Web Application SQL Fuzzer

Alex Dalton

May 15, 2014

CS 460

John Bambenek

1. Introduction
This project focused on developing a web application SQL injection fuzzer. SQL injection was decided to be the focus of the fuzzer because it is a relatively simple language to fuzz and SQL injection today still poses a threat to web applications because of poor input handling.

2. Motivation
Since taking CS411 last semester I was curious to how often basic security features such as input validation are ignored or overlooked when webapp functionality is the focus of a project release. I know in my CS411 group most security features were overlooked leaving our web application vulnerable to SQL injection and XSS. I wanted to do some research and come up with some statistics on this subject. Developing a SQL injection fuzzer would aid in that research.

3. Fuzzer Overview
The fuzzer framework is fairly simple. There are three main classes, the fuzzer class, the mutator class, and the interface class. The mutator class is used for mutating a fuzz vector in a different random ways depending on the mode the user chooses. The modes are as follows:

- 0 - uses a set of best mutations to generate permutations of the fuzz vectors
- 1 - uses entire set of mutations to generate permutations of the fuzz vectors
- 2 - uses only the add word mutation, where a word is any word or delimiter currently in the fuzz vector
- 3 - uses only the add N characters mutations, n of the current characters in the fuzz vector are added randomly
- 4 - uses only the add new words mutation, where a new word is any word that appears in the new words list but not in the fuzz vector
- 5 - uses only the add N words mutation, calls add word mutation N times
- 6 - uses only the add N new words mutation, calls add new word mutation N times
- 7 - uses only the add delimiter mutation, randomly adds an entry from the delimiter list

The mutator is the main component for generating the fuzzed inputs.

The fuzzer class is the top level class which uses an instance of the mutator class. The fuzzer operates by first reading from an input file, the fuzz vectors to be used in the given test, by default this file is the *fuzzvectors* file. A line in the fuzz vectors file can be commented out with a # at the beginning. After reading in the fuzz vectors the fuzzer will repeatedly send fuzzed SQL injection based strings to the interface for a given period of time. It sends fuzzed strings in the following pattern: first send only the fuzz vectors then continuously loop over the fuzz vectors list and send three mutations of the current vector. Repeat this part until time runs out. Time was used as the indicator for how long to fuzz because it was much better suited for applications of this fuzzer. This fuzzer will often times be ran overnight or over a weekend and a time indication is simpler than stating how many times to send data to the interface.

For each test an interface python class must be supplied via the command line (the -i option). The interface class defines how the fuzzer will interact with the desired interface. The interface class must contain a __init__() function (usually initializes variables and handles login into the website/saving cookies) and a send(self, message) function. This send function must be capable of delivering the fuzzed string to the interface being tested. This function then returns a boolean that indicates whether the fuzzed string triggered a response that indicates a SQL injection vulnerability. For now, triggering this involves seeing if anything in the response matches a list of predefined SQL error indicators. For example: “error in your SQL syntax” or “SQL Error”. This list could probably be expanded to include a lot more possible indicators, but I only used some basic ones.

program usage:

```
    ./run.py -h for help
    ./run.py -i interfacename -s secondsToRun
```

<u>4. Tests/Research</u>

To test the fuzzer I ran it against the DVWA low security SQL injection page. The test ran for about 10 seconds and if you look at the logs directory on the github you’ll see most strings that included an apostrophe triggered a SQL injection vulnerability indicator. This was mainly a proof of concept/basic functionality test.

The next tests I ran was against 5 randomly selected CS411 projects. I wanted to determine if care was put into providing simple input validation (things that would take 1 or 2 lines of code). The goal of testing these 5 projects was to try and get an idea of the degree of importance security played in the development process. Each test was ran from anywhere between 2 and 10 minutes. For some of the projects it was clear within the first few seconds that they were vulnerable to SQL injection.

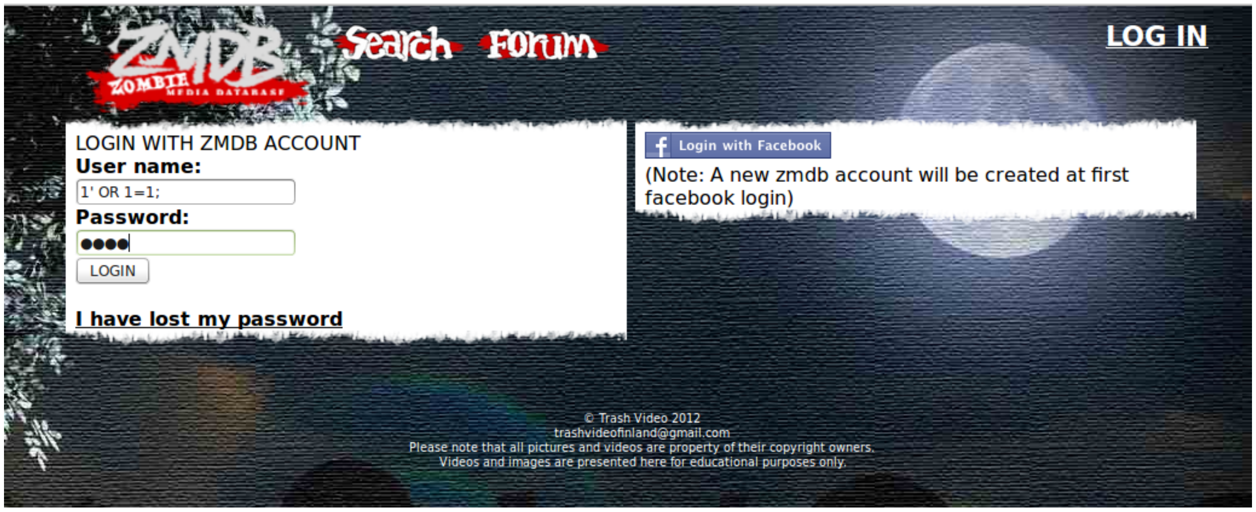
The third set of tests I ran was against web pages I found via google.com, non-big company sites like Facebook/Google/Twitter which I assume validate and check all their inputs and would be a waste of my time. The one interesting case I came across was a website called the Zombie Media Database. A small but “relatively” active site. Active in the sense that there are forum posts and new additions to the database as recently as May 10th.

<u>5. Results</u>

For the five CS411 projects I ran my fuzzer against, 4 of them gave strong a inclination that they were vulnerable to SQL injections. The interfaces commonly tested were login and search interfaces. These were likely the most vulnerable since search would definitely use the database backend and login may use it as well. A login interface being vulnerable to SQL injection also makes it a high threat vulnerability since this could give an attacker access to stored users and possibly passwords (which in the case of CS411 projects were probably stored in plaintext). I decided not to actually perform any SQL injection attacks on the the projects since they were large class projects that were potentially still being developed/graded. I understand that these projects focus mostly on functionality, but it’s frightening to think that this ⅘ statistic may apply to the entire class. Many of these vulnerabilities could be fixed with one or two lines of code to check user input. So maybe people didn't really care about security when developing their projects or maybe not enough emphasis was stressed on tying security into the development process.

The interesting site I tested my fuzzer against that I scrounged for on google was http://www.zmdb.org. From looking at the site you could tell it was likely made by an amateur web developer so it was an ideal site to test my fuzzer against. I ran the fuzzer for an hour on the search interface and login interface. The results indicated the site didn't validate input and was vulnerable to SQL injection on both the login and search interfaces.

After this indication I decided to mess around a little and see what I could do. I first attempted to inject SQL commands via the login interface and used the returned SQL errors as a guide. After several attempts I finally had something interesting happen. I input a simple SQL command that would always validate to true and put random keystrokes into the password field.



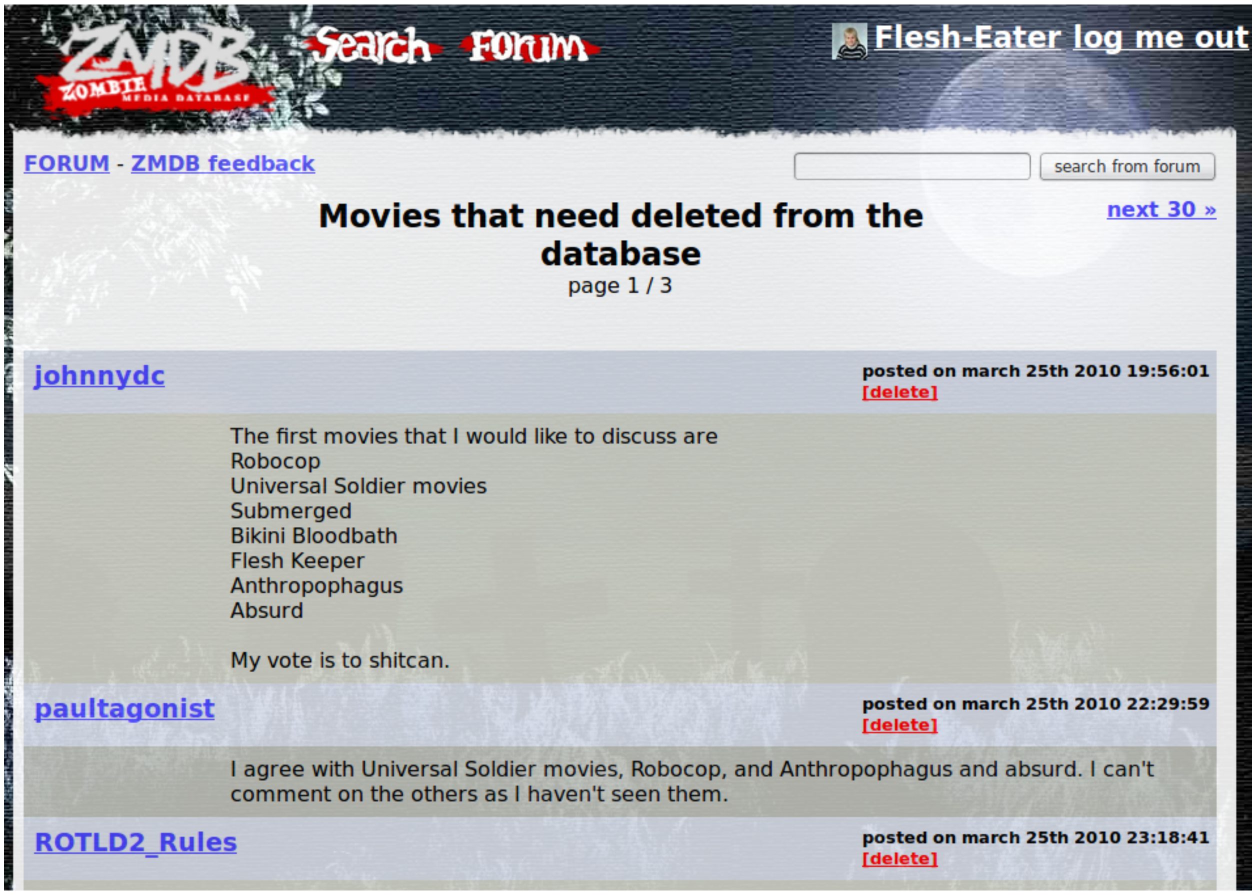
LOGIN WITH ZMDB ACCOUNT
User name: 1' OR 1=1;
Password: ●●●●

(Note: A new zmdb account will be created at first facebook login)

© Trash Video 2012
trashvideofinland@gmail.com
Please note that all pictures and videos are property of their copyright owners.
Videos and images are presented here for educational purposes only.

Figure 1. Zombie Database Login Page

After that I was logged in as some default user, which ended up being the admin of the site appropriately named "Flesh-Eater" (heh). This granted me access to delete any forum post or delete/edit any movie entry, and probably other stuff. I didn't poke around too much or mess with anything though, this "Flesh-Eater" guy and his minions seemed pretty passionate about zombies and I wouldn't want to upset them. I confirmed that the aforementioned functionality was unique to Flesh-Eater by logging in on my own account. I also tested to see if I could login to the user Flesh-Eater with the random password I used before or any other password/blank password and these attempts failed. This indicated that this security breach was indeed due to improperly formatting input used in a SQL query.



FORUM - ZMDB feedback

Movies that need deleted from the database
page 1 / 3

johnnydc — posted on march 25th 2010 19:56:01

The first movies that I would like to discuss are
Robocop
Universal Soldier movies
Submerged
Bikini Bloodbath
Flesh Keeper
Anthropophagus
Absurd

My vote is to shitcan.

paultagonist — posted on march 25th 2010 22:29:59

I agree with Universal Soldier movies, Robocop, and Anthropophagus and absurd. I can't comment on the others as I haven't seen them.

ROTLD2_Rules — posted on march 25th 2010 23:18:41

Figure 2. Zombie Database Forum with Flesh-Eater User Access

To see some of the results see the log directory in the git repo. I didn't include some of the longer hour long tests since it just logs the tested string and a True or False value to indicate whether a possible SQL injection vulnerability exists or not.

6. Conclusion

In the end SQL injection is still a real threat because people are mostly lazy, or uninformed which was probably Flesh-Eater's situation. It's interesting to see that a majority of the tested CS411 projects were vulnerable to SQL injection attacks as well, despite the class predominantly being about databases (database security being one of the main sections of the course). I think this fact shows that the pressures of deadlines and importance placed on functionality lead to ignoring security in the design process. This SQL fuzzer tool was fun and relatively easy to develop. Most the work came from investigating other web applications and using the tool to see which interfaces were vulnerable. This report only highlights the more interesting tests, to no surprise a lot of sites actually do validate and check user input! I'm not really sure what else to put in this report, hopefully it isn't too lengthy as it is now. Have a great summer, and maybe a great life too. :D