



Securing Applications

6.5.1

Securing the Data



It is no secret that security is a major issue in today's world. That applies to both data and applications. If one is secure and the other is not, both are vulnerable.

In this part, you will look at some of the issues involved in securing both your data and your application, starting with data.

Data is not just the heart of your application; it is said to be the new priceless resource; and it has got to be protected, both for practical and legal reasons. That applies whether data is being stored (also known as data at rest) or transferred from one server to another (also known as data in flight or in motion).

Best practices for storing encrypted data

When it comes to protecting data at rest, there are a few things you need to take into consideration.

Encrypting data

You have probably seen plenty of news stories about data breaches. These are typically a matter of individuals accessing data that is stored but not protected. In this context, this means that data is stored in such a way that not only can an individual gain access, but that when they do, the data is readily visible and usable.

Ideally, unauthorized persons or applications will never gain access to your systems, but obviously you cannot guarantee that. So when a person with bad intentions (who could just as easily be a disgruntled employee who has legitimate access) gets access to your database, you do not want them to see something like this:

| first_name | last_name | username | password | birthdate | state_id_number |
|------------|-----------|----------|------------|------------|-----------------|
| Casey | Jones | cjones | catflower | 04-07-1990 | DL228400184 |
| Taylor | Smith | smithee | filmbuffer | 08-25-1984 | DL88271004 |

Instead, you want them to see something more like this:

| first_name | last_name | username | password | birthdate | state_id_number |
|------------|-----------|----------|---------------------------------|-----------------|----------------------|
| Casey | Jones | cjones | Ds9982dan82@\$9asdjwh3shoe3i4s | 3278124586-234 | q343sdfgu2pontf8hq3g |
| Taylor | Smith | smithee | sdadjq34ds99n3trxS#TAVefgno3rvd | 29114385982-523 | 5u62p;wkf 34u5oeh-qt |

There are two methods for encrypting data: one-way encryption, and two-way encryption.

Two-way encryption is literally what it sounds like; you encrypt the data using a key, and then you can use that key (or a variation on it) to decrypt the data to get it back in plaintext. You would use this for information you would need to access in its original form, such as medical records or social security numbers.

One way encryption is simpler, in that you can easily create an encrypted value without necessarily using a specific key, but you cannot unencrypt it. You would use that for information you do not need to retrieve, just need to compare, such as passwords. For example, let's say you have a user, `bob`, who has a password of `munich`. You could store the data as:

| username | scrambled_password |
|----------|--------------------|
| Bob | @#\$\$SD@\$drw |

In this case, scrambling `munich` gives you `@#$$SD@$drw`, but there is no way to get "munich" back from that. To check Bob's password when he logs back in, you would need to do something like this:

```
select * from users where username = 'bob' and  
scrambled_password=scrambledversion('munich')
```

This would evaluate to:

```
select * from users where username = 'bob' and scrambled_password='@#$$SD@$drw'
```

Of course, the question then becomes, if you are going to encrypt your data using a key, where do you store the key safely? You have a number of different options, from specialized hardware (good, but difficult and expensive), to using a key management service such as Amazon Key Management Service (uses specialty hardware but is easier and less expensive), to storing it in the database itself (which is not best practice, has no specialty hardware or physical characteristics, and is vulnerable to attack).

Software vulnerabilities

When it comes to software vulnerabilities, you need to worry about two different types: your own, and everyone else's.

Most developers are not experts in security, so it is not uncommon to inadvertently code security vulnerabilities into your application. For this reason, there are a number of different code scanning tools, such as Bandit, Brakeman, and VisualCodeGrepper, that will scan your code looking for well-known issues. These issues may be embedded in code you have written yourself, or they may involve the use of other libraries.

These other libraries are how you end up with everyone else's vulnerabilities. Even software that has been in use for decades may have issues, such as the Heartbleed bug discovered in OpenSSL, the software that forms the basis of much of the internet. The software has been around since 1998, but the bug was introduced in 2012 and sat, undetected, for two years before it was found and patched.

Make sure that someone in your organization is responsible for keeping up with the latest vulnerabilities and patching them as appropriate.

Storing too much data

Remember that hackers cannot get what you do not store. For example, if you only need a credit card authorization code for recurring billing, there is no reason to store the entire credit card number. This is particularly important when it comes to personally identifying information such as social security numbers and birthdays, and other information that could be considered "private", such as a user's history.

Unless you need data for an essential function, do not store it.

Storing data in the cloud

Remember that when you store data in "the cloud" you are, by definition, storing it on someone else's computer. While in many cases a cloud vendor's security may be better than that of most enterprises, you still have the issue that those servers are completely outside of your control. You do not know which employees are accessing them, or even what happens to hard drives that are decommissioned. This is particularly true when using SSDs as storage, because the architectural structure of an SSD makes it difficult or impossible to truly wipe every sector. Make sure that your cloud data is encrypted or otherwise protected.

Roaming devices

In May of 2006, the United States Department of Veterans Affairs lost a laptop that contained a database of personal information on 26.5 million veterans and service members. The laptop was eventually recovered, but it is still a great example of why information must be stored in a secure way, particularly because the world's workforce is much more mobile now than it was in 2006.

In addition, apps are increasingly on devices that even more portable than laptops, such as your tablet and especially your mobile phone. They are simply easier to lose. These might not even be traditional apps such as databases, but apps targeted at the end user. Be sure you are not leaving your data vulnerable by encrypting it whenever possible.

Best practices for transporting data

Data is also vulnerable when it is being transmitted. In fact, it may be even more vulnerable because of the way the internet is designed, where packets pass through multiple servers (that may or may not belong to you) on their way to their final destination.

This structure makes your data vulnerable to “man in the middle” attacks, in which a server along the way can observe, steal, and even change the data as it goes by. To prevent these problems you can use:

- **SSH** - When connecting to your servers, always use a secure protocol such as SSH, or secure shell, rather than an insecure protocol such as Telnet. SSH provides for authentication and encryption of messages between the source and target machines, making it difficult or impossible to snoop on your actions.
- **TLS** - These days, the vast majority of requests to and from the browser use the `https://` protocol (rather than `http://`). This protocol was originally called SSL, or Secured Sockets Layer, but over the years it has been gradually replaced with TLS, or Transport Layer Security. TLS provides message authentication and stronger ciphers than its predecessor. Whenever possible you should be using TLS.
- **VPN** - Virtual Private Networks, are perhaps the most important means for protecting your application. A VPN makes it possible to keep all application-related traffic inside your network, even when working with remote employees. The remote employee connects to a VPN server, which then acts as a proxy and encrypts all traffic to and from the user.

Using a VPN has several benefits. First, traffic to and from the user is not vulnerable to snooping or manipulation, so nobody can use that connection to damage your application or network. Second, because the user is essentially inside the private network, you can restrict access to development and deployment resources, as well as resources that do not need to be accessible to end users, such as raw databases.

6.5.2

What is SQL Injection?



SQL injection is a code injection technique that is used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker). SQL injection must exploit a security vulnerability in an application's software. Two examples are when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements, or user input is not strongly typed and unexpectedly executed. SQL injection is mostly known as an attack vector for websites but can be used to attack any type of SQL database.

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

SQL in Web Pages

SQL injection is one of the most common web hacking techniques. It is the placement of malicious code in SQL statements, via web page input. It usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.

Look at the following example which creates a SELECT statement by adding a variable (`uid`) to a select string. The variable is fetched from user input using `request.args("uid")`:

```
uid = request.args("uid");  
str_sql = "SELECT * FROM Users WHERE UserId = " + uid;
```

One example is SQL Injection based on `1=1` is always true (in SQL-speak).

Take a look at the code to create an SQL statement to select user profile by UID , with a given UserProfile UID.

If there is not an input validator to prevent a user from entering "wrong" input, the user can enter input like this:

UID:

2019 OR 1=1

The output SQL statement will be like this:

```
SELECT * FROM UserProfiles WHERE UID = 2019 OR 1=1;
```

The SQL statement above is valid, but will return all rows from the "UserProfiles" table, because OR 1=1 is always TRUE.

What will happen if the "UserProfiles" table contains names, emails, addresses, and passwords?

The SQL statement will be like this:

```
SELECT UID, Address, Email, Name, Password FROM UserProfiles WHERE UID = 2019 or  
1=1;
```

A malware creator or hacker might get access to all user profiles in database, by simply typing **2019 OR 1=1** into the input field.

Another example is SQL Injection based on `""=""` is always true. Here is that example.

Username: user_a Password: pass_user_a

Example:

```
u_name = request.args("uid");  
u_pass = request.args("password");  
sql = 'SELECT * FROM UserProfiles WHERE Name =' + u_name + ' ' AND Pass =' +  
u_pass + ' ';
```

Here is the expected SQL statement:

```
SELECT * FROM UserProfiles WHERE Name ="user_a" AND Pass ="pass_user_a"
```

But the hacker can get access to user names and passwords in a database by simply inserting `" OR ""=""` into the user name or password text box:

User Name:

`" OR ""=""`

Password:

`" OR ""=""`

The output code will create a valid SQL statement at server side, like this:

Output:

```
SELECT * FROM UserProfiles WHERE Name ="" OR ""="" AND Pass ="" OR ""=""
```

The SQL above is valid and will return all rows from the "Users" table, because `OR ""=""` is always TRUE.

SQL Injection based on batched SQL statements

Most databases support batched SQL statements. A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "UserProfiles" table, then delete the "UserImages" table.

Example:

```
SELECT * FROM UserProfiles; DROP TABLE UserImages
```

Look at the following example:

```
uid = request.args("uid");  
strSQL = "SELECT * FROM UserProfiles WHERE UID = " + uid;
```

Now you input the UID:

```
2019; DROP TABLE UserImages
```

The SQL statement would look like this:

SQL:

```
SELECT * FROM Users WHERE UID = 2019; DROP TABLE UserImages;
```


Hopefully these examples influence you to design your data intake forms to avoid these common security hacks.

6.5.3

How to Detect and Prevent SQL Injection



SQL injection vulnerability exists because some developers do not care about data validation and security. There are tools that can help detect flaws and analyze code.

Open source tools

To make detecting a SQL injection attack easy, developers have created good detection engines. Some examples are SQLmap or SQLninja.

Source code analysis tools

Source code analysis tools, also referred to as Static Application Security Testing (SAST) Tools, are designed to analyze source code and/or compiled versions of code to help find security flaws.

These tools can automatically find flaws such as buffer overflows, SQL Injection flaws, and others.

You can detect and prevent SQL injection by using a database firewall. Database Firewalls are a type of Web Application Firewall that monitor databases to identify and protect against database specific attacks. These attacks mostly seek to access sensitive information stored in the databases.

Work with a database firewall

SQL injection filtering works in a way similar to email spam filters. Database firewalls detect SQL injections based on the number of invalid queries from a host, while there are OR and UNION blocks inside of the request, or others.

Use prepared statements

The use of prepared statements with variable binding (also known as parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

```
// Get customer's name from parameter
String          custname = request.getParameter("custname");
```

```
// Perform input validation to detect attacks
String          query = "SELECT account_balance FROM user_data WHERE user_name
= ? ";
PreparedStatement pStatement = connection.prepareStatement( query );
pStatement.setString( 1, custname);
ResultSet        results = pStatement.executeQuery();
```

Use Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely. This is the norm for most stored procedure languages.

They require the developer to just build SQL statements with parameters, which are automatically parameterized unless the developer does something largely out of the norm. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it cannot be avoided, the stored procedure must use input validation or proper escaping. This is to make sure that all user supplied input to the stored procedure cannot be used to inject SQL code into the dynamically generated query. Auditors should always look for uses of `sp_execute`, `execute` or `exec` within SQL Server stored procedures. Similar audit guidelines are necessary for similar functions for other vendors.

There are also several cases where stored procedures can increase risk. For example, on MS SQL server, you have three main default roles: `db_datareader`, `db_datawriter` and `db_owner`. Before stored procedures came into use, DBA's would give `db_datareader` or `db_datawriter` rights to the web service's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default. Some setups where the user management has been centralized, but is limited to those three roles, cause all web apps to run under `db_owner` rights so that stored procedures can work. That means that if a server is breached, the attacker has full rights to the database, where previously they might only have had read-access.

Whitelist Input Validation

Various parts of SQL queries are not legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

But if user parameter values are used for targeting different table names and column names, then the parameter values should be mapped to the legal/expected table or column names to make sure unvalidated user input does not end up in the query. Please note, this is a symptom of poor design and a full re-write should be considered.

Example of table name validation


```
String tableName;  
switch(PARAM):  
    case "Value1": tableName = "fooTable";  
                    break;  
    case "Value2": tableName = "barTable";  
                    break;  
    ...  
    default      : throw new InputValidationException("unexpected value provided"  
                                                         + " for table name");
```

The `tableName` can then be directly appended to the SQL query because it is now known to be one of the legal and expected values for a table name in this query. Keep in mind that generic table validation functions can lead to data loss because table names are used in queries where they are not expected.

For something simple like a sort order, it would be best if the user supplied input is converted to a boolean, and then that boolean is used to select the safe value to append to the query. This is a standard need in dynamic query creation.

For example:

```
public String someMethod(boolean sortOrder) {  
    String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC");`  
    ...  
}
```

Any user input can be converted to a non-string (such as a date, numeric, boolean, or enumerated type). If you convert before the input is appended to a query, or used to select a value to append to the query, this conversion ensures you can append to a query safely.

Input validation is also recommended as a secondary defense in ALL cases. More techniques on how to implement strong whitelist input validation is described in document [Open Web Application Security Project \(OWASP\) Input Validation Cheat Sheet](#).

Escaping all user-supplied input

This technique should only be used as a last resort, when none of the above are feasible. Input validation is probably a better choice as this methodology is frail compared to other defenses and we cannot guarantee it will prevent all SQL Injection in all situations.

This technique is to escape user input before putting it in a query. Its implementation is very database-specific. It is usually only recommended to retrofit legacy code when implementing input validation is not cost effective. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you.

The Escaping works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

There are some libraries and tools that can be used for Input Escaping. For example, OWASP Enterprise Security API or ESAPI is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications.

The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications and serve as a solid foundation for new development.

Additional defenses

Beyond adopting one of the four primary defenses, we also recommend adopting all of these additional defenses in order to provide defense in depth. These additional defenses can be:

Least privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables for which they need access. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.

If you adopt a policy where you use stored procedures everywhere, and do not allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Do not grant them any rights directly to the tables in the database.

SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. Minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

You should also minimize the privileges of the operating system account that the DBMS runs under. Do not run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default. Change the DBMS's OS account to something more appropriate, with restricted privileges.

Multiple database users

Web applications designers should avoid using the same owner/admin account in the web applications to connect to the database. Different DB users could be used for different web applications.

In general, each separate web application that requires access to the database could have a designated database user account that the web-app uses to connect to the DB. That way, the designer of the application can have detailed access control, thus reducing the privileges as much as possible. Each DB user will then have select access to what it needs only, and write-access as needed.

As an example, a login page requires read access to the username and password fields of a table, but no write access of any form (no insert, update, or delete). However, the sign-up page certainly requires

insert privilege to that table; this restriction can only be enforced if these web apps use different DB users to connect to the database.

SQL views

You can use SQL views to further increase the access detail by limiting read access to specific fields of a table or joins of tables. It could potentially have additional benefits. For example, suppose that the system is required to store the passwords of the users, instead of salted-hashed passwords. The designer could use views to compensate for this limitation; revoke all access to the table (from all database users except the owner or admin), and create a view that outputs the hash of the password field and not the field itself. Any SQL injection attack that succeeds in stealing DB information will be restricted to stealing the hash of the passwords (even a keyed hash), because no database user for any of the web applications has access to the table itself.

6.5.4

Secure the Application



Securing your application is a topic that merits its own book, but the next few pages illustrate some of the more common issues you should look for as part of your deployment process.

What is OWASP?

The Open Web Application Security Project (OWASP) is focused on providing education, tools, and other resources to help developers avoid some of the most common security problems in web-based applications. Resources provided by OWASP include:

- **Tools** - OWASP produces tools such as the OWASP Zed Attack Proxy (ZAP), which looks for vulnerabilities during development, OWASP Dependency Check, which looks for known vulnerabilities in your code, and OWASP DefectDojo, which streamlines the testing process.
- **Code projects** - OWASP produces the OWASP ModSecurity Core Rule Set (CRS), generic attack detection rules that can be used with web application firewalls, and OWASP CSRFGuard, which helps prevent Cross-Site Request Forgery (CSRF) attacks.
- **Documentation projects** - OWASP is perhaps best known for its documentation projects, which include the OWASP Application Security Verification Standard, the OWASP Top Ten, which describes the 10 most common security issues in web applications, and the OWASP Cheat Sheet Series, which explains how to mitigate those issues.

Let's look at some of the most common of those Top Ten issues.

SQL injection

You have learned about using data in your application, and how to protect it. One of the issues with using data in your application is that if you incorporate user interaction, you can create a potentially dangerous situation.

For example, would you ever want to execute a statement like this?

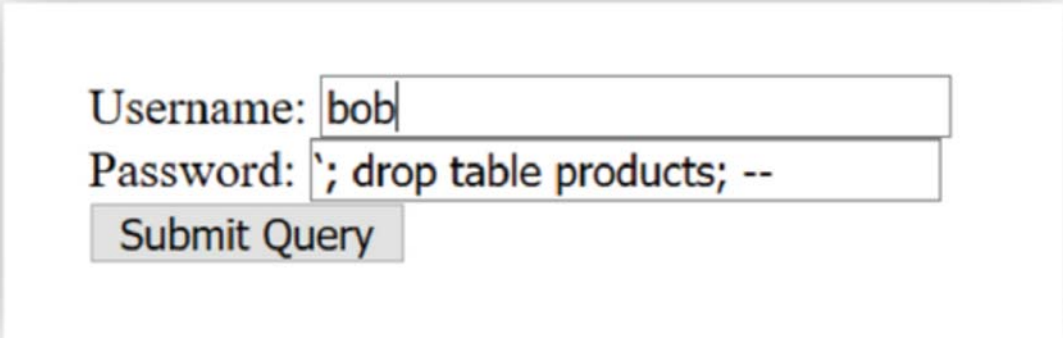
```
<p><code>select * from users where username = 'bob' and password = 'pass'; drop  
table products;</code></p>
```

Of course not. Because if you did, you would have deleted your products table. But if you are not careful, you could do exactly that. How? Consider this example. Let's say you have a form:



A login form with two input fields and a submit button. The first field is labeled "Username:" and the second is labeled "Password:". Below the fields is a button labeled "Submit Query".

The user enters the following data:



The login form with malicious input. The "Username:" field contains "bob" and the "Password:" field contains "; drop table products; --". The "Submit Query" button is still present.

That is an odd thing to enter, but follow the example through. If you have code that simply integrates what the user typed, you will get the equivalent of this:

```
username = "bob"  
userpass = "; drop table products; --"
```

```
sqlstatement = "select * from users where username='"+username+"' and  
password='"+userpass+"'"
```

If you make that substitution, you get:

```
sqlstatement = "select * from users where username='bob' and password=''; drop  
table products; --"
```

In this case, the hacker does not even have to enter a valid password for bob (or any username, for that matter); the important part is that the dangerous statement, drop table products, gets executed no matter what, and that double-dash (--) is a comment that prevents anything after it from causing an error and preventing the statement from running.

How do you prevent it? While it is tempting to think that you can simply "sanitize" the inputs by removing single quotes ('), that is a losing battle. Instead, prevent it from happening by using parameterized statements. How to achieve this is different for every language and database, but here is how to do it in Python:

```
with connection.cursor() as cursor:  
    cursor.execute("SELECT * FROM users WHERE username = %(username)s and  
password = %(userpass)s", {'username': request.args.get('username'), 'userpass':  
scrambled(request.args.get('userpass'))})  
    result = cursor.fetchone()
```

By doing it this way, you are creating these string variables, `username` and `userpass`, that are dynamically inserted into the string in such a way that the user cannot use them to create multiple statements.

One place this situation often appears is in regard to search, where by definition the user is entering what will become part of the database statement. Consider this code:

```
from flask import request  
from flask import render_template  
sample = Flask(__name__)  
DBHOST = 'NOT SET'  
def getdb(targethost = ''):  
    import MySQLdb  
    if targethost == '':  
        global DBHOST  
        targethost = DBHOST  
    return MySQLdb.connect(host=targethost,  
                           user="devnetstudent",  
                           passwd="pass",  
                           db="products")  
  
@sample.route("/")  
def main():  
    return render_template("index.html")  
  
@sample.route("/test")  
def test():
```

```

    return "You are calling me from "+request.remote_addr
@sample.route("/config")
def config():
    return render_template("config.html")
@sample.route("/get_config")
def get_config():
    global DBHOST
    return DBHOST
@sample.route("/config_action", methods=['GET', 'POST'])
def config_action():
    global DBHOST
    DBHOST = request.args.get('dbhost')
    return "Saved database host as "+DBHOST
@sample.route("/search")
def search():
    db = getdb()
    cur = db.cursor()
    search_term = request.args.get('search_term')
    cur.execute("select * from products where title like '%s'" % search_term)
    output = ""
    for row in cur.fetchall():
        output = output + str(row[0]) + " -- " + str(row[1]) + "<br />"
    db.close()
    return output
if __name__ == "__main__":
    sample.run(host="0.0.0.0", port=80)

```

How would you fix this code?

```

...
@sample.route("/search")
def search():
    db = getdb()
    cur = db.cursor()
    search_term = request.args.get('search_term')
    # Here you are ensuring that the search term is treated as a string value
    cur.execute("select * from products where title like %(search_term)s",
{'search_term': search_term})
    output = ""
    for row in cur.fetchall():
        output = output + str(row[0]) + " -- " + str(row[1]) + "<br />"
    db.close()
    return output
if __name__ == "__main__":
    sample.run(host="0.0.0.0", port=80)

```




Cross-Site Scripting (XSS)

Cross site scripting attacks happen when user-submitted content that has not been sanitized is displayed to other users. The most obvious version of this exploit is where one user submits a comment that includes a script that performs a malicious action, and anyone who views the comments page has that script executed on their machine.

For example, consider a page displayed by this code:

```
...
@sample.route("/product_comments")
def search():
    db = getdb()
    cur = db.cursor()
    prod_id = request.args.get('prod_id')
    cur.execute("select * from products where id = %(prod_id)s", {'prod_id':
prod_id})
    output = ""
    for row in cur.fetchall():
        output = output + str(row[0]) + ": " + str(row[1]) + "<br />"
    db.close()
    return output
...
```

This code simply extracts comment data from the database and displays it on the page. If a user named Robin, were to submit content such as:

```
<script type="text/javascript">alert("Gotcha!")</script>
```

Then a user coming to the page would get content that looks like this:

```
Robin: <script type="text/javascript">alert("Gotcha!")</script>
```

When that user loads the page, they would see an alert box, triggered by the inserted Javascript.

Now, in this case, we are just displaying an alert, which is harmless. But that script could just as easily have done something malicious, such as stealing cookies, or worse.

The bigger problem is that you are dealing with more than the data that is stored in your database, or “Stored XSS Attacks.” For example, consider this page, which displays content from a request parameter:

```
...
<h1>Search results for {{ request.args['search_term'] }}</h1>
{ for item in cursor }
...
```

A hacker could trick someone into visiting your page with a link in an email that provides malicious code in a parameter:

```
http://www.example.com?
search_term=%3Cscript%3Ealert%28%27Gotcha%21%27%29%3C%2Fscript%3E
```

This link, which includes a “url encoded” version of the script, would result in an unsuspecting user seeing a page of:

```
...
<h1>Search results for <script>alert('Gotcha!')</script></h1>
...
```

And that, of course, would execute the script.

This is called a **Reflected XSS Attack**.

So how do you prevent it?

The main strategy is to sanitize content where possible, and if it cannot be sanitized, do not display it.

Experienced web developers usually know to check for malicious content in comments, but there are other places you must check for “untrusted” content. OWASP recommends never displaying untrusted content in the following locations:

- Inside script tags
- Inside comments
- As part of attribute names
- As part of tag names
- In CSS (within style tags)

You can display content in some locations, if it is sanitized first. These locations include:

- As the content of an HTML tag
- As the value of an attribute
- As a variable within your Javascript

Sanitizing content can be a complicated process to get right, as you can see from the wide variety of options an attacker has. It is worth it to use a tool that is built just for sanitizing content, such as OWASP Java HTML Sanitizer, HtmlSanitizer, or Python Bleach.

6.5.6

Cross-Site Request Forgery (CSRF)



Another type of attack that shares some aspects of XSS attacks is Cross Site Request Forgery (CSRF), sometimes pronounced “Sea Surf.” In both cases, the attacker intends for the user to execute the attacker’s code, usually without even knowing it. The difference is that CSRF attacks are typically aimed not at the target site, but rather at a **different** site, one into which the user has already authenticated.

Here is an example. Let’s say the user logs into their bank website, **<http://greatbank.example.com>**. In another window, they are on a discussion page that includes an interesting looking link, and they click it.

Unfortunately, the link was for **http://greatbank.example.com/changeemail?new_email=attacker@example.com**. The browser thinks this is just a normal link, so it calls the URL, sending the cookies for greatbank.example.com, which, as you recall, include the user’s authentication credentials. As far as the bank is concerned, this request came from the user, and it executes the change. Now the attacker can go ahead and change the user’s password, then log into the bank and do whatever damage they want.

Note that even if the user is smart enough not to click on a strange link like that, if a site is vulnerable to XSS attacks, this attack can be carried out without the user having to do anything. A carefully crafted

`` tag can achieve the same result.

An interesting aspect of CSRF is that the attacker never actually gets the results of the attack; they can only judge the results after the fact, and they have to be able to predict what the effects will be to take advantage of a successful attack.

CSRF attacks are notoriously difficult to prevent, but not impossible. One method is to include a hidden token that must accompany any requests from the user. For example, that bank login form might look like this:

```
...
<form action="https://greatbank.example.com" method="POST">
Username: <input type="text" name="username" style="width: 200px" />
Password: <input type="text" name="password" style="width: 200px" />
<br >
<input type="hidden" name="CSRFToken" value="d063937d-c117-46e6-8354-6f5d8faff095"
/>
<input type="submit" value="Log in">
</form>
...
```

That `CSRFToken` has to accompany every request from the user for it to be considered legitimate. Because it is impossible for the attacker to predict that token, a request such as https://greatbank.example.com/changeemail?new_email=attacker@example.com will automatically be rejected.



The OWASP Top 10

Now that you know about three of the most well-known attacks, here is the entire OWASP Top 10 list.

- **Injection** - This item consists of all sorts of injection attacks. We talked earlier about SQL injection, but this is only the most common. All databases, such as LDAP databases, Hibernate databases, and others, are potentially vulnerable. In fact, any action that relies on user input is vulnerable, including direct commands. You can mitigate these types of attacks by using parameterized APIs, escaping user input, and by using `LIMIT` clauses to limit exposure in the event of a breach.
- **Broken Authentication** - This item relates to multiple problems with user credentials, from stolen credentials database to default passwords shipped with a product. You can mitigate these attacks by avoiding default passwords, by requiring multi-factor authentication, and using techniques such as lengthening waiting periods after failed logins.
- **Sensitive Data Exposure** - This item refers to when attackers steal sensitive information such as passwords or personal information. You can help to prevent these attacks by storing as little personal information as possible, and by encrypting the information you do store.
- **XML External Entities (XXE)** - This item refers to attacks made possible by a feature of XML that enables users to incorporate external information using entities. You can solve this problem by disabling XML Entity and DTD processing, or by simply using another format, such as JSON, instead of XML.
- **Broken Access Control** - This item refers to the need to ensure that you have not built an application that enables users to circumvent existing authentication requirements. For example, attackers should not be able to access admin functions just by browsing directly to them. In other words, do not rely on “security through obscurity”. Make sure to protect all resources and functions that need to be protected on the server side, ensuring that any and all access really is authorized.
- **Security Misconfiguration** - This item refers to the need to ensure that the system itself is properly configured. Is all software properly patched and configured? Is the firewall running? Prevention of these types of problems requires careful, consistent hardening of systems and applications. Reduce the attack surface that is available. To do this, only install the services you actually need, and try to separate out components that are not related to different systems to reduce the attack surface further.
- **Cross-Site Scripting (XSS)** - This item refers to the ability for an attacker to use the dynamic functions of a site to inject malicious content into the page, either in a persistent way, such as within the body of comments, or as part of a single request. Mitigating these problems requires careful consideration of where you are including untrusted content in your page, as well as sanitizing any untrusted content you do include.
- **Insecure Deserialization** - This item describes issues that can occur if attackers can access, and potentially change, serialized versions of data and objects, that is, text versions of objects that can be reconstituted into objects by the server. For example, if a user’s information is passed around as a JSON object that includes their access privileges, they could conceivably give themselves admin privileges by changing the content of that object. Because objects can include executable code, this exploit can be particularly dangerous, even if it is not necessarily simple to exploit. To help prevent issues, do not accept serialized objects from untrusted sources, or if you must, ensure validation before deserializing the objects.
- **Using Components with Known Vulnerabilities** - One of the advantages today’s developers have is that most of the core functions you are trying to perform have probably already been written and included in an existing software package, and it is probably open source. However, many of the packages that are available also include publicly available exploits. To fix this, ensure that you are using only necessary features and secure packages, downloaded from official sources, and verified with a signature.
- **Insufficient Logging and Monitoring** - This item reminds you that your most basic responsibility is to ensure that you are logging everything important that is happening in your system so that you can

detect attacks, preferably before they succeed. It is particularly important to ensure that your logs are in a common format so that they can be easily consumed by reporting tools, and that they are auditable to detect (or better yet prevent) tampering.

6.5.8

Evolution of Password Systems



Simple Plaintext Passwords

The first passwords were simple plaintext ones stored in databases. These allowed multiple users using the same core processor to have unique privacy settings. This was before sophisticated hacking networks and password-cracking programs came into being.

The rule for plaintext passwords is very simple: Just store them. You have a database with a table for all your users, and it would probably look something like `(id, username, password)`. After the account is created, you store the username and password in these fields as plaintext, and on login, you extract the row associated with the inputted username and compare the inputted password with the password from the database. If it matches, you let your user in. Perfectly simple and easy to implement.

The following sample shows how to create/verify a new user profile with plaintext format:

```
#####Plain Text
#####
@app.route('/signup/v1', methods=['POST'])
def signup_v1():
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS USER_PLAIN
                (USERNAME TEXT PRIMARY KEY NOT NULL,
                 PASSWORD TEXT NOT NULL);''')
    conn.commit()
    try:
        c.execute("INSERT INTO USER_PLAIN (USERNAME,PASSWORD) "
                  "VALUES ('{0}', '{1}').format(request.form['username'],
request.form['password']))
        conn.commit()
    except sqlite3.IntegrityError:
        return "username has been registered."
    print('username: ', request.form['username'], ' password: ',
request.form['password'])
    return "signup success"
```

Verify the new signed-up user account with login function in plaintext format, and you can see that user profile's data all are in plaintext format.

```
def verify_plain(username, password):
    conn = sqlite3.connect('test.db')
    c = conn.cursor()
    query = "SELECT PASSWORD FROM USER_PLAIN WHERE USERNAME =
'{0}'".format(username)
    c.execute(query)
    records = c.fetchone()
    conn.close()
    if not records:
        return False
    return records[0] == password
@app.route('/login/v1', methods=['GET', 'POST'])
def login_v1():
    error = None
    if request.method == 'POST':
        if verify_plain(request.form['username'], request.form['password']):
            error = 'login success'
        else:
            error = 'Invalid username/password'
    else:
        error = 'Invalid Method'
    return error
```

Obviously, plaintext is an insecure way of storing passwords. If your database was hacked, your user's passwords would be exposed to hackers directly.

Password Hashing

Storing passwords is risky and complex at the same time. A simple approach to storing passwords is to create a table in your database that maps a username with a password. When a user logs in, the server gets a request for authentication with a payload that contains a username and a password. We look up the username in the table and compare the password provided with the password stored. A match gives the user access to the application. The security strength and resilience of this model depends on how the password is stored. The most basic, but also the least secure, password storage format is cleartext.

Storing passwords in cleartext is the equivalent of writing them down in a piece of digital paper. If an attacker was to break into the database and steal the passwords table, the attacker could then access each user account. This problem is compounded by the fact that many users re-use or use variations of a single password, potentially allowing the attacker to access other services different from the one being compromised. The attack could come from within the organization. A software engineer with access to the database could abuse that access power, retrieve the cleartext credentials, and access any account.

Hashing

A more secure way to store a password is to transform it into data that cannot be converted back to the original password. This mechanism is known as hashing.

By dictionary definition, hashing refers to "chopping something into small pieces" to make it look like a "confused mess". That definition closely applies to what hashing represents in computing.

In cryptography, a hash function is a mathematical algorithm that maps data of any size to a bit string of a fixed size. This function input can be referred to as message or simply as input. The fixed-size string function output is known as the hash or the message digest. As stated by OWASP, hash functions used in cryptography have the following key properties:

- It is easy and practical to compute the hash, but difficult or impossible to re-generate the original input if only the hash value is known.
- It is difficult to create an initial input that would match a specific desired output.

Thus, in contrast to encryption, hashing is a one-way mechanism. The data that is hashed cannot be practically "unhashed".

Hashing example

Here is a code example to import the constructor method of the SHA-256 hash algorithm from the `hashlib` module:

```
from hashlib import sha256
# Create an instance of the sha256 class
h = sha256()
# Uses the update() method to update the hash object
h.update(b'devnetpassword1')
# Uses the hexdigest() method to get the digest of the string passed to the
update() method
hash = h.hexdigest()
# The digest is the output of the hash function.
# Print the hash variable to see the hash value in the console:
print(hash)
```

As an example, you would see the following output:

```
a75e46e47a3c4cf3aaefe1e549949c90e90e0fe306a2e37d2880702a62b0ff31
```

Salted password

There are many passwords are encrypted but can be guessed. They have been mined from hacked sites and placed into lists. These lists have made hashed passwords much easier to crack. To guarantee the uniqueness of the passwords, increase their complexity, and prevent password attacks even when the inputs are the same, a salt (which is simply random data) is added to the input of a hash function. This is known as a salted password.

Using cryptographic hashing for more secure password storage

The irreversible mathematical properties of hashing make it a phenomenal mechanism to conceal passwords at rest and in motion. Another critical property that makes hash functions suitable for password storage is that they are deterministic. A deterministic function is a function that, given the same input, always produces the same output. This is vital for authentication because you need to have the guarantee that a given password will always produce the same hash. Otherwise, it would be impossible to consistently verify user credentials with this technique.

Adding salt to password hashing

A salt is added to the hashing process to force hash uniqueness. It increases complexity without increasing user requirements, and mitigates password attacks such as rainbow tables. In cryptography, salting hashes refers to adding random data to the input of a hash function to guarantee a unique output, which is the hash, even when the inputs are the same. Consequently, the unique hash produced by adding the salt can protect against different attack vectors, while slowing down dictionary and brute-force attacks.

Sample

Hashed passwords are not unique to themselves due to the deterministic nature of hash function: when given the same input, the same output is always produced. If `devnet_alice` and `devnet_bob` both choose `devnetpassword1` as a password, their hash would be the same:

| username | hash |
|---------------|--|
| devnet_alice | 0e8438ea39227b83229f78d9e53ce58b7f468278c2ffcf45f9316150bd8e5201 |
| devnet_ava | a75e46e47a3c4cf3aaefe1e549949c90e90e0fe306a2e37d2880702a62b0f |
| devnet_bob | 0e8438ea39227b83229f78d9e53ce58b7f468278c2ffcf45f9316150bd8e5201 |
| devnet_blaine | 6421e62bf41b6d52963b42d5467e25ed18d0ef26e5dfde8825e639600d2c |
| devnet_devon | 9314342333718a996b107ff2de51e8105466a9f48310f1b47b679f64d60f5 |
| devnet_dave | 5d86d07ab6c68ccdeab2815b26598c6d9ce0db92f455d499f70bca5067cc |

And this example reveals, `devnet_alice` and `devnet_bob` have the same password because we can see that both share the same hash:

```
0e8438ea39227b83229f78d9e53ce58b7f468278c2ffcf45f9316150bd8e5201
```

The attacker can better predict the password that legitimately maps to that hash. After the password is known, the same password can be used to access all the accounts that use that hash.

Mitigating password attacks with a salt

To mitigate the damage that a rainbow table or a dictionary attack could do, salt the passwords. According to OWASP Guidelines, a salt is a fixed-length cryptographically-strong random value that is added to the input of hash functions to create unique hashes for every input, regardless of whether the input is unique. A salt makes a hash function look non-deterministic, which is good as you do not want to reveal password duplications through your hashing.

Let's say that you have password `devnet_password1` and the salt `salt706173776f726473616c74a`. You can salt that password by either appending or prepending the salt to it. For example:

```
devnetpassword1salt706173776f726473616c74a or  
salt706173776f726473616c74adevnetpassword1
```

are valid salted passwords. After the salt is added, you can then hash it and get different hash values:

```
cefee7f060ed49766d75bd4ca2fd119d7fcabe795b9425f4fa9d7115f355ab8c and  
d00c162358af0e645b90bf291836cbf7d523157baf85e96492b151e0624ee041
```

Let's say `devnet_alice` and `devnet_bob` decide to use both the same password, `devnetpassword1`. For `devnet_alice`, we'll use `salt706173776f726473616c74a` again as the salt. However, for `devnet_bob`, we'll use `salt706173776f726473616253b` as the salt:

Hashed and salted password examples

User: `devnet_alice`

Password: `devnetpassword1`

Salt: `salt706173776f726473616c74a`

Salted input: `devnetpassword1salt706173776f726473616c74a`

Hash (SHA-256): `cefee7f060ed49766d75bd4ca2fd119d7fcabe795b9425f4fa9d7115f355ab8c`

User: `devnet_bob`

Password: `devnetpassword1`

Salt: `salt706173776f726473616253b`

Salted input: `devnetpassword1salt706173776f726473616253b`

Hash (SHA-256): `41fffe05d7aca370abaff6762443d9326ce22107783b8ff5bb0cf576020fc1d5`

Different users, same password. Different salts, different hashes. If someone looked at the full list of password hashes, no one would be able to tell that `devnet_alice` and `devnet_bob` both use the same password. Each unique salt extends the password `devnetpassword1` and transforms it into a unique password.

In practice, store the salt in cleartext along with the hash in your database. You would store the salt `salt706173776f726473616c74a`, the hash `cefee7f060ed49766d75bd4ca2fd119d7fcabe795b9425f4fa9d7115f355ab8c`, and the username together so that when the user logs in, you can lookup the username, append the salt to the provided password, hash it, and then verify if the stored hash matches the computed hash.

Additional factors for authentication

Even with the incorporation of these password strength checkers, single-factor authentication still leaves your enterprise vulnerable. Hackers can crack any password, it may take significant time, but they can. Moreover, they could always steal passwords via social engineering. Thus, you should incorporate multi-factor authentication (MFA). Incorporating other authentication factors confounds hackers who may have cracked your password.

Single-factor authentication (SFA)

Single-factor authentication is the simplest form of authentication methods. With SFA, a person matches one credential to verify himself or herself online. The most popular example of this would be a password (credential) to a username. Most verification today uses this type of authentication method.

What are the risks of single-factor authentication?

Online sites can have users' passwords leaked by a hacker. Without an additional factor to your password to confirm your identity, all a malicious user needs is your password to gain access. Hopefully it is not a website that has additional personal information stored, such as your credit card information, home address, or other personal information used to identify you.

Often a user's password is simple so that it is easy to remember. The simpler the password, the easier it is to crack. A malicious user may guess your password because they know you personally, or because they were able to find out certain things about you, such as your birthdate, favorite actor/actress or pet's name. A malicious user may also crack your password by using a bot to generate the right combination of letters/numbers to match your simple, secret identification method. In either example, it is going to be a hassle to recover your account(s). Hopefully your simple password is not being reused with other online entities.

Single-factor authentication (SFA) is quickly becoming the weak link of security measures, similar to cross domains. There is a growing number of products, websites and apps that offer two-factor and multi-factor authentication. Whether it is just two factors, or three or more, MFA is the way to make your accounts much harder for attackers to access.

Two-factor authentication (2FA)

Two-factor authentication uses the same password/username combination, but with the addition of being asked to verify who a person is by using something only he or she owns, such as a mobile device. Putting it simply: it uses two factors to confirm an identity.

Multi-factor authentication (MFA)

Multi-factor authentication (MFA) is a method of computer access control in which a user is only granted access after successfully presenting several separate pieces of evidence to an authentication mechanism. Typically at least two of the following categories are required for MFA: knowledge (something they know); possession (something they have), and inherence (something they are).

2FA is a subset of that. It is just a type of MFA where you only need two pieces of evidence, that is, two "factors". When you log in to Google, Twitter, or LinkedIn, or you make a purchase on Amazon, you can use their two-step validation to require your password (something you know) and a special text sent to your phone (something you have). If you do not have your password and your phone, you do not get in.

6.5.9

Password Cracking



Techniques used for finding a password that allows entry is known as "cracking" the security intended by the password.

Password guessing

Password guessing is an online technique that involves attempting to authenticate a particular user to the system. Password guessing may be detected by monitoring the failed login system logs. Clipping levels are used to differentiate between malicious attacks and normal users accidentally mistyping their passwords. Clipping levels define a minimum reporting threshold level. Using the password guessing example, a clipping level might be established so that the audit system only alerts if failed authentication occurs more frequently than five times in an hour for a particular user. Clipping levels can help to differentiate the attacks from legitimate mistakes; however they can also cause false negatives if the attackers can glean the threshold beneath which they must operate.

Preventing successful password guessing attacks is typically done with account lockouts. Account lockouts are used to prevent an attacker from being able to simply guess the correct password by attempting a large number of potential passwords.

Dictionary attack

In cryptanalysis and computer security, a dictionary attack is a form of brute force attack for defeating a cipher or authentication mechanism by trying to determine its decryption key or passphrase by trying hundreds or sometimes millions of likely possibilities, such as words in a dictionary.

A dictionary attack is based on trying all the strings in a pre-arranged listing, typically derived from a list of words such as in a dictionary (hence the phrase dictionary attack). In contrast to a brute force attack, where a large proportion of the key space is searched systematically, a dictionary attack tries only those possibilities which are deemed most likely to succeed.

Dictionary attacks often succeed because many people have a tendency to choose short passwords that are ordinary words or common passwords, or simple variants obtained, for example, by appending a digit or punctuation character. Dictionary attacks are relatively easy to defeat, for example, by using a passphrase or otherwise choosing a password that is not a simple variant of a word found in any dictionary or listing of commonly used passwords.

Pre-computed dictionary attack or rainbow table attack

It is possible to achieve a time/space tradeoff by pre-computing a list of hashes of dictionary words, and storing these in a database using the hash as the key. This requires a considerable amount of preparation time, but allows the actual attack to be executed faster. The storage requirements for the pre-computed tables were once a major cost, but are less of an issue today because of the low cost of disk storage. Pre-computed dictionary attacks are particularly effective when a large number of passwords are to be cracked. The pre-computed dictionary need be generated only once, and when it is completed, password hashes can be looked up almost instantly at any time to find the corresponding password. A more refined approach involves the use of rainbow tables, which reduce storage requirements at the cost of slightly longer lookup-times. Pre-computed dictionary attacks, or "rainbow table attacks", can be thwarted by the use of salt, a technique that forces the hash dictionary to be recomputed for each password sought, making pre-computation infeasible, provided the number of possible salt values is large enough.

Social engineering

Social engineering for password cracking involves a person convincing or tricking another person into supplying the attacker with access.

Information security culture

Employee behavior can have a big impact on information security in organizations. Cultural concepts can help different segments of the organization work effectively towards information security. "Exploring the Relationship between Organizational Culture and Information Security Culture" provides the following definition of information security culture: "ISC is the totality of patterns of behavior in an organization that contribute to the protection of information of all kinds." (Reference: Lim, J. S., Chang, S., Maynard, S., & Ahmad, A. (2009). *Exploring the Relationship between Organizational Culture and Information Security Culture*.)

Techniques

All social engineering techniques are based on cognitive biases, where humans make decisions based on known or assumed attributes. Cognitive biases are exploited in various combinations to create attack techniques, such as stealing an employee's private information. Phone calls and conversations are the most common type of social engineering. Another example of a social engineering attack would be someone posing as a service provider such as exterminators, technicians, or safety inspectors to gain access to people or information systems.

Techniques can involve gaining trust through conversations on social media sites that lead to eventually asking for banking information or valuable passwords.

Another example would be using an authoritative, trusted source of information like a company bulletin board to change the help desk number to a false number to collect information. When an employee reads the posted, but fake, information on the board and calls the number, they are more willing to give over passwords or other credentials because they believe the post on the board is real.

Social engineering relies heavily on the six principles of influence established by Robert Cialdini in his book, *Influence*.

Six key principles of human influence

- **Reciprocity** – Our social norms mean that we tend to return a favor when asked. You can imagine this works well for phone conversations to gain information.
- **Commitment and consistency** – When people commit, whether in person, in writing, or on a web site, they are more likely to honor that commitment in order to preserve their self-image. An example is the checkbox on a popover on a web site stating, "I'll sign up later." When someone checks it, even when that original incentive goes away, they tend to honor that social contract, or go ahead and sign up so that they do not feel dishonest.
- **Social proof** – When people see someone else doing something, such as looking up, others will stop to do the same. This type of conformity lets people manipulate others into sharing information they would not usually leak.
- **Authority** – People nearly always obey authority figures, even if they are asked to do a dangerous or harmful task to another person. This authority principle means that attackers who seem to be authoritative or representing an authority figure are more likely to gain access.
- **Liking** – Liable people are able to persuade others more effectively. This principle is well-known by sales people. People are easily persuaded by familiar people whom they like.
- **Scarcity** – When people believe or perceive that something is limited in amount or only available for a short time, people will act positively and quickly to pick up the desired item or offer.

There are four social engineering vectors, or lines of attack, that can take advantage of these influence principles.

- **Phishing** means the person is fraudulently gaining information, especially through requests for financial information. Often the attempts look like a real web site or email, but link to a collector site instead.
- **Vishing** stands for "voice phishing," so it is associated with voice phone calls to gather private personal information for financial gain.
- **Smishing** involves using SMS text messaging for both urgency and asking for a specific course of action, such as clicking a fake link or sending account information.
- **Impersonation** involves in-person scenarios such as wearing a service provider uniform to gain inside access to a building or system.

Password strength

Password strength is the measure of a password's efficiency to resist password cracking attacks. The strength of a password is determined by:

Length - This is the number of characters the password contains.

Complexity - This means it uses a combination of letters, numbers, and symbols.

Unpredictability - Is it something that can be guessed easily by an attacker?

Here is a practical example of three passwords:

- password
- passw0rd123
- #W)rdPass1\$

The screenshot shows a password strength checker interface. At the top, the password 'password' is entered in a text box. Below the text box, a red banner displays 'Very Weak'. Underneath, it states '8 characters containing:' followed by a green checkmark for 'Lower case' and three red X marks for 'Upper case', 'Numbers', and 'Symbols'. A table below shows 'Time to crack your password:' as '0 seconds' and a 'Review:' section stating: 'Oh dear, using that password is like leaving your front door wide open. Your password is very weak because it is a common password.' At the bottom, a note says: 'Your passwords are never stored. Even if they were, we have no idea who you are!'

The word `password` used as a password is very weak. It costs no time to crack it.

Passw0rd123

Very Weak

11 characters containing: ✓ Lower case ✓ Upper case ✓ Numbers ✗ Symbols

| | |
|--|--|
| Time to crack your password: 0 seconds | Review: Oh dear, using that password is like leaving your front door wide open. Your password is very weak because it contains a common password and a sequence of characters. |
|--|--|

Your passwords are never stored. Even if they were, we have no idea who you are!

The password used is `passw0rd123`, but the strength of it is still very weak. Again, it takes little time to crack it.

#W)rdPass1\$

Very Strong

11 characters containing: ✓ Lower case ✓ Upper case ✓ Numbers ✓ Symbols

| | |
|---|--|
| Time to crack your password: 21 years | Review: Fantastic, using that password makes you as secure as Fort Knox. |
|---|--|

Your passwords are never stored. Even if they were, we have no idea who you are!

In this example, the password used is `#W)rdPass1$`, and it has strength. One estimate is that it would take about 21 years to crack it.

Password strength checkers and validation tools

An example of a password strength checker is a password manager. This type of tool is a best approach to ensure that you are using security strength password. In general, the password strength validation tool is built in with password system as the input validator to make sure the user's password is compatible with latest identity management guidelines.

Best practices

There are a few best practices to secure user login attempts. These include notifying users of suspicious behavior, and limiting the number of password and username login attempts.

NIST Digital Identity Guidelines

Here's a brief summary of the NIST 800-63B Digital Identity Guidelines:

- 8-character minimum when a human sets it
- 6-character minimum when set by a system/service
- Support at least 64 characters maximum length
- All ASCII characters (including space) should be supported
- Truncation of the secret (password) shall not be performed when processed
- Check chosen password with known password dictionaries
- Allow at least 10 password attempts before logout
- No complexity requirements
- No password expiration period
- No password hints
- No knowledge-based authentication such as, who was your best friend in high school?
- No SMS for two-factor authentication, instead use a one-time password from an app like Google Authenticator

You can read more about NIST Special Publication 800-63B on the National Institute of Standards and Technology site.

6.5.10

Lab – Explore the Evolution of Password Methods



In this lab, you will create an application that stores a username and password in plaintext in a database using python code. You will then test the server to ensure that not only were the credentials stored correctly, but that a user can use them to login. You will then perform the same actions, but with a hashed password so that the credentials cannot be read. It is important to securely store credentials and other data to prevent different servers and systems from being compromised.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Explore Python Code Storing Passwords in Plain Text
- Part 3: Explore Python Code Storing Passwords Using a Hash

 Explore the Evolution of Password Methods



6.4

Networks for Application Development an...

Summary: Application Deployment and Se...

6.6

