profi.com®
we make IT work

CASE STUDY | SICHERES ONLINEBANKING

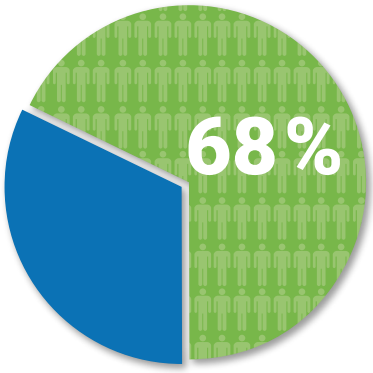
Spätestens seit den Enthüllungen von Edward Snowden ist IT-Sicherheit ein nicht nur in den Medien intensiv diskutiertes Thema. Auch die Bundesregierung will mit dem in Kraft getretenen IT-Sicherheitsgesetz eine Verbesserung des Schutzes der Wirtschaft und der Daten der Bürger erreichen.

Dabei führen nicht nur die Aktivitäten von Geheimdiensten, sondern in erster Linie die mit zunehmender Professionalisierung ausgeführten Angriffe Krimineller zu einer ernstzunehmenden Verunsicherung der Verbraucher und Kunden.

Speziell im Fall von Bankgeschäften und allgemein bei Bezahlvorgängen aller Art drohen sowohl den Kunden und Nutzern, als auch den Anbietern wie Banken, Webshops oder Dienstleistern durch Cyberangriffe direkte finanzielle Schäden. Darüber hinaus führen erfolgreiche Angriffe oft zu einer erheblichen Image-Schädigung des betroffenen Anbieters.

Der hier vorgestellte Projektbericht stellt keine allgemein gültige Handlungsanweisung dar, sondern basiert auf den im Projekt gesammelten Erfahrungen.

Wirksamer Schutz gegen vielfältige Bedrohungen



Gut zwei von drei Internetnutzern (68 Prozent) in Deutschland ab 14 Jahren setzen auf Online-Banking. Das entspricht 37 Millionen Bundesbürgern. Allein im Jahr 2014 haben Experten mehr als 145 Millionen Internetadressen identifiziert, über die Schadsoftware heruntergeladen werden konnte. Diese heimlichen Downloads, auch »Drive-by-Downloads« genannt, gehören aktuell zu den größten IT-Bedrohungen, da sich die Viren rasant verbreiten (Quelle: BITKOM).

Es ist daher inzwischen unverzichtbar und für Betreiber kritischer Infrastrukturen zum Teil zwingend vorgeschrieben, dass einerseits die Sicherheitsvorgaben für Aufbau, Konfiguration und Betrieb solcher Systeme eingehalten werden müssen und auch durch unabhängige Prüfungen nachzuweisen sind. Andererseits muss die Wirksamkeit der Sicherheitsmaßnahmen auch durch simulierte Angriffe, sogenannte Penetrationstests nachgewiesen werden.

In unserem Praxisbeispiel galt es, ein zentrales Bankensystem auf die Wirksamkeit und Robustheit seiner Sicherheitsmaßnahmen zu überprüfen. Bei dem System handelt es sich um eine klassische Webanwendung mit [3]-Schicht-Architektur, also einem Webfrontend, einer Anwendungsschicht und einer Datenbank, die untersucht werden sollte.

Bei der durchzuführenden Prüfung war nachzuweisen, dass es auch mit verschiedenen Angriffsmethoden nicht gelingt, die Sicherheitsmaßnahmen auszuhebeln und z. B. Zugriff auf die in der Datenbank gespeicherten Daten zu erlangen.

Basierend auf den Empfehlungen des OWASP[1] (Open Web Application Security Project) sollte die Anwendung auf folgende Schwachstellen untersucht werden.

1. **SQL Injection**
2. **Fehler in Authentifizierung und Session-Management**
3. **Cross-Site Scripting (XSS)**
4. **Unsichere direkte Objektreferenzen**
5. **Sicherheitsrelevante Fehlkonfiguration**
6. **Verlust der Vertraulichkeit sensibler Daten**
7. **Fehlerhafte Autorisierung auf Anwendungsebene**
8. **Cross-Site Request Forgery (CSRF)**
9. **Verwendung von Komponenten mit bekannten Schwachstellen**
10. **Ungeprüfte Um- und Weiterleitungen**

Im Projekt stellte sich heraus, dass die Hauptangriffspunkte Nummer 1 und 3 sind. Die restlichen Punkte auf obiger Liste waren nicht relevant hinsichtlich des Anwendungsbereichs oder rückten außerhalb des Geltungsbereichs.

(1) http://www.owasp.org

(2) https://www.owasp.org/images/4/42/OWASP_Top_10_2013_DE_Version_1_0.pdf

Intelligente Tool-Auswahl passend zu den Kundenanforderungen

Zunächst wurde eine Toolstudie durchgeführt, um die am Markt gängigen Werkzeuge für Penetrationstests auf ihre Anwendbarkeit auf die IT-Landschaft des Kunden zu prüfen. Aus dem großen Portfolio von Werkzeugen für Penetrationstests wurden folgende Tools ausgewählt:

- **XSS-me**
  (http://labs.securitycompass.com/exploit-me/xss-me)
- **SQL-Inject-me**
  (https://labs.securitycompass.com/exploit-me/sql-inject-me/)

Hinsichtlich der finalen Umsetzungen wurden die beiden Tools **XSS-me** und **SQL-Inject-me** genutzt, da diese die Anforderungen an die Sicherheitsrichtlinien des Penetrationstests hinreichend erfüllten.

Für die anderen beiden Tools bestand kein notwendiger Bedarf, jedoch könnten diese zu einem beliebig späteren Zeitpunkt hinzugezogen werden. Ein Kompendium für derzeit gängige SQL-Injection und XSS liefert OWASP. Dies bildete die Basis für unsere Cyberattacken und lieferte die Vorlage für die Angriffsszenarien.

Vorbereitung und Durchführung des Penetrationstests

In unserem Beispiel wird der Zahlungsverkehr per Webclient auf gängigen Browsern überwacht. Dahingehend erfolgte die Durchführung der verschiedenen Angriffe auf gängigen Webclients (Browser)[3].

Die aufgelisteten Tools, welche in vorigem Kapitel besprochen wurden, lassen sich als einfache Add-Ons für die Browser installieren. Abbildung 1 zeigt die beiden installierten Add-Ons auf dem Mozilla Firefox.

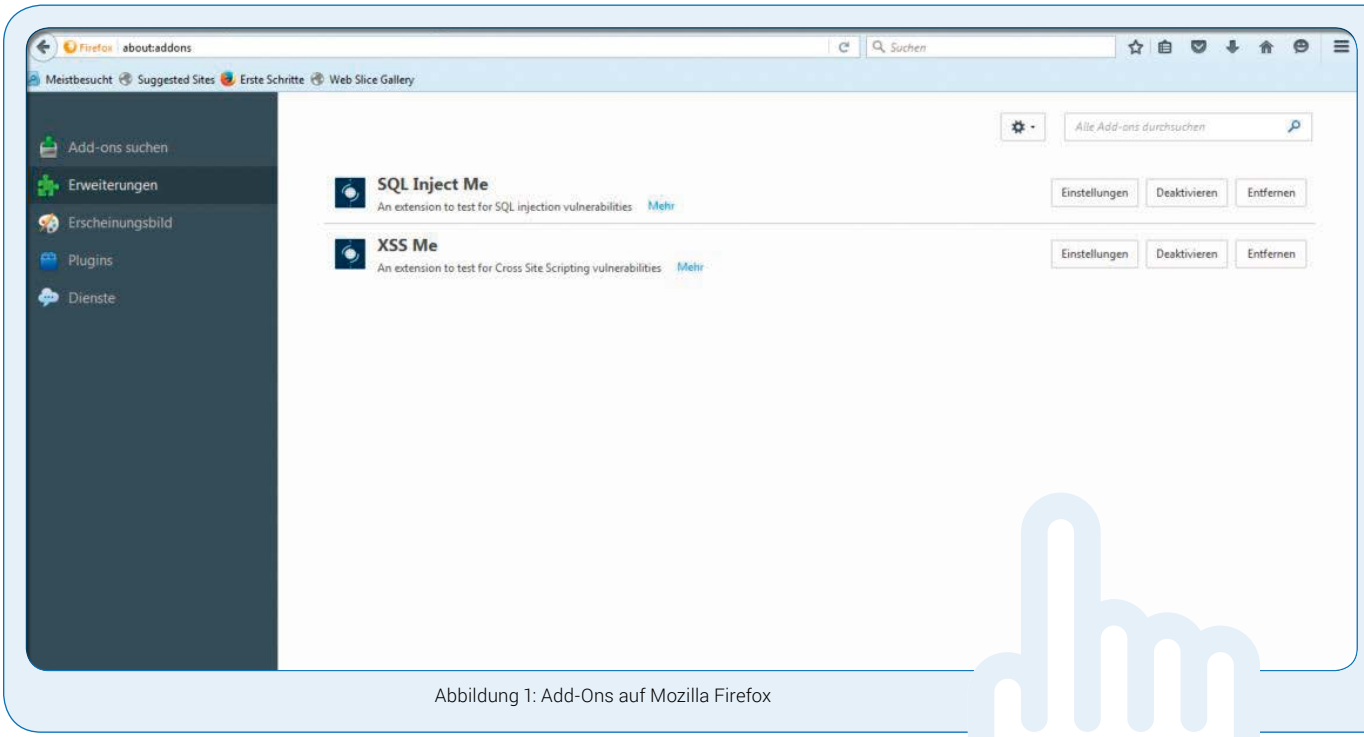


Abbildung 1: Add-Ons auf Mozilla Firefox

(3) Durchführung auf Firefox 38 ESR und Firefox 40

Unter Einstellungen ist es für beide Add-Ons möglich, die Injection Strings bzw. Optionen anzupassen, siehe Abbildung 2 und 3.

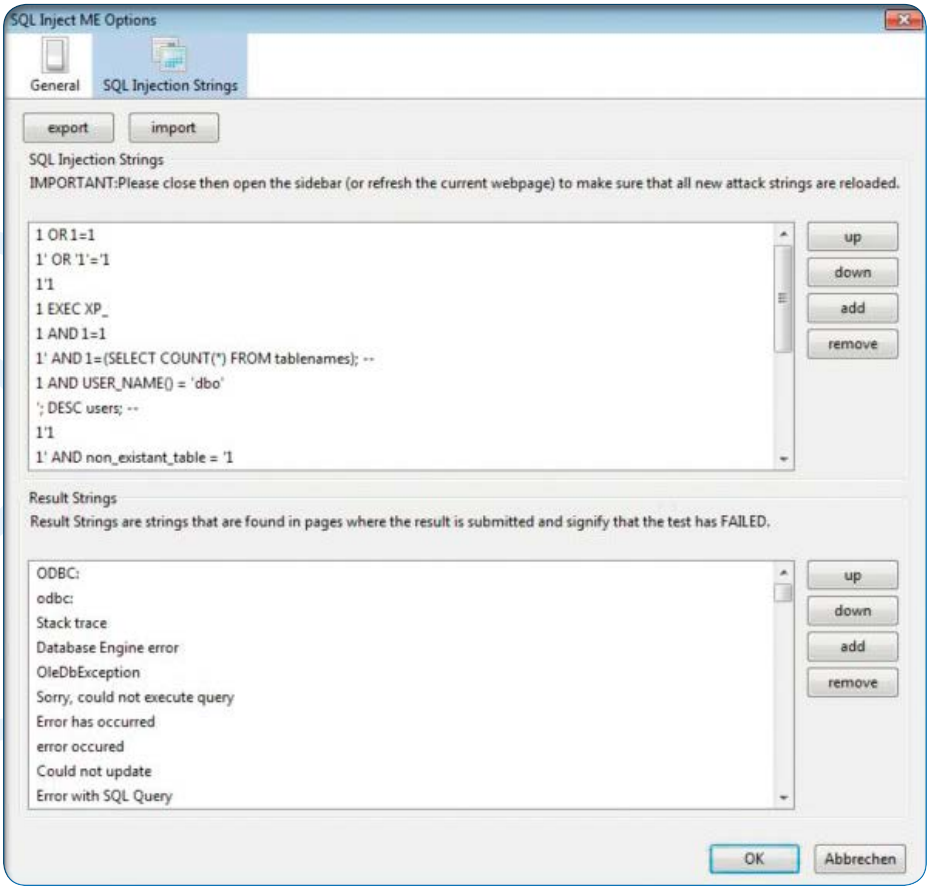


Abbildung 2: SQL Injection Strings (SQL Inject ME)

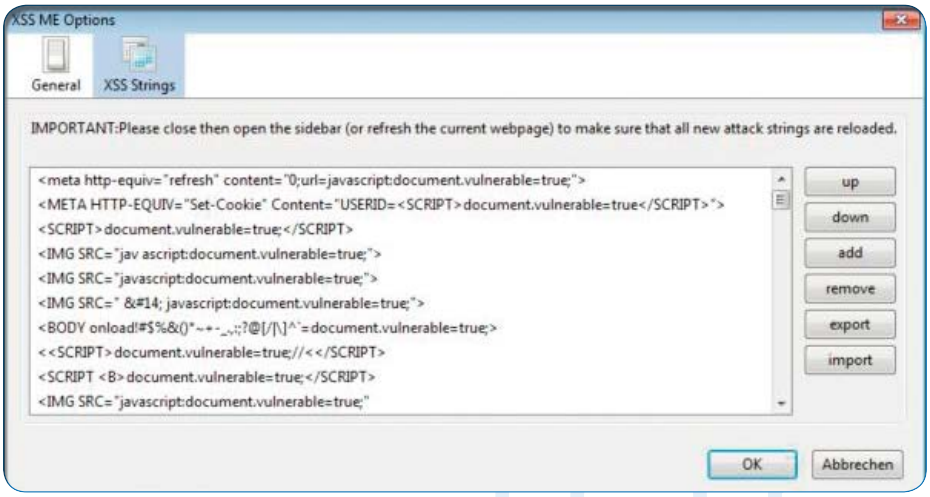


Abbildung 3: XSS Strings (XSS ME)

Die Durchführung der Tests geschah dabei folgendermaßen[4]:

1. **Öffnen des Browsers und Aufruf der Applikation**
2. **Öffnen der spezifischen Maske (URL) im Browser** (siehe Abbildung 4)

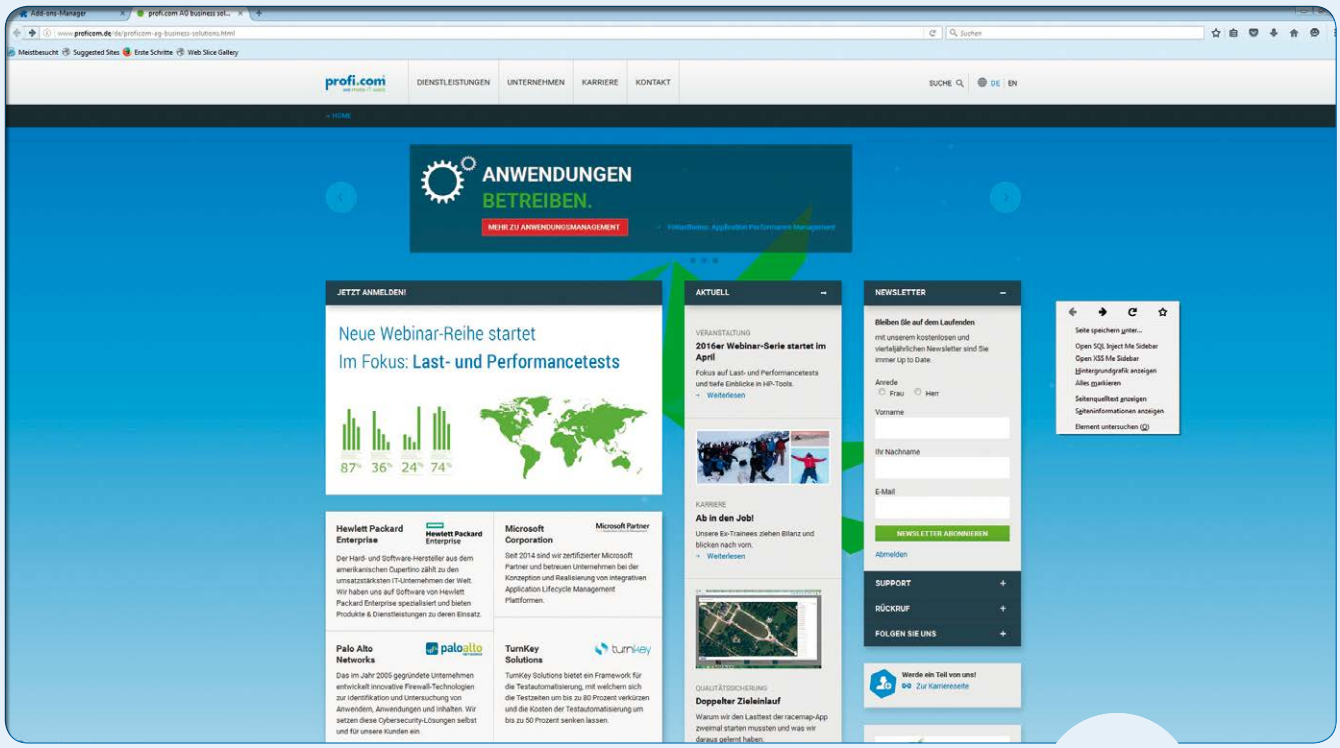


Abbildung 4: Start des Add-Ons im Browser

3. **Öffnen des Add-Ons im Browser**
4. **Auswählen der Injections (Angriffe) aus der definierten Liste des Add-Ons**

(4) Aus Datenschutzgründen wird das Beispiel exemplarisch auf http://www.proficom.de/ durchgeführt.

5 Auswählen der Objekte auf der Maske, auf denen die Injections angewendet werden
(siehe Abbildung 5 und 6)

- Eingabefelder (Login, Suchfelder …)
- Buttons
- Grafiken
- Tabellen uvm.

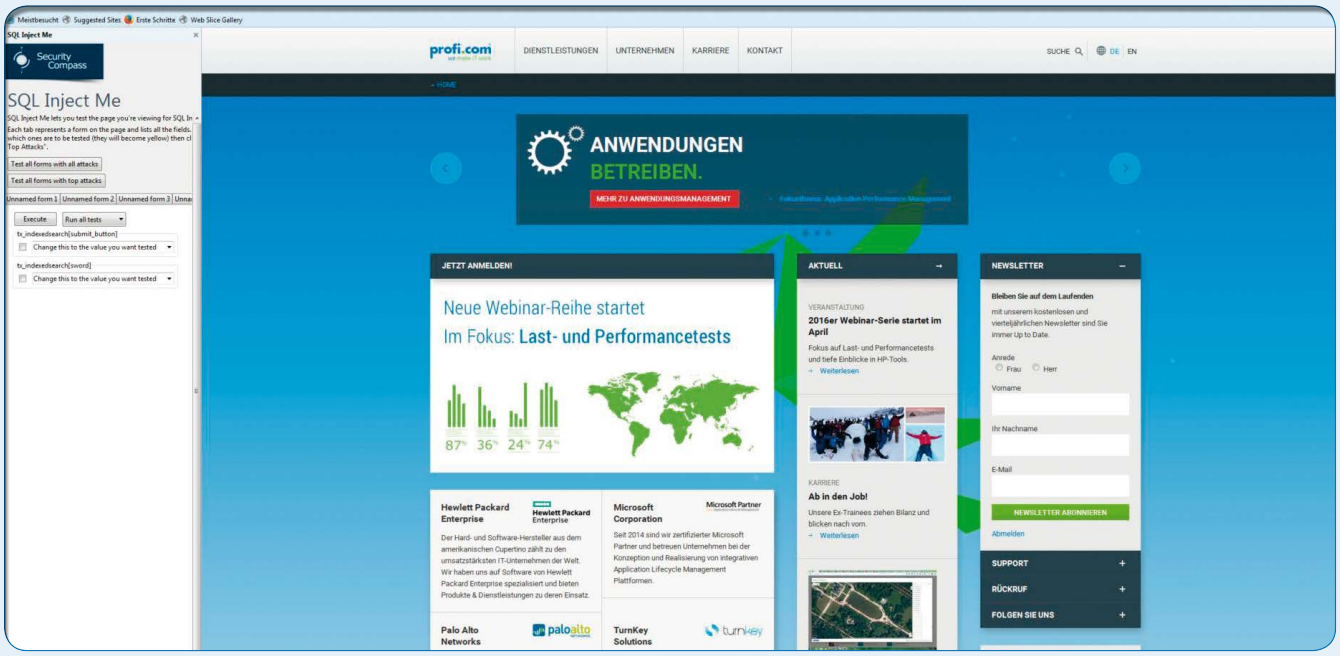


Abbildung 5: Auswahl der SQL-Injections und Ojekte auf der URL (SQL Inject ME)

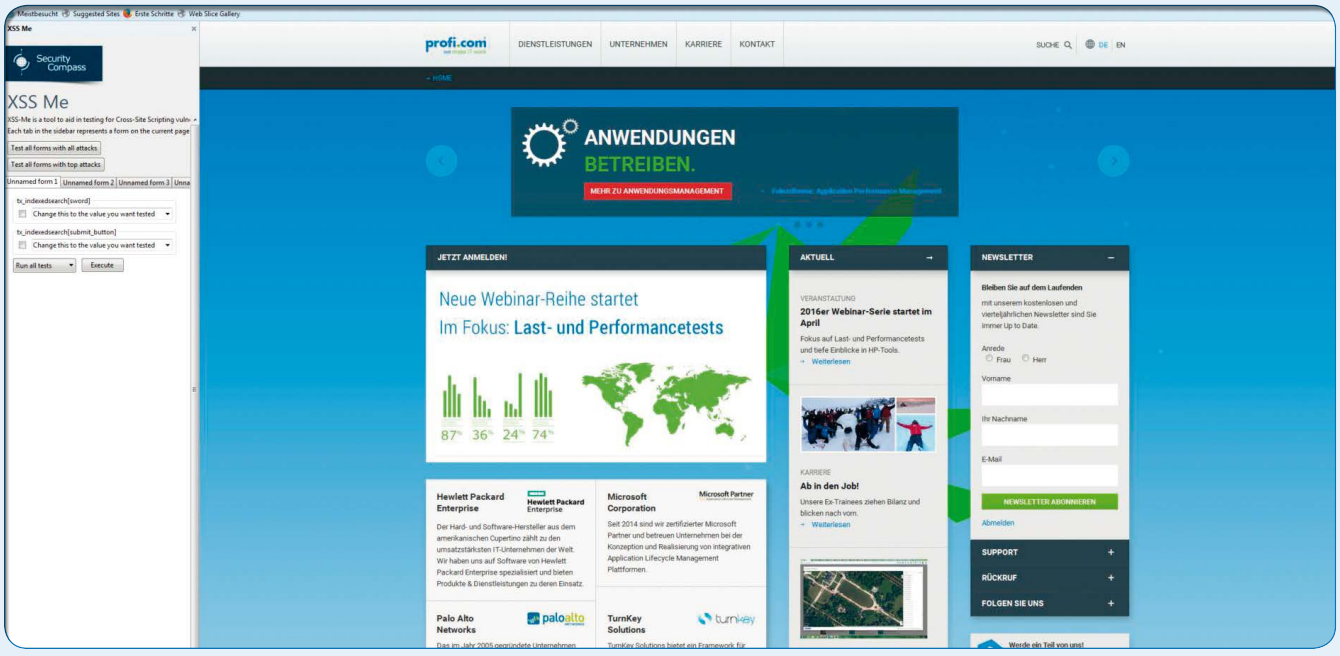


Abbildung 6: Auswahl der XSS-Attacken und Objekte auf der URL (XSS ME)

6 Durchführen der Injections bzw. XSS

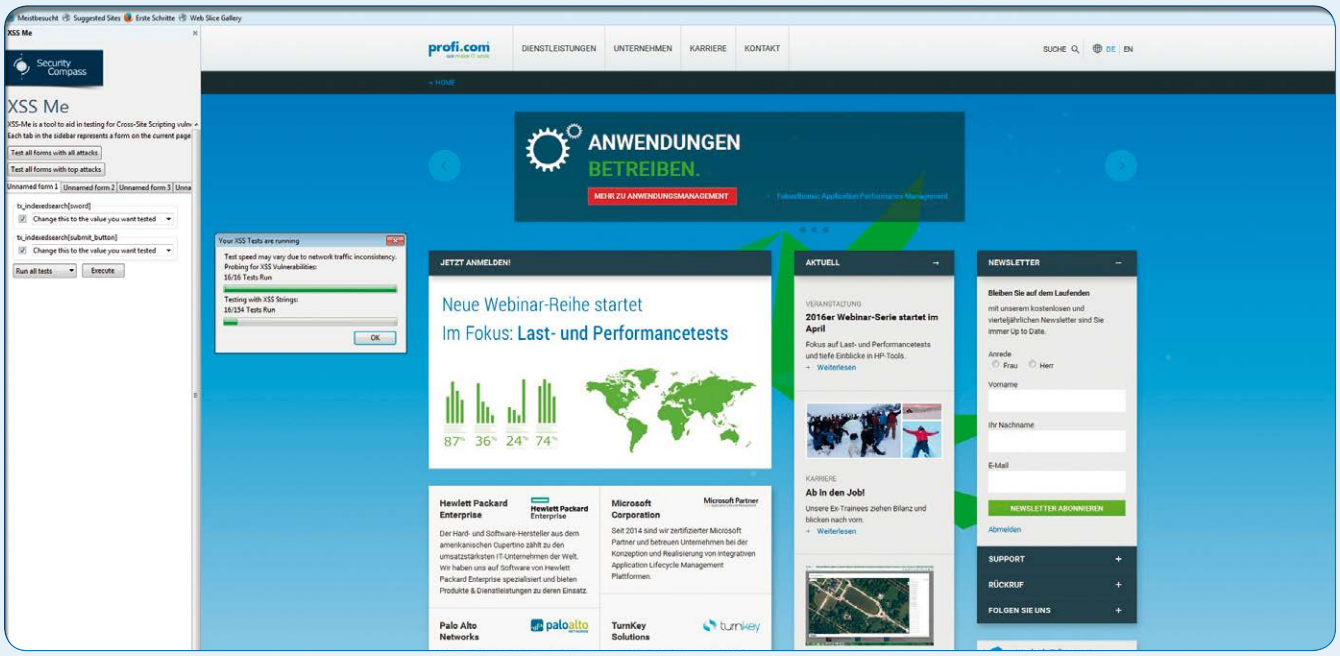


Abbildung 7: Durchführen eines XSS

7 Analyse der Resultate

Die Ergebnisanalyse erfolgt zum einen direkt im Browser (siehe Abbildung 8) oder auch im Editor mit den erstellten Ergebnisfiles (Tools -> XSS ME -> Options and click »Show passed results in final report«). Dies ist für SQL Inject Me analog.

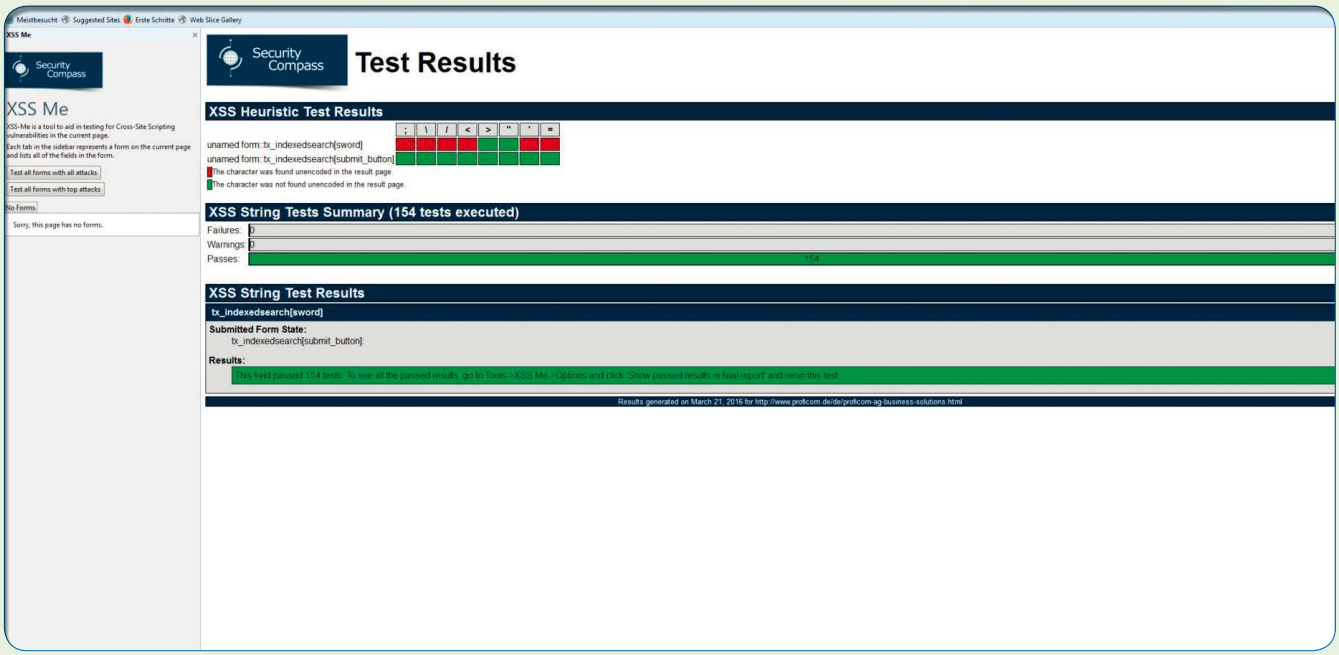


Abbildung 8: Exemplarisches Ergebnis für eine XSS-Attacke (XSS ME)

KOMPROMITTIERUNG MITTELS SQL INJECTION

Exemplarisch soll die Prüfung auf SQL-Injections[5] am bekannten Beispiel »1 = 1« erklärt werden.
Hierbei denken wir uns eine klassische Login-Maske, an der sich der Nutzer einloggen kann. Dazu werden im Allgemeinen die Daten »Username« und »Password« eingegeben, und anschliessend der »OK«-Button geklickt. Folgendes SQL-Statement könnte dabei in der Applikation enthalten sein, welches die Userinformation des Eingebenden ausliest.

```
statement = „SELECT * FROM users WHERE name ='" + userName + "';"
```

Für eine Eingabe mit validem Nutzer liefert diese Abfrage genau ein erwartetes Ergebnis, nämlich die Informationen zum abgefragten Nutzer. Sollte der Nutzer nicht existieren, wird kein Ergebnis zurückgegeben. Bei SQL-Injections versucht nun ein Angreifer die Eingangsdaten so zu manipulieren, dass er mit falschen bzw. nicht korrekten Zugangsdaten dennoch Zugang erhält. Im Beispiel geschieht dies, wenn die Nutzereingabe ungefiltert auf »Escape Characters« direkt in das SQL-Statement weitergegeben wird. Damit können potentiell gefährliche Eingaben getätigt werden, welche dem Eindringling viel mehr gestatten, als eigentlich vorgesehen. Beispielsweise mit dem Setzen der »userName«-Variable zu:

```
, OR ‚1'='1
```

Oder indem man den Rest der Abfrage einfach auskommentiert.

```
, , OR ‚1'='1' --

, OR ‚1'='1' ({

, OR ‚1'='1' /*
```

Das Resultat einer solchen Eingabe ist auf Datenbankebene dann:

```
SELECT * FROM users WHERE name = ‚' or ‚1'='1';

SELECT * FROM users WHERE name = ‚' or ‚1'='1' -- ‚;
```

Wird dieses Stück Code während einer Authentifizierung verwendet, so wird die WHERE-Klausel quasi außer Kraft gesetzt, da immer ein valider Username zurückgegeben und der Ausdruck »1 = 1« immer wahr ist!

KOMPROMITTIERUNG MITTELS CROSS SITE SCRIPTING (XSS)

Cross-Site-Scripting[6] ist eine Variante der HTML Injection. Cross-Site-Scripting tritt dann auf, wenn eine Webanwendung Daten annimmt, die von einem Nutzer stammen, und diese Daten dann an einen Browser weitersendet, ohne den Inhalt zu überprüfen. Damit ist es einem Angreifer möglich, auch Skripte indirekt an den Browser des Opfers zu senden und damit Schadcode auf der Seite des Clients auszuführen.

Es gibt drei grundlegende Arten von Cross-Site-Scripting-Angriffen:
reflektierte, persistente und DOM-basierte. Diese werden im Folgenden erläutert.

In den Beispielen wird zur Veranschaulichung der einfache JavaScript-Code **alert(„XSS“);** verwendet, der mithilfe des Script-Elements in ein HTML-Dokument eingebunden wird:

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

Dieser JavaScript-Code öffnet einen Alarm-Dialog mit dem Text »XSS« und einem »OK«-Button.

Reflektiert oder nicht-persistent

Das nicht-persistente (non-persistent) oder reflektierte (reflected) Cross-Site-Scripting ist ein Angriff, bei dem eine Benutzereingabe direkt vom Server wieder zurückgesendet wird. Enthält diese Eingabe Skriptcode, der vom Browser des Benutzers anschließend interpretiert wird, kann dort Schadcode ausgeführt werden.

Hierbei wird ausgenutzt, dass dynamisch generierte Webseiten ihren Inhalt oft an über URL (HTTP-GET-Methode) oder Formulare (HTTP-POST-Methode) übergebene Eingabewerte anpassen.

Nicht-persistent heißt dieser Typ, da der Schadcode nur temporär bei der jeweiligen Generierung der Webseite eingeschleust, nicht aber gespeichert wird. Wird die Seite danach ohne die manipulierte URL oder das manipulierte Formular erneut aufgerufen, ist der Schadcode nicht mehr enthalten.

Persistent oder beständig

Persistentes (persistent) oder beständiges (stored) Cross-Site-Scripting unterscheidet sich vom reflektierten XSS prinzipiell nur dadurch, dass der Schadcode auf dem Webserver gespeichert wird, wodurch er bei jeder Anfrage ausgeliefert wird. Dies ist bei jeder Webanwendung möglich, die Benutzereingaben serverseitig speichert und diese später wieder ausliefert, solange keine Prüfung der Benutzereingaben bzw. eine geeignete Kodierung der Ausgabe stattfindet.

DOM-basiert oder lokal

Diese dritte Art des Angriffs wird DOM-basiertes (Dom Injection) oder lokales (local) Cross-Site-Scripting genannt. Im Gegensatz zu den oben genannten gängigen XSS-Varianten ist hier die Webapplikation auf dem Server gar nicht beteiligt. Somit sind auch an sich statische HTML-Seiten mit JavaScript-Unterstützung anfällig für diesen Angriff.

Der Schadcode wird zur Ausführung direkt an ein clientseitiges Skript übergeben. Dies kann beispielsweise ein JavaScript sein, das einen URL-Argumentwert zur Ausgabe von Daten verwendet, ohne diesen ausreichend zu prüfen. Ein solcher Angriff bedarf des gezielten Aufrufs einer kompromittierten URL.

(6) https://de.wikipedia.org/wiki/Cross-Site-Scripting