

Sicherheit in Webanwendungen

Seminar: “Computersicherheit für Paranoiker”, SS06

Sebastian Sproesser
<sebastian@sproesser.name>

Agenda

- Perl
- Directory Traversal
- Poison NULL Byte
- E-Mail Injection
- SQL Injection
- XSS
- Fazit
- Quellen

Perl

Allgemeines

- “Practical Extraction and Report Language”
- Erfunden in den 80ern von Larry Wall
- Ursprünglich: Textmanipulationen
- Ideal für textlastige Medien
- “The glue that holds the internet together.”

Perl

Grundlagen

- 3 verschiedene Datenstrukturen:
 - Skalar (\$)
 - Array (@)
 - Hash (%)
- reguläre Ausdrücke: Mächtige Textmanipulationen
- Variablendeklaration optional

4 / 46

Standardmäßig Variablendeklaration optional, kann aber mittels “use strict;” erzwungen werden.
Deklarationsschlüsselwort “my”.

Perl

Skalar (\$)

- Fasst:
 - Zeichenkette
 - Zahl (Integer oder Fließkomma)
 - Referenz
- kontextsensitiv (visuell)

5 / 46

- Stringlänge nur durch Speicher begrenzt
- “Aussehen” der Operatoren wichtig:

Stringoperatoren: eq ne cmp

Zahloperatoren: == != <=>

Dateitestoperatoren: -e -f -d

Perl

Arrays (@)

- Fassen mehrere Skalarwerte
- Verwirrungsgefahr
 - @ → Array
 - \$ → Skalar
- Auch hier: Kontextsensitivität!

```
my @array;  
$array[5] = 'Test';
```

6 / 46

```
$a = 'abc';  
@a = ($a, 'def', 'ghi');  
$a = 'xyz';  
print $a[0];          # -> abc  
print $a;            # -> xyz  
for $value (@a) {  
    print $value . "-"; # -> abc-def-ghi-  
}
```

Kontextsensitivität geht so weit, dass
Funktionen je nach Kontext
Unterschiedliches zurückgeben.

Perl

Hashes (%)

- assoziative Arrays

```
my %hash;  
$hash{'value'} = 5;
```

7 / 46

```
%a = ('key1' => 'value1',  
      'key2' => 'value2');
```

```
print ${key1}; # -> value1
```

```
for $key (keys %a) {  
    print $key . '->' . ${$key} . ' - ';
```

```
}  
# -> key1->value1 - key2->value2
```

Schlüssel können alles sein (Strings,
Integer, Referenzen)

Perl

Sonstiges

- Default-Variable: \$_
- Einzeiliges for, if, while wird nachgestellt!

8 / 46

```
@a = ('a', 'b', 'c');  
print for (@a);      # -> abc  
# Entspricht:  
# for $value (@a) {  
#   print $value;  
# }
```

Perl

Reguläre Ausdrücke

- Regulärer Ausdruck: Ein String, der ein Muster beschreibt
- Wesentlicher Faktor bei Textmanipulationen

```
if ("abcde" =~ /a/) {  
    print "Match!\n";  
} else {  
    print "Kein Match!\n";  
}
```

Perl

Reguläre Ausdrücke

- Metacharaktere `{ } () ^ $. | * + ? \` escapen
 - `.`: beliebiges Zeichen
 - `[]`: Zeichenklasse
 - `*`: vorangestelltes Zeichen mehrfach (auch 0mal, greedy)
 - `+`: vorangestelltes Zeichen mehrfach (mindestens 1mal, greedy)
 - `^/$`: Zeilenanfang/ende
- greedy-Operatoren nehmen so viel wie möglich

Perl

Reguläre Ausdrücke

```
$a = 'Hallihallo Welt';  
$a =~ /^Hallihallo/;      # Match  
$a =~ /Hallihallo$/;     # Kein Match  
$a =~ /Hal+ihal*o/;      # Match  
$a =~ /Had*llihallo/;    # Match  
$a =~ /^[Hh]all[io]+ Welt$/; # Match
```

Perl

Reguläre Ausdrücke

- `$a =~ s/RegExp/String/` im Klartext:
“Ersetze in `$a` das erste Vorkommen, auf das `RegExp` matcht, durch `String`”
- “magische” Variablen `$1` bis `$9`: Merken von Werten

12 / 46

```
$a = 'Hallo Welt';  
$a =~ s/(Hall)o (.*)/$2, $1hallo!/  
print $a; # -> Welt, Hallihallo!
```

Perl

“TIMTOWTDI
There Is More Than One Way To Do It”
- Inoffizielles Perl-Motto

“Perl: Der geglückte Versuch, einen Brindump direkt
ausführbar zu machen.”
- Fachbegriffe der Informatik, #282

Directory Traversal

- Klassischer Anfängerfehler
- Fehlerquelle:
 - Programmierfehler
 - großzügige Zugriffsrechte
- Anwendungsszenario: dynamische Dateieinbindung, Dateiname benutzerdefiniert

Directory Traversal

```
[...]  
<a href="cgi-bin/trav.pl?f='page.html'">Link</a>  
[...]
```

trav.pl:

```
use CGI qw/param/;  
$filename = param('f');  
  
print "Content-Type: text/html\n\n";  
  
open(FILE, $filename);  
while ($_ = <FILE>) {  
    print $_;  
}  
close(FILE);
```

Directory Traversal

- Knackpunkt:
open(FILE, \$filename);
- \$filename ungeprüft
- cgi-bin/trav.pl?f=../../../../etc/passwd
- Zu viele “..” egal

Directory Traversal

Pesky Pipe Problem

- Scheinbar nicht schlimm
- Allerdings: `open(FILE, "/usr/bin/ls|");` führt ls aus!
- Dateien lesen, Befehle ausführen →
Webserveraccount kompromittiert
- `/proc` → Wertvolle Systeminformationen!

Directory Traversal

- Lösungsvorschlag:
open(FILE,\$filename);
wird zu
open(FILE, \$filename.'.html');
- cgi-bin/trav.pl?f=../../../../etc/passwd öffnet
/etc/passwd.html → existiert nicht
- Ebenso Lösung des Pesky Pipe Problem
- Oder?

Poison NULL Byte

- Recht Perl-spezifisch
- Perl greift auf C-Library-Funktionen zurück
- Für Perl:
 - 0-Byte ist Teil eines Strings
- Für C:
 - 0-Byte markiert Ende eines Strings

Poison NULL Byte

- 2. Directory-Traversal-Beispiel:
open(FILE,\$filename.'.html');
- 0-Byte anhängen:
cgi-bin/trav.pl?f=../../../../etc/passwd%00
- Datei: /etc/passwd\0.html
- C-Funktion (2)open bekommt C-String, öffnet
/etc/passwd
- Lösung: s/\0//g; oder SSI

20 / 46

SSI: Server Side Includes

file.shtml

```
<!--#if expr="$QUERY_STRING=news" -->  
<!--#include file="inc/news.html" -->  
<!--#else -->  
<!--#include file="inc/main.html" -->  
<!--#endif -->
```

Achtung: Nix mit

```
<!--#include file="inc/$QUERY_STRING.html" -->
```

E-Mail Injection

- Beliebt für Spamversand
- Fehlerquelle:
 - mail()-Aufruf in PHP
 - sendmail-Aufruf
- Worst Case: beliebige Mails an beliebige Empfänger

21 / 46

Webserver wird unfreiwillig zum Spam-Relay

E-Mail Injection

Beispiel in PHP:

```
<?php mail($recipient,  
    $subject,  
    $message,  
    $headers); ?>
```

Erzeugt:

```
To: $recipient  
Subject: $subject  
$headers
```

```
$message
```

E-Mail Injection

Beispiel in PHP:

```
<?php mail("recipient@victim.xxx",  
  "Hello",  
  "Hi,\nYour site is great.\nBye",  
  "From: sender@anonymous.xxx\n"); ?>
```

Erzeugt:

```
To: recipient@victim.xxx  
Subject: Hello  
From: sender@anonymous.xxx
```

```
Hi,  
Your site is great.  
Bye
```

23 / 46

Subject, From, Message kommen “direkt”
aus Webinterface

E-Mail Injection

- Knackpunkt: Zusätzliche Header, insbesondere From durch Userinput!
- RfC822 (Standard for the Format of ARPA Internet Text Messages) definiert u.a.
 - Cc: (Carbon-Copy)
 - Bcc: (Blind Carbon-Copy)
- Headerzeilen getrennt durch "Line Feed"
(Hex-Code: 0x0A)

E-Mail Injection

Beispiel in PHP:

```
<?php mail($recipient,  
  $subject,  
  $message,  
  $headers); ?>
```

Erzeugt:

```
To: $recipient  
Subject: $subject  
$headers
```

```
$message
```

E-Mail Injection

Beispiel in PHP:

```
<?php mail("recipient@victim.xxx",  
  "Get Rich F@ST!!!!!!1eins",  
  "My Message...",  
  "From: sender@a.xxx%0ABcc:somebloke@grrrr.xxx"); ?>
```

Erzeugt:

```
To: recipient@victim.xxx  
Subject: Get Rich F@ST!!!!!!1eins  
From: sender@a.xxx  
Bcc:somebloke@grrrr.xxx,someotherbloke@oooops.xxx
```

My Message...

26 / 46

Vorteil: Wenigstens bekommt der festverdrahtete Empfänger mit, dass da was war.

Erwähnenswert: Manipulation von From oder Subject klappt nicht, hier lässt PHP nur einzeiliges zu. => Gaukelt Sicherheit vor

E-Mail Injection

- FormMail: Sehr bekanntes Perl-Script zum Mailversand per Webformular
- Frühere Version: gravierende Sicherheitslücke
- Empfänger: Verstecktes Feld im Formular

27 / 46

(Seit 1995 > 2.000.000 Downloads)

Inzwischen wird Empfänger mit interner Liste abgeglichen

E-Mail Injection

- Anderer, “beliebter” Anfängerfehler:

```
open(MAIL, '|/usr/lib/sendmail -t') or die;  
print MAIL <<EOM;  
To: $empfaenger  
From: $sender  
...  
EOM
```

- Manipulation an allen userdefinierten Headern

E-Mail Injection

- Scheinbare Lösung:
die "Zeilenumbruch!" if (\$from =~ \n/);
- Leider: \n plattformabhängig
- Besser: Positives Filtern
- Problem: Erkennung von gültigen Mailadressen

29 / 46

\n entspricht:

- UNIX, Linux, MacOS X: <CR> \n
- Mac (bis OS9): <LF> \r
- Windoze: <CR><LF>, \r\n

RfC822 erlaubt u.A. Kommentare u.ä.:
sebastian(hallo,
leute!)@sproesser(muhaha).org

E-Mail Injection

- Evtl. einfacherer (unvollständiger) regulärer Ausdruck:

```
$mail =~ /^w+@[a-z0-9](w\.)+$/
```

- Problem: Im Localpart (fast) alle Zeichen erlaubt: abc&def...@example.invalid
- Lösung: Vorgefertigte Module verwenden, z.B. Email::Valid, RFC:RFC822::Validate

SQL Injection

- Fehlerquellen:
 - Zunehmende Verwendung von LAMP-Systemen (Linux, Apache, MySQL, PHP)
 - Unerfahrene (PHP- ?) Programmierer
 - Programmierfehler
 - Rechte
- Weit verbreitet

SQL Injection

- “Piggyback”-Prinzip
- Typisches Beispiel:

`http://server/cgi-bin/sqlinject.pl?id=5`

```
$id = $cgi->param('id');  
[...]  
$mysql->execute(  
    "SELECT * FROM table WHERE id=$id"  
);
```

33 / 46

Zu legalen SQL-Abfragen potentiell
Schadhaftes einschleusen

SQL Injection

- Mehrere Abfragen/Anweisungen pro Zeile
(korrekte Syntax)
- `id=5 => id=5; drop database;` (destruktiv!)
- Meist komplexere Anfragen:
`SELECT * FROM table1, table2 WHERE table1.id=$id AND
table2.id = table1.table2;`
- Kommentarzeichen: `--`
`id=5 => id=5; drop database; --`

SQL Injection

- Beliebtes Angriffsziel: PHP-Webforen (phpBB, vBulletin)
- Alte Müsli-Version (Übungsgruppenverwaltung in der Mathe) anfällig

```
$query = "SELECT * from " . $_SESSION['short_name'] . '_groups  
WHERE id=' . $_POST['selected_group'] . '";
```

SQL Injection

- Sehr großes Unfugspotential:
 - Datenbanken löschen/leeren
 - User-IDs, Passworte (oder -Hashes) auslesen/ändern
 - Stored Procedures
 - ...

SQL Injection

- Problem: DB-Sonderzeichen (z.B.: ; --)
- Gegenmaßnahme: Strings filtern (DBMS hilft)

Statt

`$db->execute("SELECT * FROM table WHERE id=$id")` **besser:**

`$db->execute("SELECT * FROM table WHERE id = ?", 0, $id);`

XSS

- XSS = “Cross Site Scripting”
- Bezeichnung unzureichend:

“This issue isn't just about scripting, and there isn't necessarily anything cross site about it. So why the name? It was coined earlier on when the problem was less understood, and it stuck. Believe me, we have had more important things to do than think of a better name.”

-- Marc Slemko (XSS-Pionier)

- 3 Typen von XSS-Lücken

XSS Type 0

- Client-seitig
- JavaScript holt Parameter; schreibt ungefiltert
- Parameter manipulieren
- Ausführung mit Rechten der Ursprungsseite
- Untergrabung des Sandbox-Prinzips

39 / 46

Parametermanipulation: z.B.
neues Script laden

XSS

Type 0

- Opfer muss manipulierten Link klicken
(Social Engineering)
- Altbekannte Lücke
- Sensibilisierung (?) der User durch
Medienberichte über Phishing

XSS

Type 1

- Verbreitetste Lücke
- Nicht-Persistent (wie Type 0)
- Serverseitig
- Parameter ungefiltert auf nächster Seite
- Beispiel: Einfaches Suchinterface

41 / 46

Beispiel für Umlaute: <http://www.uloc.de/>

XSS

Type 1

- Wieder: “Social Engineering”
 - User ruft manipulierten URL auf
- Oft als nicht sicherheitskritisch eingestuft

XSS Type 2

- Persistent
- Gefährlichste Form
- Server speichert Userdaten ab, stellt diese allgemein bereit:
 - Profile
 - Postings in Webforen
- Keine Filterung der Daten
 - (Fast) alles möglich!

43 / 46

Beliebiges Javascript, ...

XSS

Type 2: MySpace-Wurm

- Bekannter Fall: MySpace-Wurm

“MySpace ist eine englischsprachige Website, die es ermöglicht, Fotos, Blogs, Nutzerprofile und Gruppen kostenlos einzurichten. MySpace wird als die bekannteste sogenannte "social networking"-Website angesehen. Laut "Alexa Ranking" ist es die fünft-beliebteste englischsprachige Website.”

-- <http://de.wikipedia.org/wiki/MySpace>

Fazit

Goldene Regel der Webprogrammierung:

Never trust user input!

45 / 46

Deshalb: Nie bestimmte Zeichen verbieten, immer nur erlaubte Zeichen zulassen. Nie nur Client-Side (JavaScript o.ä.) filtern.

Quellen

- Perl-Dokumentation (Ansatzpunkt: man perl)
- [http://\(de|en\).wikipedia.org/](http://(de|en).wikipedia.org/)
- Phrack #55 <http://www.phrack.org/>
- [http://www.securephpwiki.com/index.php/
Email_Injection](http://www.securephpwiki.com/index.php/Email_Injection)
- [http://www.cpan.org/authors/Tom_Christiansen/
scripts/ckaddr.gz](http://www.cpan.org/authors/Tom_Christiansen/scripts/ckaddr.gz)

Perl

Skalare (\$) als Strings

- Wichtigste Stringhilfsmittel: ' " . eq ne cmp x

```
$a = 'abc' . 'def';  
print $a;          # -> abcdef  
print "$a";       # -> abcdef  
print '$a';       # -> $a  
if ($a eq 'abcdef') {  
    print 'a' x 6;  # -> aaaaaa  
}
```

Perl

Skalare (\$) als Zahlen

- Wichtigste Zahlenoperatoren: + - * / ** ++ --
- Achtung: Kontextsensitivität!

```
$a = 42; $b = 3.141; $c = '1abc'; $d = 'aaa';  
print $a ** 2;    # -> 1764  
print $b * 2;    # -> 6.282  
print $c + 1;    # -> 2  
print $a + 1;    # -> 1  
print ++$a;      # -> aab
```