому скалярных переменных. Т.е. конструкция "$a=`type /etc/passwd`;" занесет в переменную $a содержимое файла "/etc/passwd", но "$a=<>;", независимо от того, что введет пользователь, никогда не приведет к подобному результату. Поэтому символ обратной кавычки никакой угрозы в себе не несет и совершенно ни к чему его фильтровать.

Гораздо больше проблем связано с вызовом внешних программ, работающих с данными, введенными пользователем. Основная сложность заключается в невозможности заведомо узнать какие символы потенциально опасны, а какие нет. Большинство приложений помимо документированных функций имеют множество недокументированных особенностей или хуже того – ошибок реализации.

Никогда нельзя быть абсолютно уверенным, что ваш почтовый агент не воспримет вполне легальный адрес назначения как собственный ключ или управляющее сообщение. Но даже если и отмахнутся от подобных экзотических угроз, составление списка фильтруемых символов по-прежнему будет представлять проблему, т. к. из документации не всегда бывает ясно как поведет себя приложение, встретив ту или иную комбинацию символов. Помимо явно опасного перенаправления ввода-вывода, вызова конвейера, использования символов-джокеров, символов-разделителей и переноса строк, иногда приходится сталкиваться с такими неожиданными "подлостями" как, например, возможность автоматического развертывания UUE-сообщений, что, с одной стороны, вроде бы и отмечено в документации (порой между строк), но, с другой, не настолько очевидно, чтобы сразу обратить на себя внимание.

Лучше всего полностью отказаться от вызова внешних программ, реализуя все необходимое самостоятельно. Ту же процедуру отправки писем не сложно выполнить и средствами самого языка Perl, без каких либо обращений к SendMail-у или другому MTA, и файлы на диске искать не вызовом grep, а собственноручно написанным модулем. Усложнение программы компенсируется увеличением ее надежности и безопасности.

Очень важно понимать, что фильтрацию ввода можно осуществлять только на ***серверной***, но ни в коем случае не клиентской стороне! Часто эту операцию поручают Java-апппетам, а то и вовсе Java-скриптам, не подумав, что они могут быть модифицированы или блокированы злоумышленником, поскольку исполняются на его собственной машине и не существует никакого способа отличить запрос, посланный Java-скриптом от запроса, посланного самими злоумышленников в обход скрипта. Java может быть полезна лишь для быстрого уведомления клиента об ошибке ввода, но не более того!

Техника фильтрации. Существует два возможных подхода к фильтрации пользовательского ввода: а) выдача сообщения об ошибке и прекращение работы, если обнаружен хотя бы один опасный символ; б) "выкусывание" всех таких символов без выдачи предупредительных сообщений с продолжением работы.

Последний подход неоправданно популярен вопреки логике и здравому смыслу. Почему? Пусть, например, адрес легального посетителя выглядит так: "*horn&hoff@mail.org*". Программа фильтрации "видит" потенциально опасный символ "&" и "на всякий случай" решает его удалить – в результате письмо направляется совсем по другому адресу, а пользователь "ждет у моря погоды", не понимая почему оно до него не дошло.

Независимо от того, умышленно вставлен опасный символ или нет, он приводит к невозможности дальнейшего использования таких данных. Скрипт обязан прекратить работу и объяснить причину своего неудовольствия пользователю.

Поиск заданного перечня символов легче всего осуществляется использованием регулярных выражений, например, так:

```
if ($filename =~/[<>\|\-&\.\\\/\0]/)
         {die "Ошибка ввода! Недопустиый символ \"$&\" \n";}
open(fh, $filename);
...
```

Перечень потенциально опасных символов зависит от того, где и для чего их планируется использовать. Создание же универсального фильтра "на все случаи жизни" не представляется возможным.

Например, при добавлении новой записи в гостевую книгу разумно выполнить проверку на предмет присутствия "нехороших" тегов, но не стоит запрещать посетителям использовать теги вообще. В то же время, передавая e-mail посетителя внешнему MTA, необходимо убедится в отсутствии символов перенаправления стандартного ввода (синтаксически неотличимых от кавычек, обрамляющих тэги), иначе злоумышленник сможет ввести нечто вроде "*hacker2000@hotmail.com; mail hacker2000@hotmail.com </www/cgi-bin/mycgi.pl*" и получить исходный текст скрипта (или любого другого файла) "с доставкой на дом"!

Поиск уязвимых мест в скриптах значительно облегчает механизм ***меченных данных*** (*tainted data*). Если запустить Perl с ключом "-T", он станет требовать явной инициализации или фильтрации всех скалярных переменных, передаваемых функциям eval, system, exec и некоторым другим, потенциально опасным ***с его точки зрения***. Любые переменные, полученные извне (стандартный поток ввода, командная строка, переменные окружения и т. д.), считаются "***зараженными***" и не могут быть переданы "опасным" функциям до тех пор, пока не будут "обеззаражены" фильтром регулярных выражений. Если одна "зараженная" переменная присваивается другой – та тоже становится "зараженной"!

Но Perl не проверяет корректности фильтрации символов, допуская даже сквозную фильтрацию – "*(.*)*", и не считает опасной функцию *print* (как, впрочем, и многие другие). Конструкция "$a=<>; print $a" не вызывает нареканий со стороны Perl, а ведь переменная $a может содержать нехорошие тэги и вызовы SSI!

Механизм меченных данных, во всяком случае его реализация в языке Perl, – не панацея! И расслабляться программистам, от мнимой уверенности, что они надежно защищены от всех угроз, не стоит. "Зараженный режим" разумно использовать как дополнительное средство самоконтроля, но если он не нашел никаких ошибок, это еще не дает оснований считать, что их там действительно нет. Сказанное относится и к ключу "-w", заставляющего Perl выполнять дополнительные проверки, "ругаясь" при попытке чтения неинициализированных переменных, попытке модификации файловых манипуляторов и т. д. – если это и усилит безопасность программы, то на самую малость.

Уязвимость самостоятельных модулей. Редкое приложение состоит всего лишь из одного скрипта. Программная оснастка даже скромного сайта представляет собой десятки модулей, связанных друг с другом сложной иерархической зависимостью. Это порождает две основные проблемы: ***отсутствие контроля входных данных в служебных скриптах*** и ***возможность обхода системы авторизации***.

Например, посетитель вводит в форму "Search" свой запрос, браузер извлекает его и передает поисковому скрипту, который отыскав файл документа с таким содержанием, передает его имя другому скрипту, предназначенному для его отображения.

Разработчик, предполагая, что второй скрипт всегда будет вызываться только первым, скорее всего не предпримет никаких усилий по фильтрации ввода второго скрипта. Но ведь ничто не мешает злоумышленнику непосредственно вызывать этот скрипт самому, передав ему имя любого файла, который он хочет посмотреть (например, "/etc/passwd"), а не только *.html!

Любые данные, принимаемые программой извне, должны быть проверены! Необходимо убедиться в том, что запрошенный файл пользователю действительно можно смотреть. Этого не так-то просто добиться, как может показаться на первый взгляд – проверка запроса на соответствие именам запрещенных файлов ничего не решает – злоумышленник может запросить и базу данных клиентов сервера, и содержимое интересующего его скрипта, и т. д., – всего не перечислишь! Ограничение области видимости текущей директорией так же не приводит к желаемому результату, во всяком случае, без фильтрации символов обращения к вышележащем каталогам.

Надежнее всего передавать не имя отображаемого файла, а его индекс в списке доступных для просмотра файлов. Существенные недостатки такого решения – излишняя сложность скрипта и необходимость постоянной коррекции списка доступных файлов при всяком добавлении или удалении новых документов на сервере.

Некоторые приемы позволяют избежать этих утомительных проверок. Например, пусть служебный скрипт принимает в качестве дополнительного параметра пароль, известный только вызываемому коду (и, разумеется, самому владельцу сайта). В этом случае фильтрацию ввода необходимо осуществлять только в модулях, непосредственно взаимодействующих с пользователем, а во всех остальных можно смело опустить. При отсутствии ошибок реализации, связанных с обработкой и хранением пароля, злоумышленник не сможет непосредственно исполнить ни один служебный скрипт, если конечно, не сумеет подобрать пароль, чему легко противостоять (перебор даже пятисимвольного пароля по протоколу HTTP займет очень-очень много времени).

Такой подход снимает и другую проблему – возможность обхода подсистемы авторизации "ручным" вызовом нужного скрипта. В сети нередко встречаются почтовые системы, работающие по следующему алгоритму: "входной" скрипт проверяет имя и пароль пользователя и, если они верны, передает одно лишь имя пользователя (без пароля!) другому скрипту, непосредственно работающему с почтовым ящиком. Злоумышленник, вызвав последний скрипт напрямую, получит доступ к корреспонденции любого пользователя! Удивительно, но подобная ошибка встречается и в Internet-магазинах: скрипту, осуществляющему покупку, передают

не полную информацию о пользователе (номер кредитной карты, адрес и т. д.), а идентификатор, по которому этот скрипт и получает все эти сведения из базы данных, хранящейся на сервере. Если идентификатор представляет небольшое предсказуемое число (как часто и бывает), злоумышленник сможет заказать товар от имени любого из постоянных покупателей магазина.

Сюда же относятся и ошибки обработки динамически генерируемых форм. Чтобы скрипт мог отличить одного посетителя странички от другого, он добавляет в форму особое скрытое поле, содержащее имя пользователя, введенное им при регистрации (так, например, функционирует большинство чатов). Злоумышленник может сохранить переданную ему страницу на диск, модифицировать по своему усмотрению скрытое поле, выдавая тем самым себя за другое лицо.

Не стоит надеяться на проверку переменной HTTP_REFERER – она заполняется самим HTTP-клиентом и может содержать все, что угодно! Грамотно спроектированный скрипт должен помещать в скрытое поле не только имя пользователя, но и его пароль. Для чатов такая защита вполне подойдет, но в более ответственных случаях пароль следует шифровать по алгоритму с несимметричными ключами или хешировать по схеме "запрос-отклик". В противном случае злоумышленнику, перехватившему трафик, не составит большого труда его узнать. Техника шифрования – тема отдельного большого разговора, в контексте же настоящей статьи вполне достаточно указать на необходимость шифрования, устойчивую к перехвату, а как ее реализовать – это уже другой вопрос.

> ***Важное замечание**: никогда не следует передать секретные данные методом GET, поскольку, он помещает их в тело URL, а браузеры в специальной переменной хранят URL предыдущего посещенного сайта и передают эту переменную при переходе с одного сервера к другому. Конечно, вероятность того, что им окажется сервер злоумышленника очень невелика, но все-таки этой угрозой не стоит пренебрегать.*
>
> *Кроме того, браузеры Internet Explorer и Netscape Navigator заносят в общедоступный журнал URL всех посещенных за такой-то период сайтов – на компьютерах коллективного использования это приводит к возможности перехвата секретной информации, переданной на сервер методом GET. В отличие от него, метод POST помещает содержимое запроса в HTTP-заголовок, что намного безопаснее.*

Защищенность секретных файлов. Настройки скрипта, пароли и другая секретная информация, как правило, хранится не в теле программы (хотя случается и такое), а в отдельных файлах, зачастую помещенных в тот же самый каталог, в котором расположены скрипты или WEB-страницы. Это приводит к возможности просмотра их содержимого в экране браузера, стоит только узнать требуемое имя и сформировать соответствующий URI.

Единственная проблема, стоящая перед злоумышленником – узнать полный путь к этим файлам. Если просмотр WWW-директорий сервера запрещен (что вовсе не факт!), остается действовать последовательным перебором. При условии, что файлы имеют осмысленные имена (как часто и бывает) перебор не будет слишком долгим. К тому же, используя общедоступные скрипты не все WEB – мастера изменяют имена секретных и конфигурационных файлов. Наконец, получить содержимое директорий можно и через ошибки реализации других сетевых служб и самого сервера.

Словом, никогда не стоит надеяться, что злоумышленник не сможет выкрасть секретный файл только потому, что не знает его имени. Гораздо надежнее разместить его так, чтобы WEB-клиенты не имели к нему никакого доступа (скрипту, исполняющемуся с привилегиями локального пользователя, это не причинит никаких проблем и он по-прежнему сможет читать и писать в такой файл). Другое возможное решение – сконфигурировать WEB-сервер так, чтобы он не видел эти файлы или запрещал к ним доступ.

Оба способа требуют для своей реализации определенных привилегий, которые предоставляются не всеми провайдерами (и очень немногими службами бесплатного хостига) – тогда приходится помещать секретную информацию в сам Perl-скрипт. При отсутствии ошибок реализации и конфигурации WEB-сервера клиент никогда не может увидеть содержимое скрипта, а только результат его работы. Напротив, сам скрипт может работать с самим собой как с обычным текстовым файлом. Для этой цели удобно использовать лексему DATA, доступную через одноименный манипулятор.

Такой прием обеспечивает достаточно высокую защищенность секретных данных, но срабатывает не во всех случаях, случается, что сервер в силу определенных обстоятельств отображает не результат работы, а содержимое скрипта. Забавно, что для исправления такой ошибки разработчикам Microsoft Internet Information Server пришлось выпустить три (!) следующих одна за другой заплатки, прежде чем проблема была решена.

Сперва обнаружилось, что точка, добавленная к имени скрипта, вместо запуска приводила к отображению его содержимого. Это быстро исправили, впопыхах не вспомнив, что ту же самую точку можно представить и в виде шестнадцатеричного кода, предваренного символом процента. Пришлось вслед за первым выпускать второй пакет, а за ним еще и третий, поскольку выяснилось, что обращение к файлу по альтернативному короткому имени так же приводило к показу его содержимого. И до сих пор ни у кого нет полной уверенности, что выловлены все ошибки и завтра не откроется какая-нибудь новая.

Всегда следует учитывать угрозу просмотра секретного файла и соответствующим образом ей противостоять. Это вовсе не так трудно, как может показаться!

Основным объектом охоты злоумышленников обычно становятся пароли, а их легко зашифровать встроенной в Perl функцией *crypt*. Строго говоря, *crypt* пароли не шифрует, а хеширует, подвергая их необратимому преобразованию, поэтому, даже получив доступ к секретному файлу злоумышленник не сможет восстановить ни один пароль иначе, чем тупым перебором всех возможных вариантов.

Упрощенный пример программы аутентификации с использованием функции ***crypt*** приведен ниже (в нем не используются привязка, поэтому, одинаковые пароли будет иметь идентичные хеши, что в сотни раз ослабляет стойкость защиты ко взлому лобовым перебором).

> ***Внимание****: в некоторых реализациях Perl функция crypt возвращает переданную ей строку без каких бы то ни было изменений. Проверьте, как ведет себя ваша версия Perl! В крайнем случае придется создавать собственную реализацию Perl (в этом помогут исходные тесты UNIX-подобных операционных систем).*

```
print "Password:";                      #запрос пароля пользователя
$passwd=<>;
chop $passwd;                           #удаление символа \n
$encrypt=crypt($paswsd,"sl");           #вычисления хеша пароля
open (fh,"passwd") || die;              #открытие файла паролей
while($pass=<fh>){                      #поиск подходящего пароля
         chop $pass;                    #удаление символа \n
         if ($pass eq $encrypt){        #подходящий пароль?
                print "Password ok\n";
                $flag=1;                #пароль найден, установить флаг
         break;                         #и выйти из цикла
         }
}
if (!$flag)                             # если пароль не найден - выход
         {print "BAD password!\n"; die;}
```

Использование необратимого шифрования даже при наличии других ошибок реализации и неправильном администрировании сервера чрезвычайно затрудняет атаку, делая ее практически неосуществимой. Во всяком случае, взломщику понадобится весьма сильная мотивация, чтобы тратить свое время на такую защиту. Хакеры, как и угонщики, редко охотятся за какой-то одной, конкретно выбранной машиной – они скорее найдут растяпу, забывшего ключи в замке зажигания, чем будут ювелирно нейтрализовать сложную систему сигнализации.

Ненадежность переменных окружения. Переменные окружения очень удобны для хранения различных настроек и организации межпроцессорного взаимодействия. Единственный существенный их недостаток – абсолютная незащищенность. Бытует мнение, что удаленный пользователь не может манипулировать переменными окружения на сервере, а потому их использование вполне безопасно.

Но это не так! Протоколы telnet, FTP и некоторые другие даже анонимному пользователю разрешают не только читать, но и изменять любые переменные окружения по своему желанию. Поэтому полагаться на них, право же не стоит.

Особенно опасно хранить в них пути к библиотекам или файлам данных – в этом случае, изменив переменную окружения, злоумышленник сможет "подсунуть" программе свою "троянизированную" библиотеку, последствия исполнения которой очевидны. Однако, описание подобных атак, выходит за рамки протоколов HTTP и CGI, поэтому в этой статье подробно не рассматривается.

Заключение

Единственный способ гарантированно избежать ошибок – не писать программы! Древние мудрецы говорили – "*падает, тот кто бежит; тот, кто лежит – уже не падает*". Допуская ошибки, человек приобретает иммунитет, помогающий впредь их не совершать. Только так, на собственном опыте и приобретается профессионализм. Покорное следование советам руководств по безопасности незначительно увеличивает качество кода – "дырка" вылезет не в одном, так в другом месте. (Это не значит, что данная статья бесполезна, просто не стоит строить иллюзий, будто бы она автоматически решит все проблемы).

Умение писать надежный безопасный код сродни езде на велосипеде – пока сам этому не научишься, никакие наставления не помогут. Но даже научившись, вряд ли сможешь внятно объяснить другим как следует держать равновесие и что конкретно требуется для этого делать.

Единственное, от чего хотелось бы предостеречь – не выезжайте на автомагистраль, то есть не беритесь за ответственные проекты, пока не будете полностью уверены в своих силах и опыте.