Security Assignment

Bradley J. Mackey

for 4th December 2017

1 Vulnerability

Path traversal vulnerability on the FileServer [`http://localhost:8080`].

1.1 Exploit

A malicious user is able to access any file on the entire server (including files not owned by `user` - from `root` and the `backup` user) by traversing up and down to any file, by using url encoding in the link.

1.2 Mitigation

Improved **url input sanitation** on the FileServer would prevent url encoding that allows such traversal. This could be achieved by only allowing alpha-numeric characters to be entered in a url, followed by a dot and then a short file extension. If the url is detected not to match this pattern, the request can be rejected.

2 Vulnerability

SQL injection vulnerability on 'shop' web server [`http://localhost:80`].

2.1 Exploit

From the 'login' page of the shop system, a maliciously crafted username can allow access to other user accounts on the database without a password (e.g. when username is "`' OR 1=1 -- `" [**no password**], the system allows the login, and allows us to view the credits of some user, which in the case of the demo data is `sally`). This happens because the malicious username modifies the query to return all users in the database (as '`1=1`' always evaluates to `true` and '`-- `' comments out the remainder of the query), then the first row of the query is selected by the PHP script and we can see that first user's credits. We cannot perform other queries though due to the fact that the PHP method `mysql_query()` only supports a single query at a time.

2.2 Mitigation

Improved input sanitation, by **escaping** user input through prepared statements, which will prevent user data being interpreted as an SQL query. Stricter input validation could also be applied to the username and password fields, such as only allowing alpha-numeric characters (and maybe *some* symbols). In addition, the PHP script used to handle login could be modified to ensure that only

one result is returned, otherwise display an error. It could also **double check** that the query's returned username and password matches the username and password that was entered.

3 Vulnerability

XSS vulnerability on 'shop' web server [`http://localhost:80`].

3.1 Exploit

If a user enters a JavaScript snippet for a username (or password) when signing up, the browser interprets this as code rather than data, meaning arbitrary code could execute in the browser of anyone that views this user's username. This could, for example, perform a malicious act such as cookie theft on another client's computer.

3.2 Mitigation

This can be mitigated by **improved input sanitisation**, such as only allowing a short alphanumeric username and only allowing certain characters within the password. **Escaping** user input will also prevent the user data being interpreted as code.

4 Vulnerability

`user` is given unnecessary `root` privilege.

4.1 Exploit

In order to backup the `shop` database to the `backup` account using the `mysqldump` and the FileServer, `user` is granted `root` privilege in order to have access to the database. This is not necessary because of the fact that `user` already has full permission to read, write and '`mysqldump`' the database without this `root` permission. With these heightened permissions (and the `root` password located in `user/outputDB.c`), an attacker would have `root` control over the computer even if they were *only* able to access `user/` (via the FileServer or just logging in as `user`). In addition, `user` could unintentionally perform damage to the system (if they were to run an incorrect command) without challenge, because the level of privilege for `user` is too high.

4.2 Mitigation

The reason `user` thinks they require `root` permission is because in the `mysqldump` command in `user/outputDB.c`, they supply the `root` password as a parameter (`-pletmein`), also exposing this unnecessarily. If `user` tried to run this command without passing the `root` password in, the command would work without this escalated privilege.

5 Vulnerability

`root` has a very weak password.

5.1 Exploit

The `root` user has a password of '`letmein`' (discovered in the previous vulnerability) which is an extraordinarily weak password for the user with the most power on the computer, and given access to this account, an attacker could perform almost any function they wished (certainly more than logging into another user would give them).

5.2 Mitigation

Change the password for `root` to be a strong alphanumeric string, at least 9 characters in length, making use of upper and lowercase letters, numbers and symbols.

6 Vulnerability

Passwords stored in plaintext in SQL database for 'shop'.

6.1 Exploit

If a malicious user (including a malicious database administrator) gained access to the database, they are able to view a list of usernames and passwords for the shop in plaintext, inside the `users` table in the `shop` database. This would mean they could impersonate any user in the database and spend their credits. As users typically use the same password across multiple websites, this could also be used to hack into other accounts owned by that user.

6.2 Mitigation

Instead of storing plaintext passwords in the database, storing a **salted hash** for each user's password would mean that even if the database was compromised, user's accounts will not be. This is because the salted hash is only used to *check* if a user's password is correct by using a one-way hash function. The additional layer of salt (that should be a randomised string for each user) will ensure that even if 2 user's passwords are the same, they will not have the same hash and also means using password cracking facilities such as rainbow tables are much less feasible due to this additional layer of randomness.

Of course, users should also use a strong password (which it appears the current users of the shop do not have) in order to avoid having their password easily brute-forced or dictionary brute-forced. This could be made mandatory when a user signs up for an account, by requiring that passwords are of at least a certain length and use a mixture of letters, numbers and symbols.

7 Vulnerability

The shop SQL database is backed up unencrypted.

7.1 Exploit

When the shop's database is backed up, the entire contents of the database is sent unencrypted through a socket to some receiving server (the `backup` user in this case). While in development this is less of an issue (as the database file is only being sent to and received by `localhost`). Although, when this is deployed to production an attacker could perform a man-in-the-middle

attack to intercept of the database while it is being sent from the server to the backup server. This would give them access to the entire database and all of the (potentially sensitive) data.

7.2 Mitigation

Encrypt the database before transmission and then decrypt on arrival to the backup location, perhaps using the AES-256 encryption standard with a strong password (see vulnerability #5). This will prevent unauthorised viewing of the contents of the database.

8 Vulnerability

Source code placed alongside complied code throughout box (by `user`'s own admission, `user/README.txt`).

8.1 Exploit

Placing source code alongside compiled could allow an attacker to easily reverse engineer any the systems on the box, for example the FileServer. This is because if they gain access to the system, they would also be able to view the source code to see how a given service on the box works (they could even use the FileServer in this case to read these source files). It is then trivial for an attacker to reverse engineer and exploit this service.

8.2 Mitigation

Always keep all source code away from production servers, so that the chance of an attacker getting hold of it is nil (at least make reverse engineering *a little* hard for them).