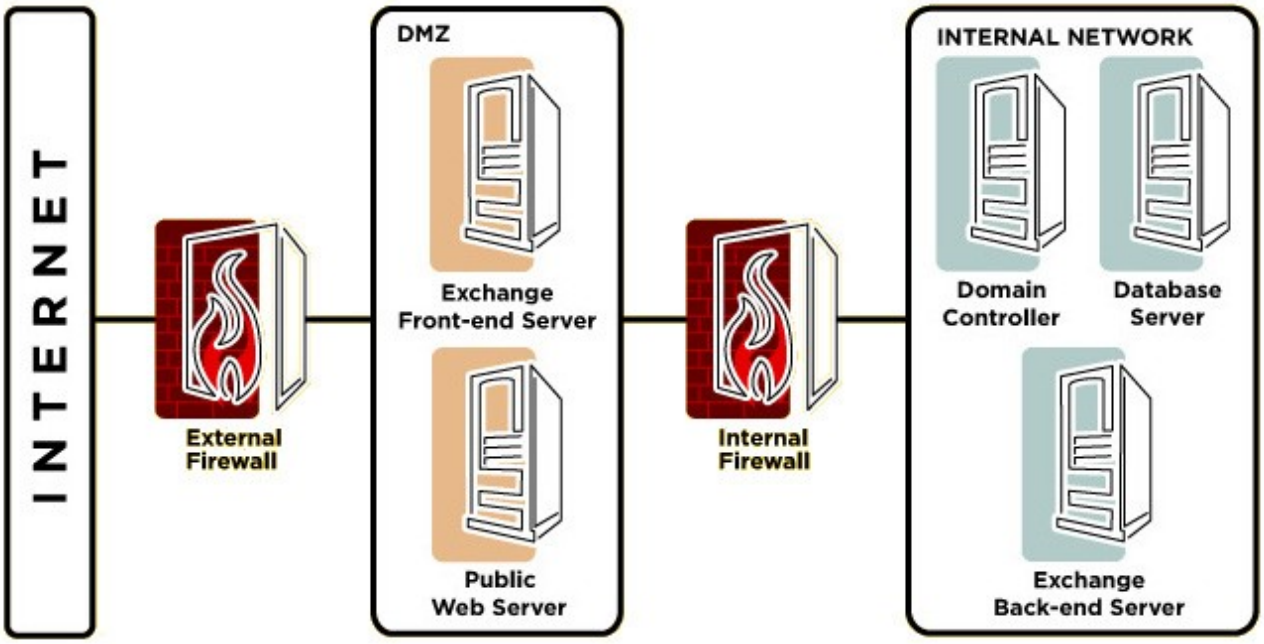
DB Best Practices

our experience?
- mostly a DB on the same machine (server) as some (probably) web-based frontend
- we usually restrict access to the DB such that only the localhost can connect
- we then (hopefully) set a good username and password as login credentials to the DB
- we then (also hopefully) use practices to access the DB through the web interface
  - i.e., use secure queries that minimize SQL injection, sanitize input, etc

but how is it done in practice?
- mostly by implementing a DMZ of some sort

DMZ?
- demilitarized zone (or perimeter network)
- physical or logical subnet that contains and exposes external-facing services
- exposed to the outside (i.e., the Internet – but can mean any external untrusted network)
- external to an internal (secure) network (LAN) that we don't want the outside to have access to
- adds an additional later of security to a LAN
- external nodes can only access what's been exposed in the DMZ
- gives extra time to mitigate attacks, breaches, etc, before they penetrate to the LAN
- DMZ is derived from the non-technical (non-computing) term of the same name
  - an area between nation states in which military operation is not permitted
  - e.g., the DMZ between North and South Korea



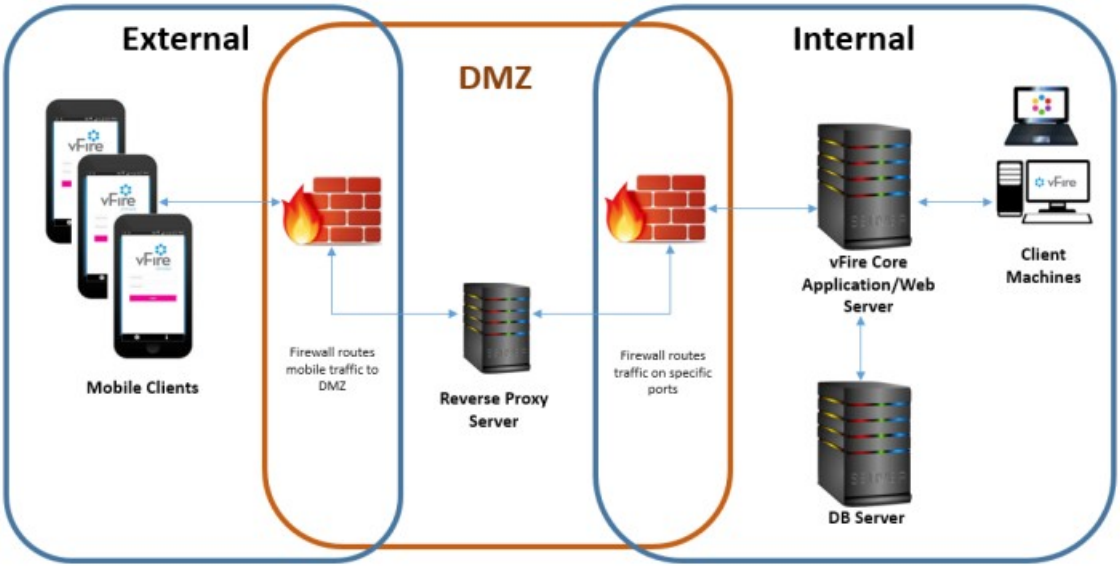
- the area between the Internet and the DMZ can (and usually is) firewalled
- the area between the DMZ and the internal LAN is (always) firewalled
- the rules are:
  - connections from outside can only get to hosts in the DMZ (and on specific ports)
  - connections from the outside to the inside are (absolutely) blocked
  - connections from the inside to either the DMZ or the outside are fine and dandy
  - only hosts in the DMZ may establish connections to the inside
    - again, only on well known and permitted ports

most organizations use a DMZ
exposed services are placed in the DMZ
firewalls filter traffic from the Internet to the DMZ
firewalls further filter requests from services in the DMZ to the LAN

so services within the DMZ can make (secured) requests to the (protected) LAN
DBs, for example, are located in the secured LAN
username/password credentials, credit card info, etc, are all located within the secured LAN
so breaches of services within the DMZ don't provide attackers with (typically) meaningful info
this is not foolproof!
a breach in a misconfigured service in the DMZ is possible (we are fallible!)
this *could* be used to gain unauthorized access to the LAN
note that this setup does nothing to protect against insider attacks!

proxy servers
sometimes, we want to monitor internal users when they make requests to the Internet
sometimes, we want to control what can be sent from a LAN to the Internet (and vice versa)
e.g., HIPAA (Health Insurance Portability and Accountability Act)
so a proxy server is installed in the DMZ
all internal clients must use the proxy to gain access to the Internet
a benefit is that often requested content is cached there (so it's faster!)
often, there are several proxy servers for load balancing

reverse proxy server
it's also located within the DMZ
provides indirect access to internal LAN services from the Internet
so a bit of the reverse of a proxy server
only the reverse proxy server can access internal resources
Internet users can only access the reverse proxy server
basically functions as a relay between the Internet and services within the LAN
all responses (of requests) appear to originate from the reverse proxy
so untrusted users don't have knowledge of internal services



can hide the existence and specifics of internal (LAN) services
can be fitted with extra hardware (e.g., GPUs) to support faster encryption, etc
can serve as a load balancer
can act as a router of sorts, making multiple sources appear to come from a single one
e.g., multiple LAN sources

setting up the environment
baseline: Linux Mint *something*
update the repositories:
```
sudo apt update
```
install the Apache web server:
```
sudo apt install apache2
```
install MySQL:
```
sudo apt install mysql-server
```
install PHP (including support for Apache and MySQL):
```
sudo apt install php libapache2-mod-php php-mysql
```

testing the environment
create a PHP file (`/var/www/html/index.php`) as the main web page:
```
sudo vim /var/www/html/index.php
```

and add:
```
<?php echo "It works!"; ?>
```
remove the old web page (`/var/www/html/index.html`)
```
sudo rm /var/www/html/index.html
```
to test: browse to `http://<ip>`
test MySQL:
```
sudo mysql
```

setting up a user login/registration system
source: https://www.tutorialrepublic.com/php-tutorial/php-mysql-login-system.php
first, we need the web files (we'll place these in `/var/www/html`):
login.php
allows users to login to the system (or to register if they're new)
register.php
allows new users to register
welcome.php/index.php
provides a welcome page if a user successfully logs in
reset-password.php
allows existing users to reset/change their password
logout.php
allows users to logout of the system
we'll also use an existing CSS (bootstrap.css)

next, we need to setup the DB (we'll place these in `/var/www/html/config`):
config.php
provides DB connection info
mysql_init.sql

provides an initialization script to setup the DB and DB user

dump_db
a bash script that dumps the DB to a specified SQL file

index.php
a blank file to prevent users from seeing a directory listing if they browse here

v1.sql, v2.sql, v3.sql, …
provides initial DB configurations for the various examples below

setup the user:
```
sudo mysql < mysql_init.sql
```
configure the DB:
```
mysql -udbstuff -p < v1.sql
```
(we'll do the same with the other versions later)
note that the password for the user dbstuff is dbstuff

DB versions

v1: storing usernames and passwords (in plaintext) – BAD!
access to the DB or sniffing traffic gets everything
no one should be able to look up someone else's password
obviously, two users with the same password is obvious!

v2: storing usernames and encrypted passwords (with a symmetric key) – BAD!
knowing the key means anyone can decrypt the password
again, no one should be able to look up someone else's password
this is recoverable!
obviously, two users with the same password is obvious!
v2 could become v1 with knowledge of the key
so, some rules:
rule 1: user passwords should not be recoverable from the DB
rule 2: identical (or similar) passwords should be different when stored in the DB
rule 3: the DB should give no hints of password length

v3: storing usernames and hashed passwords (with a one-way hash function) – MEH
but users with the same password are stored identically in the DB

v4: adding salt – GOOD
salt? it's a nonce (a number that's used once)
we hash(salt + password);
good, but today's GPUs can do math very quickly
a $5K "machine" can compute 100 billion SHA-256 hashes per second!
short salts and (more importantly) short passwords are still weak
salt can be stored plainly in the DB (!)
used to prevent two users with the same password from getting the same hash
you can store the salt in plaintext without any form of obfuscation or encryption
but don't just give it out to anyone who wants it
salt addresses precomputation attacks (e.g., rainbow tables)
involve creating a DB of hashes and their plaintexts
hashes can be searched for and immediately reversed into plaintext
can also add pepper (!)
basically a second salt that is constant between individual passwords
it's not stored in the database (instead, in the (PHP) code)
an attacker has to have access to both the DB and the code

we hash(salt + password + pepper)

v5: using hash stretching – BEST

many individual hash calculations

it would take significantly longer to compute on the attacker's side

but also a bit longer from the legitimate user's perspective (but not so bad as to notice)

standards: PBKDF2 (maybe the better one?), bcrypt, scrypt

good hashing algorithm: HMAC-SHA-256

HMAC-SHA-256

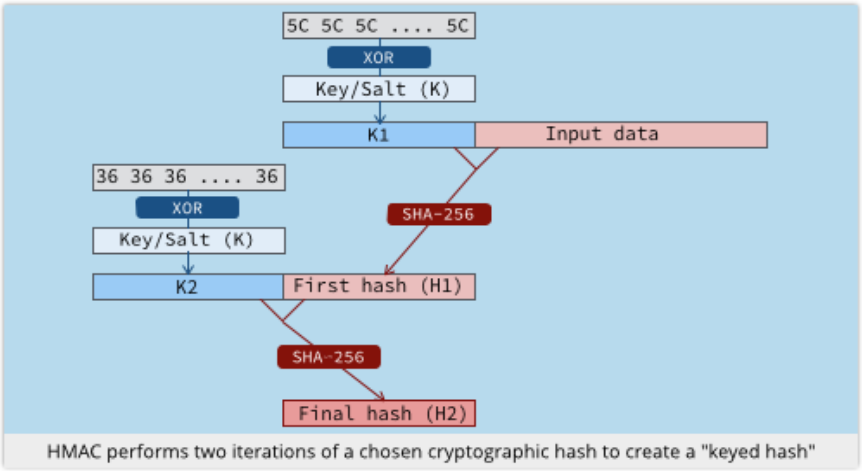
a special way of using SHA-256 without just having a straight hash

take a random key or salt K, flip some bits (XOR with 5C5C5C…) → K1

compute SHA-256 of K1 plus the user's password → H1

flip a different set of bits in K (XOR with 363636…) → K2

compute SHA-256 of K2 plus H1 → H2



HMAC performs two iterations of a chosen cryptographic hash to create a "keyed hash"

PBKDF2

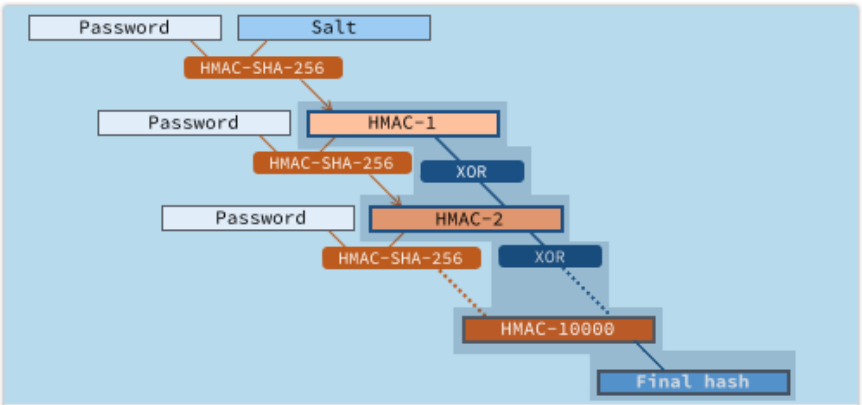
implements hash stretching (hashing many times to produce a final hash that's stored in the DB)

for the first iteration of PBKDF2:

feed the password and salt through HMAC-SHA-256

for the remaining iterations of PBKDF2:

feed the password and the previously computed hash through HMAC-SHA-256



PBKDF2 using XOR to combine 10,000 successive HMAC-SHA-256 outputs into a final hash

rules:

rule 1: use a strong random number generator to create a >= 16-byte salt
rule 2: feed the salt and password into PBKDF2
rule 3: use HMAC-SHA-256 as the core hash inside PBKDF2
rule 4: perform >= 100K iterations (for now, this is good enough)
rule 5: take 32 bytes (256 bits) of output from PBKDF2 as the final password hash
rule 6: store the iteration count, salt, and final hash in the DB
rule 7: increase the iterations as computational power increases

PHP stores the algorithm used, algorithm options, salt, and hashed password in a single field