Security and Data Integrity Analysis

Ryan Evans, Max Morgan, Adit Suvarna

1. Privacy Analysis

As Mush vs Grump is a local single player game, there is not much personal information gathered or stored in the database; therefore, managing our users' privacy should be a relatively simple task. However, the primary piece of user data that needs to be secured is the user's password. Often, many people use the same password for multiple services; therefore, restricting access of passwords stored in our database should be of primary concern.

A user will only have access to viewing and changing their own password and cannot modify or view any other user's password in any way. A user can only view or modify their password once they have successfully logged into their account with the correct username and password.

When a user types in their username and password in the Java front end it will be checked for any special characters, such as semicolons or double dashes (to avoid SQL Injection). After the check has passed, the username and password are salted, then encrypted before they are sent to the database through a stored procedure. The database will then decrypt the username and password and check it against the table, returning if it is a valid combination with the username or not.

All passwords on the database will be encrypted as well. This provides a safety net against any kind of unforeseen hacking.

2. Security Analysis

No client-side users will be given access to the database. Users should not be able to modify the database in any regard through their own queries, they should only be able to execute the given stored procedures through the interface provided. As this is a game, users should not be able to cheat or "hack" the game whatsoever. Thus, inputs should be sanitized, to prevent SQL Injection. We have created a method on the client side (in Java) which checks every potential input string for any semicolons (;), double-dashes (--), or apostrophes (‘), to avoid SQL Injection attacks. One potential

avenue a user could use to perform a SQL Injection, and therefore breach security, is when they are entering the username and password. At this stage in development, this is one of the few times in the game where the user types in text that will be used directly in a SQL statement in the database, and so is a prime target for a SQL Injection attack. However, we hope that the Java method that we have written (described above) will be able to detect any sort of suspicious behavior before it reaches the server. There are a few other locations (at this point in development) where it appears that a user could attempt to perform SQL Injection: when entering a name for their character, and when changing their password (and both are sent through our Java sanitizing method).

The game will utilize a low privilege SQLServer account named “mush” to access the database “MushVsGrump”. The actions allowed for “mush” on MushVsGrump is limited in order to prevent client-side code from utilizing any unnecessary sort of privileges, such as delete and select * statements (in other words, stored procedures must be used instead).

To prevent rogue users from guessing passwords based on precomputed SHA-1 encryptions, we will append a salt to each password before hashing the password, for additional security. Furthermore, we will make each salt unique, so that if one password is compromised, an attacker who might retrieve the salt we used would not be able to utilize that salt in another attack.

3. Entity Integrity Analysis

a. For the Character table, ChID must be a unique int value, as it is the primary key. ChName is a varChar with a maximum size of 20 bytes (characters), and must also be unique. ChName is also the clustering index for the primary index of the Character table. Base_HP has no restrictions besides being greater than zero.

b. For the Has table, both columns (attributes) form the primary key, and are both foreign keys that reference primary keys in other tables.

c. For the HasType table, both columns (attributes) form the primary key, and are both foreign keys that reference primary keys in other tables.

d. For the Healing_Item table, ItID is the primary key, which is also a foreign key that references the Item table. HP_Healed is a float value (that will represent the percentage of the maximum possible hp to heal) that may not be null.

e. ItID in Inventory is the only column, so it must be the primary key, and therefore may not be null and each value must be unique. Num represents the number of the given item in the inventory, and it may not be a null value, and must be greater than 0.

f. For the Item table, ItID must be a unique int value, as it is the primary key. ItName is a varchar of maximum size 40 bytes (characters). Desc is a varChar with a max size of 600 bytes (characters).

g. For the Level table, Level_num is the primary key, and is a tiny int, since we expect to have integer levels 1 through 100. Exp is a float value which represents the experience level that corresponds to each level.

h. For the level_multiplier table, Level_num is the primary key, and is also a foreign key that references the primary key of the Level table. Damage_multiplier and HP_multiplier are both floats which must be positive.

i. The User_Character table has ChID as the primary key, and ChID is also a foreign key which references Character. ChID must be greater than 0. InID is an int which may not be null.

j. The Weapon table's primary key is WeID, which is an int, and is auto-incremented with a seed of 1 and an increment of 1. WeName is varchar with a max size of 20 bytes (characters), and cannot be null. Attack_name is a varchar of max size 30 bytes (30 characters) that cannot be null. Base_Damage is a real number that must be greater than or equal to 0 and cannot be null.

k. For the Weapon_Changer row, the primary key is ItID, which is also a foreign key to the primary key in Item. Amt_Changed is a float to represent the amount that this item changes weapon damage, can cannot be null. Weapon_Type is a varchar of max size 12 bytes (characters) that cannot be null.

l. The WeaponType row has one attribute (column), which is WeaponType (which must be the primary key). WeaponType is a varchar of max size 12 bytes (characters) that will be used by weapon changing potions to refer to a certain classification of weapons.

4. Referential Integrity Analysis

Almost all of our foreign keys cascade on updates and deletes. The only exception is the foreign key relation between Level_Multiplier and Level. The reason for this is because we know the levels are going to be 1 through 100, so there is no reason to delete a level in the Level table, so the default constraint in Level_Multiplier prevents this from accidently occurring. We can still add levels past 100 if we desire to. The other foreign keys cascade on updates and deletes due to the dependencies of different tables upon each other. For example, weapon poisons (in the Weapon_Changer table) affect a certain type of weapon (represented in the Weapon_Type table). Each weapon is associated with a certain weapon type. So, if we deleted a certain weapon type, but did not delete the item changers which reference that weapon type, then those items would be useless, since the weapons they are trying to change would not be able to be found. If we remove an item, and that item is a healing item, then we want to remove the corresponding item id in the healing item table as well, so that there is no chance that that healing item (with no corresponding item id in the parent Item table) could show up anywhere in the game. If we remove an item from the database, we should definitely remove that item from the user's inventory as well, because it would be impossible for a person to use an item in their inventory that doesn't exist in the database.

5. Business Rule Integrity Analysis

There are not any business rule integrity constraints in our database. As an example, when a user uses an item, the client will execute a stored procedure which will remove that item from the database, but there won't be any table-to-table removals that occur.