

5.1: Security

Risk

It can be easy to both over and under consider the security of your website. New initiates tend to do everything possible to secure their code, which often results in overblown measures that can take considerable time to incorporate. This may only serve to protect something that may not require security in the first place. The dangers, then, are: getting tired of the complicated process, becoming complacent because nothing has happened, or neglecting to consider security in the first place. As a result, systems end up exposed.

Risk management is the practice of finding the sweet spot where the time and cost of implementing the measure is proportionate and acceptable to the perceived threat a compromise would create. Risk analysis is another of the many topics in this text that begs for much deeper study, and I would encourage everyone to read beyond what is here as it applies to everyone involved in a programming project.

In short, some things to consider about your system are its level of exposure, cost to acquire/replace its contents, importance of contents in relation to the company, and importance of the contents in relation to your clients. A number of industries are also bound to compliance measures based on certain types of information, or in order to achieve certain certifications.

Some questions to ask are things like:

1. Is this system only available on our network, or is it publically exposed?
2. Is this system only accessible through other security mechanisms like credentialing systems, SSL, etc.?
3. How much would it cost to replace the data the system holds?
4. Would it be possible to replace the data if it is lost?
5. Does the value of the data change if it is stolen or made publically available?
6. If the data is compromised, how would it affect our relationship with our customers?
7. Is any of the information required to be secured to a certain level?
8. Are we an industry (e.g., healthcare) that has to comply with a federal or other reporting guideline (e.g., HIPAA)

Answering these questions can be difficult, as you need to find quantifiable measures to questions that are better suited to qualitative analysis. An example of how to approach this is to determine risk values that address each topic. If data is irreplaceable, it might be an 8/10. If compromise poses no risk if stolen or exposed, it may be a 1/10. Using the same min/max range for each value allows us to add up our scores in order to get a summary value of the risk. This value can help you see the importance of your system better. From here, you can determine based on its value, particular values, or “red line” items that require specific minimums like federal requirements, exactly what security implementations you will need. You can also create estimates of what damages (data recovery costs, man hours to fix/replace the system, loss from lawsuits, etc.) might cost against that of implementing specific security measures as a cost/benefit ratio. For example, a mitigating effort of installing or developing a credentialing system might cost \$5,000 dollars, but if the data is exposed a lawsuit might entitle the plaintiff to \$80,000 dollars in damages. How much is spent to defend a system in relation to what the perceived losses might be will ultimately be determined by your executives and what the analysis reveals.

A common response to this issue is what if the system never gets attacked? Then the costs would be wasted! Alas, in many cases it can be hard to know or prove that risk management efforts actually stopped or prevented an attack. Time and money spent on security are difficult to defend, as their return on investment can rarely be proven. Was the system not compromised because of your efforts, or because no one tried? The typical reaction is to consider the survivability of the organization if the system is compromised or lost, and this is where strong metrics and analysis can save the day, the system, and maybe even your job.

Our example above of measuring each item on a fixed scale to determine a single threat value is just one mechanism. It is good for quick (relatively speaking) analysis and as a means of measuring multiple systems, or different possible versions of a system, against each other. More advanced techniques take more factors into consideration. For example, [the NIST SP 800-301](#) analysis includes a threat-likelihood-impact matrix where values are assigned to each element (the amount of likeliness of a given threat compared to its impact), repeated for each threat. Placing things into the matrix helps to quantify the scenario.

To perform a full risk analysis, you will want to consider all possible sources of damage to your system. These include intentionally malicious or accidental actions of users both in and outside of your company (just because a user is authorized to access data does not mean it is being used correctly or as intended), natural or man-made disasters that would affect your system (floods = water;

water + electronics = bad), or just general failure (power surge + electronics = bad). For the remainder here, we will only focus on the former as it directly applies to the topics we address in the rest of the text.

To delve deeper, look at some of the United States federal risk management initiatives.

- [National Institute of Standards and Technology \(NIST\)](#)²
- [Department of Homeland Security \(DHS\)](#)³
- [The United States Computer Emergency Readiness Team \(US-CERT\)](#)⁴

One parting note: No matter the size of your organization, even if you are the lone programmer, a risk analysis exercise should involve many minds, especially those of upper management. Their determinations will weigh heavily on your activities, and their exposure to these exercises will increase their awareness overall.

Now that we have an idea of how to discover and quantify risks, we can look at some basic methods of protecting ourselves.

PHP

One of the easiest ways for a malicious user to attempt to take advantage of our database, or gain unauthorized access, is to use our own scripting language against us. Let us consider a basic example. Say you have a form that takes a username and password to log in a user. Even if you employ an encrypted password with hashing, a malicious person could still take advantage of your form. Since your form action and field names are available to anyone who views your page's source code, they could read your form to generate the following URL: <http://yoursite.com/youractionpage.p...password=12345> or 1

Since they are using your form's variable names, your action page will, in this basic example, assume they are legitimate. The key element to note here is the apostrophe or 1 portion of the password field. If you are accepting user input without checking it first, like this:

```
1. mysqli_query("select * from users where username='$_GET[username]' and password='".md5($_GET[password])."'");
```

Your populated query would actually read as follows:

```
1. select * from users where username='FAKEUSER' and password='12345' or 1
```

When evaluated, the malicious user is automatically logged in because the “or 1” makes the logic statement true, since “or true” will always be true, regardless of the first half of the statement. Your malicious user is now logged into your site! If you did not follow the “least needed privileges” theory (we will look at this next), they would then have access to anything in the database that the account is allowed to access—potentially even other databases.

To protect against this, we can use [the sanitization](#)⁵ and [validation](#)⁶ features of PHP to ensure our users' responses are valid.

For example, if we ask a user to enter their email, we can first remove anything that should not be in an email address:

```
1. $email = filter_var($_GET['email'], FILTER_SANITIZE_EMAIL);
```

And then make sure it is still in proper format:

```
1. if(filter_var($email, FILTER_VALIDATE_EMAIL)){echo "OK!";}
2. else{ echo "Please re-enter email!"; }
```

The above code would be OK for a regular email address, but anything missing an @, or a top level domain like .com, would ask for it to be re-entered. Our sanitization for an email address will remove all characters except letters, digits and !#\$%&'*+./=?^_`{|}~@.[] . You typically want to sanitize before validating. If your sanitization removes something that was actually a desired part of the input, the validation would then fail.

MySQL

The source (for us, at least) of the risk: the data itself. We are storing our data in MySQL, which our web pages interface with in order to build our interactive experience. When we store user credentials in our database, we need to consider the fact that since our website communicates with the database, there is an inherent weakness. This is true whether or not our database is on the same physical system, or lives in the same operating system, as the site itself. One of the basic mechanisms to protecting your site is to prevent unauthorized access. Part of this can be achieved by obfuscating our users' passwords when we store this. To do this, we can employ a couple of tactics called hashing and salting.

Hashing is the process of passing the password through a mathematical algorithm that turns the password into a much longer string of characters. The algorithm is designed to be one-way, meaning a hash cannot be passed to an algorithm that reveals the original text. The algorithm will produce the same hash every time for a given password (although occasionally duplicates may occur, a fault called a “hash collision”). The result is stored in the database in place of the user’s actual password. Each time the user logs in, they type in their original password, your authentication system passes it through the algorithm, and you can compare the result to the stored value to make sure they match.

The use of a hash ensures that if someone is able to gain access to your database, they cannot simply copy the credentials and use them to their advantage, as the stored value is not the actual password they would use. A couple of popular methods that have been used to achieve this are MD5 and SHA-1, among others. I discourage the use of MD5, as its hashes are comparatively shorter than many others in use today, and enough data has been gathered about the algorithm that many password hashes can already be found just by using sites like md5decrypter.com.

Another issue of only relying on the original hashing algorithm is that other sites may be doing the same. If someothersite.com and your site are both using MD5 by itself, and someothersite.com is compromised, the attacker would have the usernames and passwords for their site. If any of your users also used someothersite.com, odds are some of them are probably using the same email address, username, and password. With all of this data, the attacker now has a set of usernames and passwords to try on your site. Since you are using the same hashing mechanism, the credentials used on both sites would work on yours as well.

The hashed value of the word password, for example, returns to us the value 5f4dcc3b5aa765d61d8327deb882cf99. If you take this value and paste it into md5decrypter.com, you will see our original text (password) given back to us.

Never fear, however, as we can defend against this with a salt. Salting is the process of adding something extra to your user’s password, like adding salt to food to make it better. This should ensure that even if a user has the same password on the other site, the one stored in our database is still different. This means a malicious person cannot use the encrypted password to get into our site.

If the salt for our user was #hsy5, our user’s password that we would hash could be password#hsy5 or #hsy5password or even pass#hsy5word. When we hash password#hsy5, we get 5b48480a7171f41d2bf52093f4850281. Now, if someone tries to use known passwords, their hash for password will not work on our site. You can use salts by either creating a salt for every password, or using one that is coded directly into your script.

Keep in mind that if you are using a single salt for all values, make sure it is not running in a script the user would have access to, like a linked JavaScript value, or would appear in your HTML as a hidden value in a form. This gives them the missing key to circumvent the extra protection you added. By only having the salt in one script, malicious agents would need to gain access to your web server as well as your database, especially if this script is segregated from your database’s home system. Storing a unique salt for each password means if one salt is discovered, not every account has been compromised. If the compromised account allows reading the salt table though, other accounts would eventually be accessible. Ultimately, using the best hashing algorithm available and both salting approaches would provide a high level of protection. How much you actually need to implement should be determined by completing a risk analysis.

Another strong self-protection method is to make sure your user accounts in your database are as locked down as possible. By this I mean abiding by a “least-needed security” approach. It is easy when developing to create a user account for your project with a simple password and wide open permissions to make development easier. Once you move your project live, however, you should take away all the permissions your users would not need and should not have. For example, your basic end users will likely not need the ability to alter or drop tables, so you should remove user access to those features. This means that if injection methods are used on your site, even if they succeed, the malicious threat still could not drop or alter your tables without also finding a vulnerability in your database.

To provide administrative users with an interface, you can set those pages to utilize a database user with higher credentials. The most sensitive operations (full database deletions, creating new administrators, etc.) should still be left in the hands of users who log into your database directly from a secured machine.

In MySQL, the full list of permissions elements are: *select, insert, update, delete, index, alter, create, drop*

Administrative privileges are: *create temp table, file, lock table, process, reload, replication, show dbs, shutdown super.*

We can quickly assign users the initial set of actions or the administrative set by using the keywords “usage” or “all” when assigning the account. Most sites can achieve everything necessary with the limited permissions of select, insert, update, and maybe

delete. Keep in mind although overall this is quite restrictive, without proper precautions injection attempts could still use insert or update to infiltrate your site.

Depending on the element, security can be restricted to the user based on the levels of global (your entire database server, i.e., every database), database, table, or column. This granularity allows you to ensure users can only change what they need to.

Our database *de jour* for this text also allows us to require SSL connections, providing greater security between the user (an admin or our webserver) and the database. MySQL also allows us to limit the number of updates over a span of time, as well as the number of concurrent connections and queries per hour. Typically, we would want these values very high to support the highest volume of end users possible, but you may have a use case where you know these numbers should be in a certain range. For example, if your company has 50 employees, and only 40 need your site, then you could cap your concurrent users to 40. Anything beyond that could be an indicator of database problems or hacking.

Types of Hashes

There are a number of strong hashing algorithms available. Some of these are considered out-dated and are easily to reverse, like DES. Others are still widely in use even though databases of known encrypted/unencrypted values for many passwords are freely available, like md5. Sites like md5decrypter.com are examples of this weakness. For this reason, you will likely want to favor the newer approaches available to you at the time. As new encryption algorithms are created, the encrypted string lengths become much larger, making the computation power required to reverse them too time consuming (if even possible) for the current hardware available.

Credentials

Never say never, but never give root. Root, or the default user on your MySQL server, has the highest level of permissions in the system. This means a PHP script that uses the root account to establish a connection can do anything, even drop all databases. Because of the damage an accidental or malicious command can have on your data, any MySQL user account that your web server utilizes to interact with your data should be as limited as possible. The vast majority of applications can get by with simple CRUD actions (that is create, read update, and delete). In most implementations, I would go one step further and not even allow deleting, replacing it instead with an “inactive” flag for records to hide them from the user as if they were deleted. Depending on the system, your web users might need additional features like temporary tables, and these should be allowed only as needed. This means if the attacker tries to perform administrative type actions, and you are not preventing them, they will still be blocked as the user they are acting as will not allow it.

JavaScript

We always (Yes, always—laziness breeds poor security in our world) want to take a look at anything a user gives us before we interact with it. This is for two reasons: first, the user may have made a mistake. Maybe they mistyped an email address, or left a required field blank. Or, perhaps, the user is a malicious person or script attempting to do something other than what we intend with access to our site. It might be using our forms to spam others, gain access to our data, or make unsolicited changes to our site. We already discussed this under PHP but we can attack the problem with JavaScript too.

Taking this into account, before we use anything a user gave us, we need to make sure (as much as possible) that it is safe data to interact with. Usually we want to do as much of this as possible on the user (or client) side, so they do not need to click submit and wait for a response from the server to find out something is not quite right. To do this, we can use client-side scripting like JavaScript to make sure things are OK as we progress. As fields are changed, JavaScript can look at the content and make sure addresses are formatted correctly, required fields are filled in, etc. Coloring, highlighting, or providing messages to the user when problems occur. We can achieve this easily by tapping into jQuery’s validation library:

```
1. <script src="/lib/js/jquery.validate.js"></script>
2. <script>
3. $(document).ready(function(){ $("#commentForm").validate(
4.  cname : { required : true, minlength: 2 }
5. );}
6. );
7. </script>
```

This example would execute the validation once the form is loaded, showing that the cname field is required and the minimum length is two characters. Not only can jQuery help us display these requirements on the form itself, we can call the validator as

fields are changed and/or when the form is submitted before leaving the page to enforce the rules we provided.

In terms of user experience, this is typically done in real time. As soon as a user leaves a field, the script makes sure it is OK, and provides confirmation of the fact (typically a green highlighting or “OK!” type of marker) or by not marking the field as bad (typically red, or prompting the user to re-enter the field).

Once the form is completed, JavaScript should ensure that the user’s submission will be good on the first try (at least content-wise—we cannot confirm things like a username and password without talking to the server). This accounts for our number one concern: mistakes from the user. Even though we checked the submission, we want to repeat this process on the server-side in more depth. If the user is malicious they may be circumventing our page, or the user may have JavaScript disabled.

The server-side script should take into account the nefarious user. If someone tried to subvert our form, JavaScript probably caught it. If, however, we are using GET or they use a script to send data directly to our action page from our form (which they can easily find in our page source) then they can get around our JavaScript.

Execution Functions

Both PHP and JavaScript support features that allow the user to access and run other programs or scripts on the web server or local system. This can be useful when you want to interface with another application or system that the language does not have the ability to communicate with directly, but it exposes a huge security risk. Anything passed to these functions will be executed as if that user was sitting at the command prompt of your web server. The implications here are fairly obvious, as anything your server’s “web user” account has permission for would be allowed. If you are passing a variable into the execute function, you have created a path directly to the heart of your system.

The best bet is to avoid using these entirely unless absolutely necessary. If you must, ensure that variables are not passed to the function if at all possible to prevent injection. Finally, if all else fails, sanitize and validate anything passed, limit your web server user role as much as possible, and keep your system as up to date as possible to deter hackers.

In PHP you will want to avoid the `exec()` function. JavaScript is a bit more removed, but some actions can create the ability, such as creating an ActiveX object:

1. `<script>`
2. `var wsh = new ActiveXObject('WScript.Shell');`
3. `wsh.run('notepad.exe');`
4. `</script>`

Segregated Systems

Your database server, ideally, would not live under the same operating system as your web server. This does not mean the same OS cannot be used on both systems, but that they are not residing on the same exact installation. This is important because if your system is exposed to, or faces, the Internet it is at a higher risk of compromise. Keeping the database within your network with a single controlled access point between the two means your data is not as compromised if the web server is.

Learn more

Keywords, search terms: Web server security, risk management, secure programming

76 Tips for Securing Your Server: <http://www.rackaid.com/resources/server-security-tips/>

Apache’s Security Tips: http://httpd.apache.org/docs/2.2/misc/security_tips.html

Symantec’s Tips for MySQL: <http://www.symantec.com/connect/articles/securing-mysql-step-step>

5.1: Security is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- 5.1: Security by [Michael Mendez](#) has no license indicated.