Einiges zum Thema CGI-Sicherheit

Die Sicherheit bei Web-Anwendungen ist ein wichtiges Thema und vor dem Hintergrund dass es keine 100%ig sicheren Webanwendungen gibt, ist es umso wichtiger, die Anwendungen möglichst nah an die 100% zu bringen. Dieser Artikel kann nicht alles zum Thema Sicherheit bei CGI-Programmen aufzeigen, aber einige wichtige Punkte ansprechen.

Benutzereingaben

Es gibt einen Grundsatz, den man immer berücksichtigen sollte: Traue keiner Benutzereingabe!

Als Programmierer sollte man immer denken „Der User will mir schaden“. Man sollte also eine gesunde Paranoia entwickeln. Besonders wichtig ist dabei auch die Erkenntnis, dass die meisten Angriffe auf eine Anwendung nicht „von außen“, sondern „von innen“ kommen. In einer Firma bedeutet das, dass die meisten Angriffe von den eigenen Mitarbeitern ausgehen. Dies sind sehr häufig keine gewollten Angriffe, sondern der Mitarbeiter gibt in einem kleinen Moment der Unachtsamkeit etwas Falsches ein, oder er weiß gar nicht, dass er mit einer bestimmten Eingabe etwas Böses anrichten kann.

Bei der Überprüfung von Benutzereingaben sollte nach Möglichkeit das „Whitelist“-Verfahren verwendet werden. Dabei werden die Eingaben daraufhin überprüft, ob sie nur *erlaubte* Zeichen enthält. Im Gegensatz dazu wird häufig das „Blacklist“-Verfahren angewendet, bei dem die Eingaben daraufhin überprüft werden, ob sie *nicht-erlaubte* Zeichen enthalten. Auf den ersten Blick sehen diese beiden Verfahren sehr ähnlich aus, aber bei genauerer Betrachtung sieht man, dass bei dem „Blacklist“-Verfahren auch eher mal unerwünschte Eingaben durchrutschen, weil nicht *alle* Zeichen als „nicht erlaubt“ markiert sind, die eine negative Auswirkung auf das System haben können.

Bei dem „Whitelist“-Verfahren werden vielleicht „zu viel“ Eingaben abgeblockt. Aber das ist leichter zu beheben als ein kompromittiertes System nach einer unvollständigen „Blacklist“.

CGI.pm

Als allererstes sollte man das Modul CGI.pm verwenden. Es nimmt einem viel Arbeit ab und reduziert so die Fehlerwahrscheinlichkeit. Das Modul wird schon seit einigen Jahren entwickelt und hat somit einen sehr guten Stand. In vielen CGI-Skripten findet man noch eine Funktion „ReadParse“ (Listing 1), die wahrscheinlich das schlechteste Überbleibsel aus „Matt‘s Script Archive“ ist. Dort werden die CGI-Parameter geparst und in ein Hash gespeichert. Dieses Parsen ist allerdings nicht so sicher wie die Verwendung der `Vars`-Methode aus dem CGI-Modul.

Da ist

```
use CGI;
my %in = CGI::Vars();
```

viel kürzer, übersichtlicher und sicherer.

Taint-Modus

Man sollte das Skript immer im Taint-Modus laufen lassen. In diesem Modus sorgt Perl dafür, dass Benutzereingaben automatisch als „tainted“ (befleckt) markiert werden und in

kritischen Funktionsaufrufen nicht verwendet werden können. Das Tainting funktioniert allerdings nicht automatisch bei der Verwendung von Modulen mit XS- oder C-Anteil, hier ist Vorsicht geboten.

Um die Skripte im Taint-Modus laufen zu lassen wird `perl` beim Aufruf die Option `-T` mitgegeben. Unter UNIX/Linux geht das einfach in der Shebangzeile am Anfang des Skriptes:

```
#!/usr/bin/perl -T
...
```

Leider gibt es hier einen Unterschied zwischen Linux und Windows. Bei Windows bringt es nichts, wenn das `-T` im Shebang steht, weil die Shebangzeile nicht vor dem Aufruf des Interpreters ausgewertet wird. Die Option muss schon beim Starten des Interpreters bekannt sein. In der Webserverkonfiguration (je nach Webserver unterschiedlich) muss festgelegt werden, wie der Webserver mit den Dateiendungen `.pl` oder `.cgi` umgehen soll. Dort muss das `-T` an den Pfad zum Perl-Interpreter angehängt werden. Dabei ist zu beachten, dass die Einstellung dann nicht nur für ein einzelnes Skript gilt, sondern für alle Skripte im so konfigurierten Teil.

Werden „tainted“ Variablen zum Beispiel bei einem `open`-Aufruf verwendet, erscheint in den Logfiles eine Fehlermeldung, dass potentiell gefährliche Daten verwendet werden:

```
Insecure dependency in open
   while running with -T switch at skript.pl
```

XSS

Ein Schlagwort, das in den letzten Monaten immer bekannter wurde, ist „Cross-Site-Scripting“ (XSS). Beim Cross-Site Scripting wird Code auf Seite des Clients ausgeführt. Daher muss der Angreifer seinem Opfer einen präparierten Hyperlink zukommen lassen, den er zum Beispiel in eine Webseite einbindet oder in einer E-Mail versendet. Gefährlich wird es besonders dann, wenn die Quelle eigentlich vertrauenswürdig ist - zum Beispiel ein Forum, in dem man sich schon seit Jahren bewegt.

Dort könnte der Angreifer die manipulierten Links in eine Privatnachricht packen oder in einen Thread. Deshalb muss hier der Programmierer besonders aufpassen.

```
sub ReadParse {
  # Read in text
  if ($ENV{'REQUEST_METHOD'} eq "GET") {
    $in = $ENV{'QUERY_STRING'};
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
    for ($i = 0; $i < $ENV{'CONTENT_LENGTH'}; $i++) {
      $in .= getc;
    }
  }

  @in = split(/&/,$in);

  foreach $i (0 .. $#in) {
    # Convert plus's to spaces
    $in[$i] =~ s/\+/ /g;

    # Convert %XX from hex numbers to alphanumeric
    $in[$i] =~ s/%(..)/pack("c",hex($1))/ge;

    # Split into key and value.
    $loc = index($in[$i],"=");
    $key = substr($in[$i],0,$loc);
    $val = substr($in[$i],$loc+1);
    $in{$key} .= "\0" if (defined($in{$key})); # \0 is the multiple separator
    $in{$key} .= $val;
  }
}
```

Listing 1

Ein klassisches Beispiel für Cross-Site Scripting ist die Übergabe von Parametern an ein CGI-Skript einer Website. Ein kleines Beispiel ist in Listing 2 zu sehen, während Listing 3 zeigt, wie einfach diese Lücke geschlossen werden kann. Wer die möglichen Auswirkungen mal testen will, kann in das Eingabefeld `<script language="javascript">alert('test');</script>` eingeben und das Formular abschicken.

HTML und BBCode

Injizierter HTML- und Javascript-Code kann zu Angriffen auf die Client-Seite einer Anwendung verwendet werden. Wenn man gewisse HTML-Elemente zulassen will (z.B. in einem Forum), ist es geschickt, eines der BBCode-Module von CPAN zu benutzen. So kann man alle HTML-Elemente, die der Benutzer direkt eingibt, „unschädlich" machen, z.B. mit HTML::Entities.

HTML-Escaping

Die Template-Module HTML::Template und HTML::Template::Compiled bieten auch die Möglichkeit, ein „default_escape" zu setzen. Ist der Standard-Wert auf ‚html' eingestellt, werden automatisch bei allen Parametern die Sonderzeichen in Entities umgewandelt. Das vereinfacht das Verhindern von XSS.

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;

my $cgi = CGI->new;
print $cgi->header;
my %params = $cgi->Vars;

if( $params{action} ){
    print "Sie haben $params{input} eingegeben";
}
else{
    print qq~
        <form method="post">
            <input type="text" name="input" />
            <input type="hidden" name="action" value="1" />
            <input type="submit" value="abschicken" />
        </form>
    ~;
}
```

Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;
use HTML::Entities;

my $cgi = CGI->new;
print $cgi->header;
my %params = $cgi->Vars;

if( $params{action} ){
    my $var = HTML::Entities::encode_entities( $params{input} );
    print "Sie haben $var eingegeben";
}
else{
    print qq~
        <form method="post">
            <input type="text" name="input" />
            <input type="hidden" name="action" value="1" />
            <input type="submit" value="abschicken" />
        </form>
    ~;
}
```

Listing 3

Datenbanken

In Datenbanken werden häufig vertrauliche Daten gespeichert - seien es Namen und Adressen von Kunden oder Zugangsdaten zu einer Plattform. Deshalb sollte hier ein besonderer Augenmerk auf der Sicherheit liegen. Angreifer versuchen mit sogenannten SQL-Injections Informationen über die Datenbank und die Inhalte zu bekommen.

Das DBI-Modul bietet die wunderbare Möglichkeit, die ?-Notation (Platzhalter) zu verwenden. Damit werden automatisch alle Sonderzeichen gequotet und die Möglichkeit des Injizierens von SQL-Code verhindert. Das Gleiche kann mit der Funktion `quote` aus dem Modul erreicht werden. Weiterhin gibt es eine (experimentelle) Möglichkeit, Ein- und Ausgabewerte als „tainted“ zu markieren (Optionen TaintIn, TaintOut, Taint).

Als Beispiel für eine mögliche Gefahr soll hier an einem Login-Vorgang gezeigt werden, wie gefährlich dies sein kann (Listing 4) und wie es sicherer (Listing 5) ist.

```
my $stmt = "SELECT count(*) FROM users
   WHERE id = '$name' and
         password = "$password'";
my $sth  = $dbh->prepare( $stmt );
$sth->execute;
```

Listing 4

Hier werden Formulareingaben nicht auf SQL-Sonderzeichen überprüft und einfach in den SQL-Befehl eingebaut. Der User soll hier als authentifiziert gelten wenn `count` größer 1 ist. Wenn ein Angreifer sowohl für `$name` als auch für `$password ' OR '1' = '1` eingibt, würde das diesen SQL-Befehl erzeugen:

```
SELECT count(*) FROM users
    WHERE id = '' OR '1' = '1' AND
    password = '' OR '1' = '1'
```

Das liefert die Anzahl der Einträge in der Tabelle users. Der Angreifer wäre eingeloggt.

```
my $stmt = "SELECT count(*) FROM users
   WHERE id = ? and
         password = ?";
my $sth  = $dbh->prepare( $stmt );
$sth->execute( $name, $password );
```

Listing 5

Durch die `?` wird das quoting hervorgerufen und DBI macht die Sonderzeichen unschädlich. Für das obige Beispiel wird so folgender SQL-Befehl ausgeführt:

```
SELECT count(*) FROM users
   WHERE id = '\' OR \'1\' = \'1' AND
   password = '\' OR \'1\' = \'1'
```

So wäre der Angreifer nicht eingeloggt.

Öffnen einer Datei

Grundsätzlich sollte man nicht den User bestimmen lassen, wie der Dateiname lautet. Man sollte eigentlich immer selbst festlegen, wie die Datei heißt und wo sie hingespeichert wird.

Außerdem ist die Drei-Parameter-Form von open() (also `open (FILEHANDLE, ">", $dateiname)`) sicherer, da das Umlenkungszeichen damit eindeutig festgelegt wird. Beim Verwenden des Taint-Modus wird man automatisch vor der Verwendung von unsicheren Dateinamen geschützt.

Öffnen einer Pipe

Wenn z.B. mit sendmail eine E-Mail verschickt werden soll, wird gerne der typische Fehler gemacht, die E-Mail-Adresse des Empfängers direkt in die Kommandozeile zu schreiben:

```
open PIPE, "|/usr/lib/sendmail $empfaenger";
```

Was fällt auf? Genau, `$empfaenger` ist hier vermutlich eine Benutzereingabe. Der sollten wir niemals trauen. In die Kommandozeile gehören nur selbst festgelegte Parameter. Den Empfänger kann man auch mit der Option -t und einer „To:“-Zeile bestimmen.

```
# Etwas besser, aber noch nicht gut:
open PIPE, "|/usr/lib/sendmail -t" or die $!;
print PIPE <<EOM;
To: $empfaenger
Subject: Blubber

Hallo $name,
usw.
EOM
...
```

Aber auch das ist nicht sicher.
Denn $empfaenger oder $name kann mehrere Zeilen beinhalten, die noch ein paar zusätzliche Header-Zeilen wie z.B. „Bcc: boeser_bube@domain.example“ beinhalten oder auch eine komplette E-Mail hinzufügen kann. sendmail betrachtet eine E-Mail als fertig, wenn im Body der Punkt alleine auf einer Zeile vorkommt.

Also, Usereingaben gehören auch nicht ungeprüft in den Header einer E-Mail, und sendmail startet man mit der Option „-oi“, die das mit dem erwähnten Punkt verhindert.

```
# Prüfe $empfaenger auf Gültigkeit,
# z.B. mit Email::Valid
# Ersetze stumpf alle unerwünschten Zeichen:
$empfaenger =~ tr/\r\n\t\f\0//d;
open PIPE, '|-',
   '/usr/lib/sendmail -t -oi' or die $!;
print PIPE <<EOM;
From: test@test.com
To: $empfaenger
Subject: Blubber

Hallo $name,
usw.
EOM
...
```

Häufig kann man solche Pipe-Verwendungen vermeiden, indem man z.B. auf Module zurückgreift. Im Falle von „Mails versenden“ kann man z.B. das Mail::Sender-Modul benutzen.

Verwendung von eval

Man sollte im Allgemeinen auf String-evals mit Daten aus unsicheren Quellen verzichten, da der String eine „böse“ Zeichenfolge beinhalten könnte. Auch hier wird man im Taint-Modus geschützt.

Wenn doch ein String-eval mit unbekannten Inhalt verwendet werden soll, kann man auf das Modul Safe zurückgreifen, mit dem man bestimmte Perl-Funktionen (Opcodes) erlauben oder ausschalten kann. Es ist allerdings möglich, dass Safe Sicherheitslücken hat!

Neueste Perl-Version

Auch das Perl-Binary kann typische, aus der C-Welt bekannte Fehler wie Buffer Overflows enthalten. Deshalb sollte man möglichst die neueste Perl-Version verwenden. Bei kritischen Bugfixes werden auch die älteren Stränge (5.6.x, 5.003_xx) aktualisiert.

Aktuelles

Wie viele Schwachstellen es in Webanwendungen gibt, kann man nur erahnen. Auf dem 24. Chaos Computer Congress (24C3) haben Hacker etliche Webseiten „angegriffen“ und kleine Veränderungen vorgenommen. Unter http://events.ccc.de/congress/2007/Hacks sind weit über 150 verschiedene Hacks aufgeführt, die während des Kongresses gemacht wurden. Viele dieser Angriffe sind SQL-Injections und XSS-Attacken. Aber auch Fälle, in denen über die URL `/etc/passwd` geöffnet werden konnten.

Um solche Attacken durchzuführen muss man kein Spezialist sein. Es sind häufig ganz einfache Sachen, die man über jede Suchmaschine finden kann.

Renée Bäcker

TPF-TICKER

Parrot Grant Update

Es gibt wieder Neuigkeiten von Parrot (http://www.parrotcode.org):

Am 17. Oktober wurde Parrot 0.4.17 veröffentlicht. Neben einigen Bugfixes wurde auch an NQP weitergearbeitet. NQP ist eine abgespeckte Version von Perl6, die sich zum Compilerbau eignet.

Einige Design-Milestones wurden erreicht, so dass auch wieder Geld vergeben wurde.

Die Roadmap und eine Übersicht über das ausgegebene Geld ist unter http://www.perlfoundation.org/parrot_grant_from_nlnet zu finden.