WEB SECURITY

Defense Against the Dark Arts 101

FUNDAMENTAL CONCERNS

Authentication

Trusting that someone is who they say they are.

Communication

Transferring data through potentially unreliable middlemen.

Authorization

Giving resource access to the right people.

Control

Limiting or understanding the capabilities of agents.

OWASP TOP 10 (2013)

Injection	Server-side code execution
Broken Authentication	Allows for impersonation
XSS	Client-side code execution
Direct References	Access control can be circumvented
Security Misconfiguration	Vulnerable default/inherited settings
Data Exposure	Data is insecurely transmitted, stored, or simply overshared
Missing Access Control	Users can do things they shouldn't be allowed to do
XSRF	Abuse the target website's trust in the browser
Vulnerable Components	Third-party tools are vulnerable
Unvalidated Redirects	Abusable open-ended forwarding

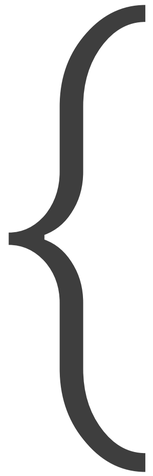
source

AUTHER (CURRENTLY)

HTTP://HUMBLE-HOMES.COM/TINY-HOUSE-PRECARIOUSLY-PERCHED-ON-A-ROCK-WITHSTANDS-SEVERE-FLOODING/

WORKSHOP

- Game with 5 rounds
- Each round
  - attack
  - attack demo
  - defense
  - defense review



- uncovering secrets
- improper access
- injection
- cross-site scripting
- data theft

ROUND 1:
UNCOVERING SECRETS
(SECURITY MISCONFIGURATION)

40 lines (33 sloc) | 1.13 KB

Raw | Blame | History

```
'use strict';

var router = require('express').Router();
var passport = require('passport');
var GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;

var User = require('../api/users/user.model');

router.get('/', passport.authenticate('google', {
  scope: 'email'
}));

router.get('/callback', passport.authenticate('google', {
  successRedirect: '/stories',
  failureRedirect: '/signup'
}));

passport.use(new GoogleStrategy({
  clientID: '238524570915-ivf9lnhm9bsfq13cle5ap8s28d4lmhrp.apps.googleusercontent.com',
  clientSecret: 'GST6VQnVmhx1YIB1vDXXB3PF',
  callbackURL: 'http://127.0.0.1:8080/auth/google/callback'
}, function (token, refreshToken, profile, done) {
  var info = {
    name: profile.displayName,
    // google may not provide an email, if so we'll just fake it
    email: profile.emails ? profile.emails[0].value : [profile.username , 'fake-auther-email.com'].join('@'),
    photo: profile.photos ? profile.photos[0].value : undefined
  };
  User.findOrCreate({
    where: {googleId: profile.id},
    defaults: info
  })
```

UNCOVERING SECRETS

- Assume attacker has access to codebase
- Defense vulnerable if…
  - application secrets are easy to discover
  - files are improperly shared
- Bad stuff: app impersonation, decryption

not (yet) about user secrets

ROUND 2:
IMPROPER ACCESS
(MISSING ACCESS CONTROL)

IMPROPER ACCESS

- Assume attacker is client
- Defense vulnerable if client can act outside of authorization
- Bad stuff: depends on the action and resource

still **not (yet) about user secrets**

IMPROPER ACCESS

- Frontend “access control” is actually just good UX
- True access control comes from backend
- Considerations (be thorough)…
  - requested resource
  - requested action
  - agent making the request

```
curl http://hogwarts.com/api/teachers
```

```
[
  {
    name: 'Dumbledore',
    position: 'headmaster',
    weaknesses: ['candy']
  },
  {
    name: 'Snape',
    position: 'Potions Master',
    weaknesses: ['unrequited love', 'conditioner']
  }
]
```

BEWARE OVERPROTECTION

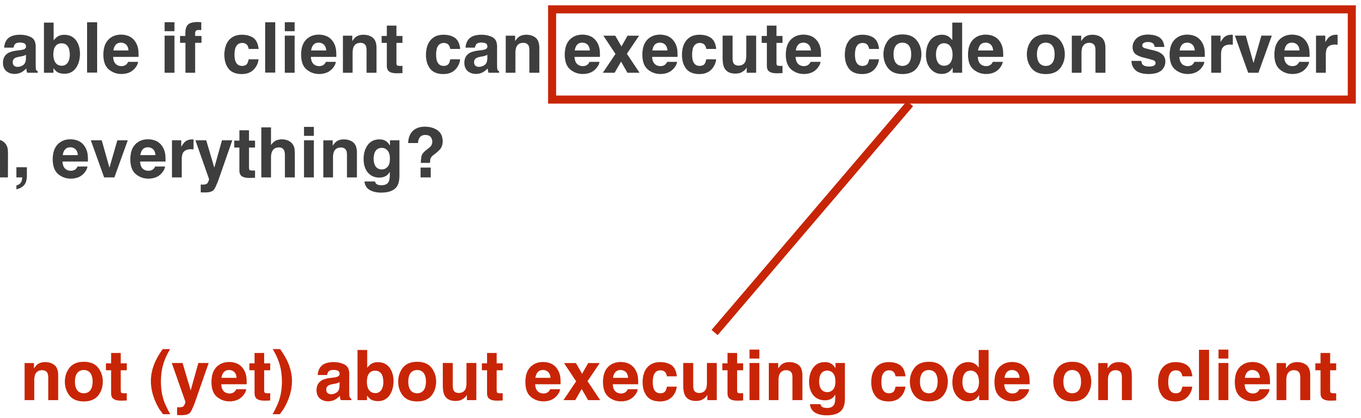
```
// "safe server"
var app = require('express')();
app.use(function (req, res) {
  res.status(403);
  res.send('NONE SHALL ENTER');
});
app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});
...
```

My house is safe from intruders because it does not have a door.

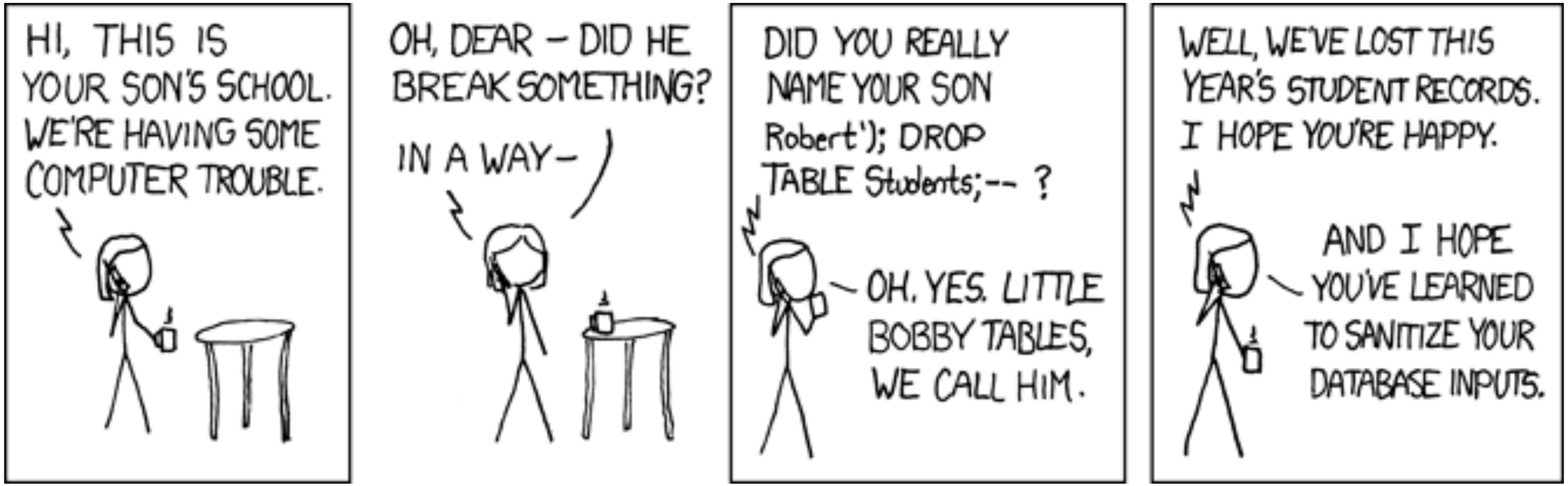
ROUND 3: INJECTION

INJECTION

- Assume attacker is non-admin client
- Defense vulnerable if client can execute code on server
- Bad stuff: umm, everything?



INJECTION



HTTPS://XKCD.COM/327/

INJECTION *server-side execution of attacker-defined code*

User.destroy({});

```
POST /whatever HTTP/1.1
...
{
  "modulePath": "./user.model",
  "method": "destroy",
  "args": [{}]
}
```

```
...
// at end of middleware stack
app.use(function (req, res, next) {
  // allow client to specify *any* module
  var theModule = require(req.body.modulePath);
  // allow client to specify *any* method
  var method = theModule[req.body.methodName];
  // allow client to specify *any* arguments
  var args = req.body.args;
  // blindly invoke!
  var result = method.apply(theModule, args);
  res.json(result);
});
...
```

ROUND 4:
CROSS-SITE SCRIPTING

XSS

- Assume attacker is non-admin client
- Defense vulnerable if client can execute code on *another* client
- Bad stuff: yeah pretty much everything
- Two flavors: “stored” and “reflected”

STORED XSS

4 Comments The Atlantic 1 Login

Recommend Share Sort by Oldest

<script>alert('i am an xss attack');</script>

LOG IN WITH

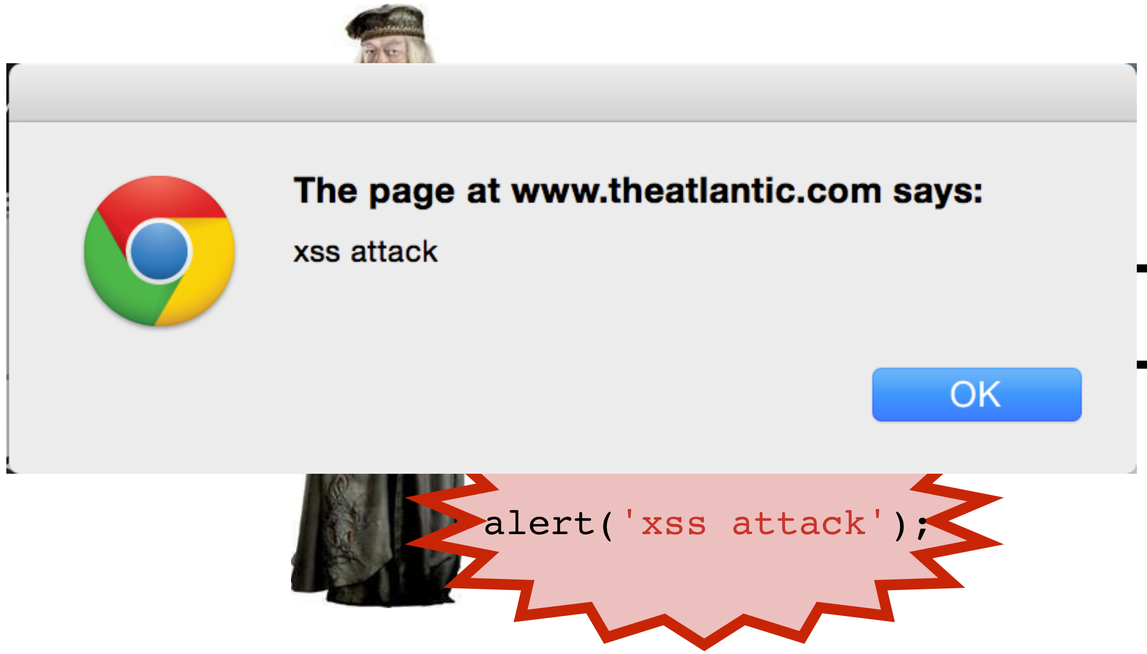
OR SIGN UP WITH DISQUS ?

Name

submit

```
<script>alert('xss attack');</script>
```

STORED XSS

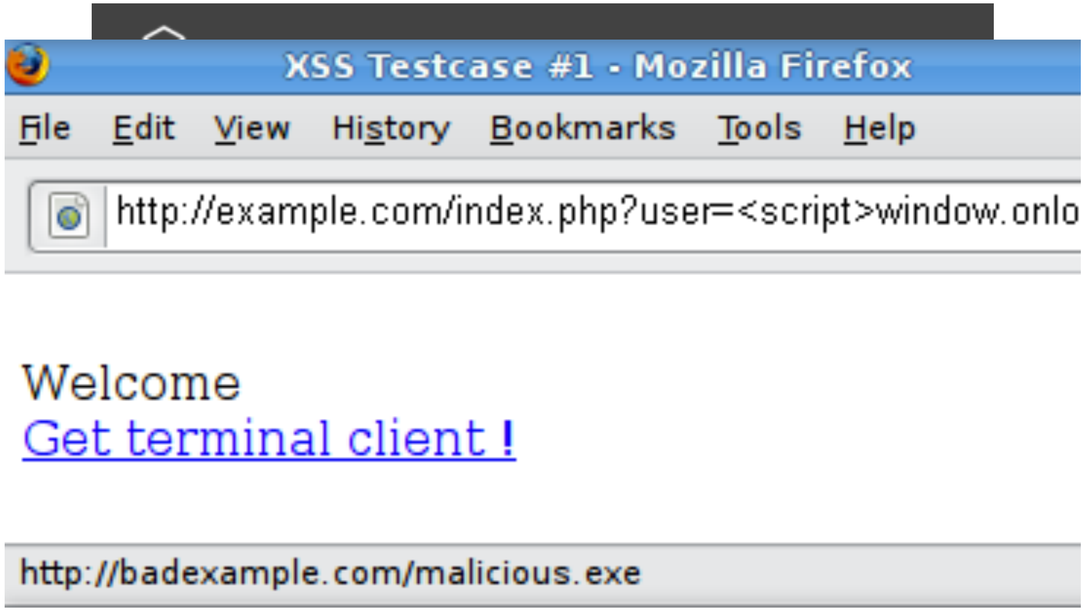

The page at www.theatlantic.com says:
xss attack
OK
alert('xss attack');


<script>alert('xss attack');</script>

REFLECTED XSS

- Server sends parts of request in response
- Attacker forms link with script in it
- Victim clicks link
- Server responds with script
- Script runs on victim’s browser

REFLECTED XSS




```
http://example.com/index.php?user=<script>window.onload = function() {var
AllLinks=document.getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</script>
```

https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_(OTG-INPVAL-001)

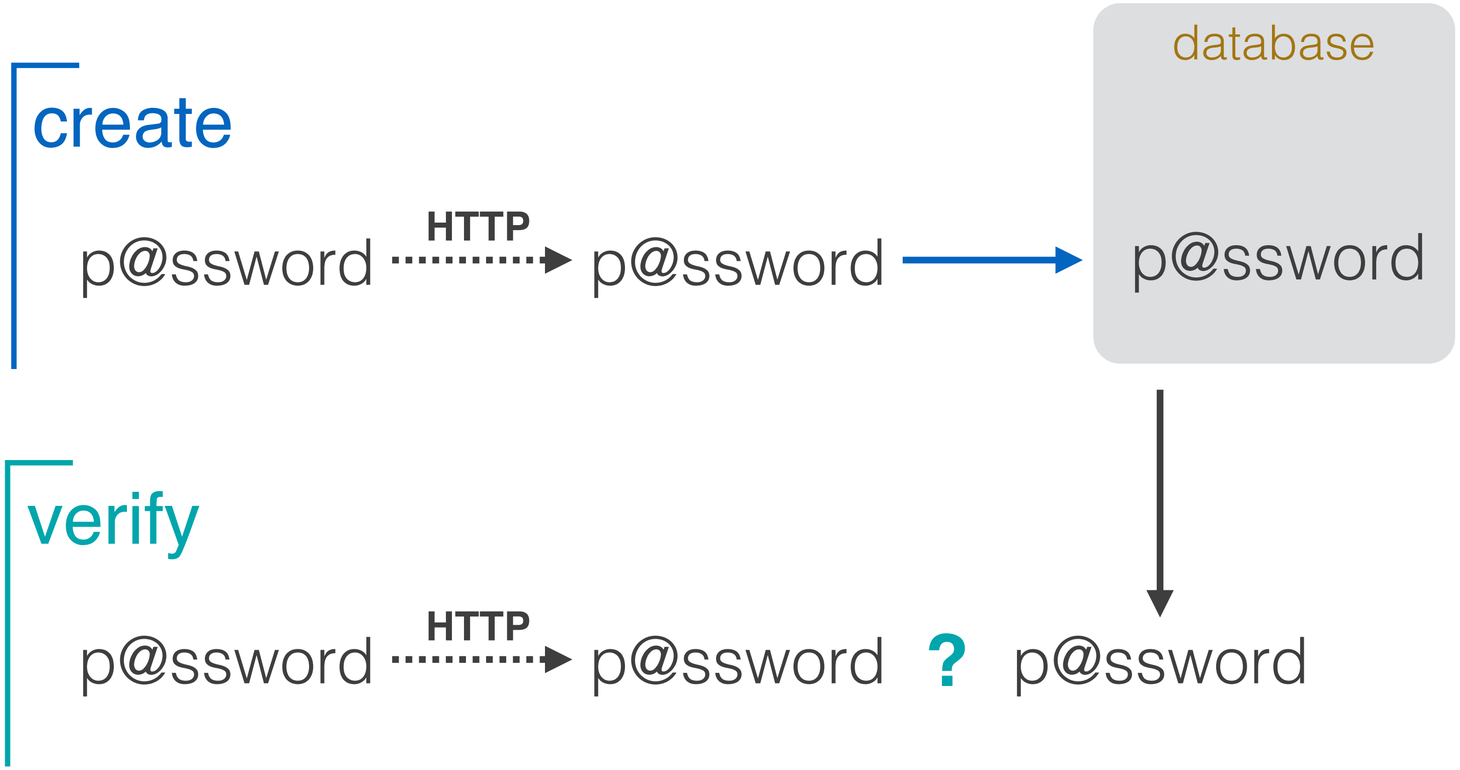
ROUND 5:
DATA THEFT
(BAD AUTH / DATA EXPOSURE)

"IDENTITY THEFT"

- Assume attacker has access to *middlemen* and *database*
- Defense vulnerable if communication or storage exposes passwords
- Bad stuff: user impersonation

PASSWORD SECURITY

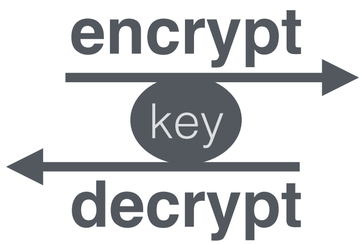
THE WORST WAY



ENCRYPTION

plaintext **scheme** **ciphertext**

word



981kje

HTTP

encryption

+ “authentic” server } SSL

HTTPS

"AUTHENTIC" SERVER

- Has an SSL certificate
- Digitally signed
- Forms a web of trust
- Self-signed in development

HTTPS IN PRACTICE

With openssl generate a private key with…

```
$ openssl genrsa -out key.pem 2048
```

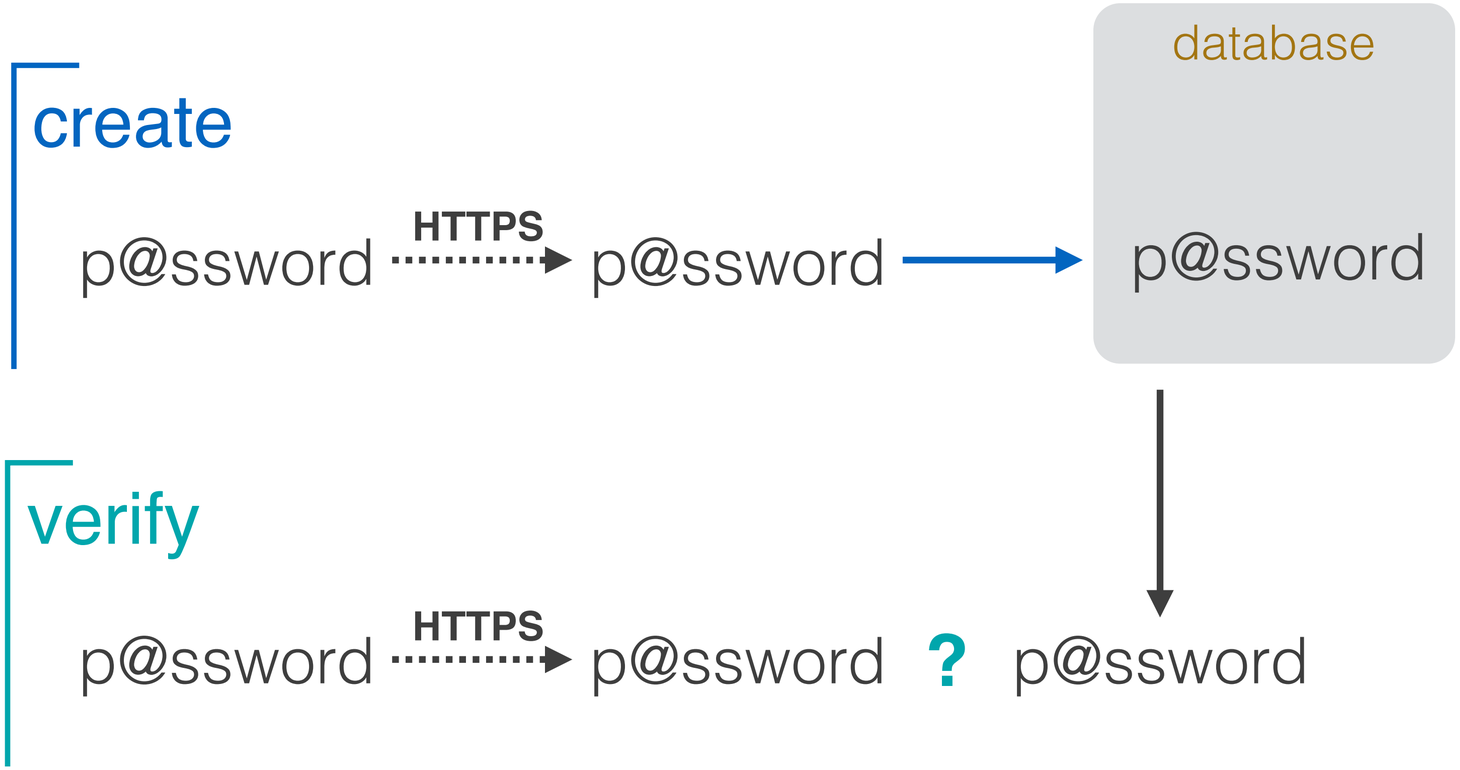
Then generate self-signed SSL certificate with…

```
$ openssl req -x509 -new -nodes -key key.pem -days 1024 -out cert.pem
```

Now use node's `https` library with this key and certificate

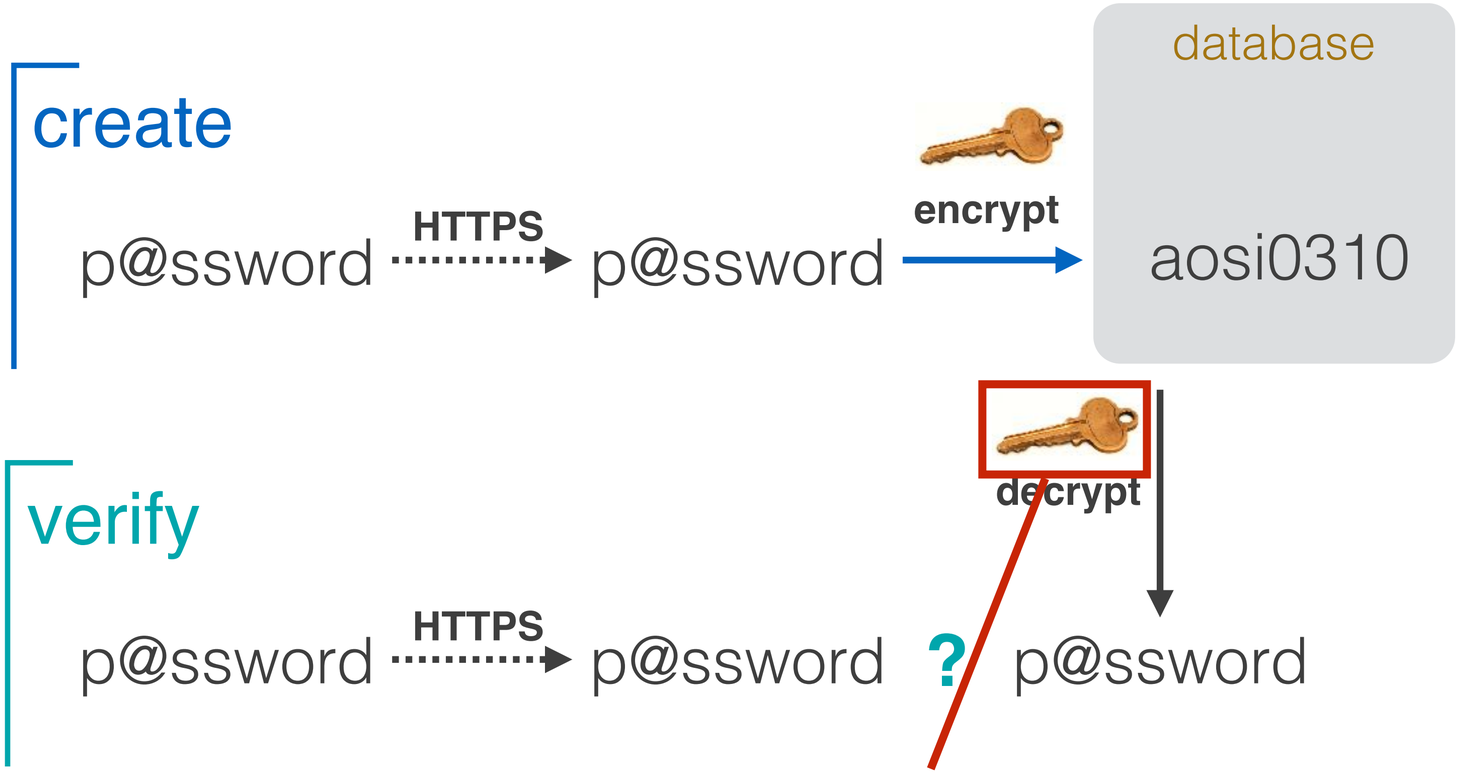
PASSWORD SECURITY

SECURE COMMUNICATION



PASSWORD SECURITY

SECURE-ISH STORAGE



HASHING

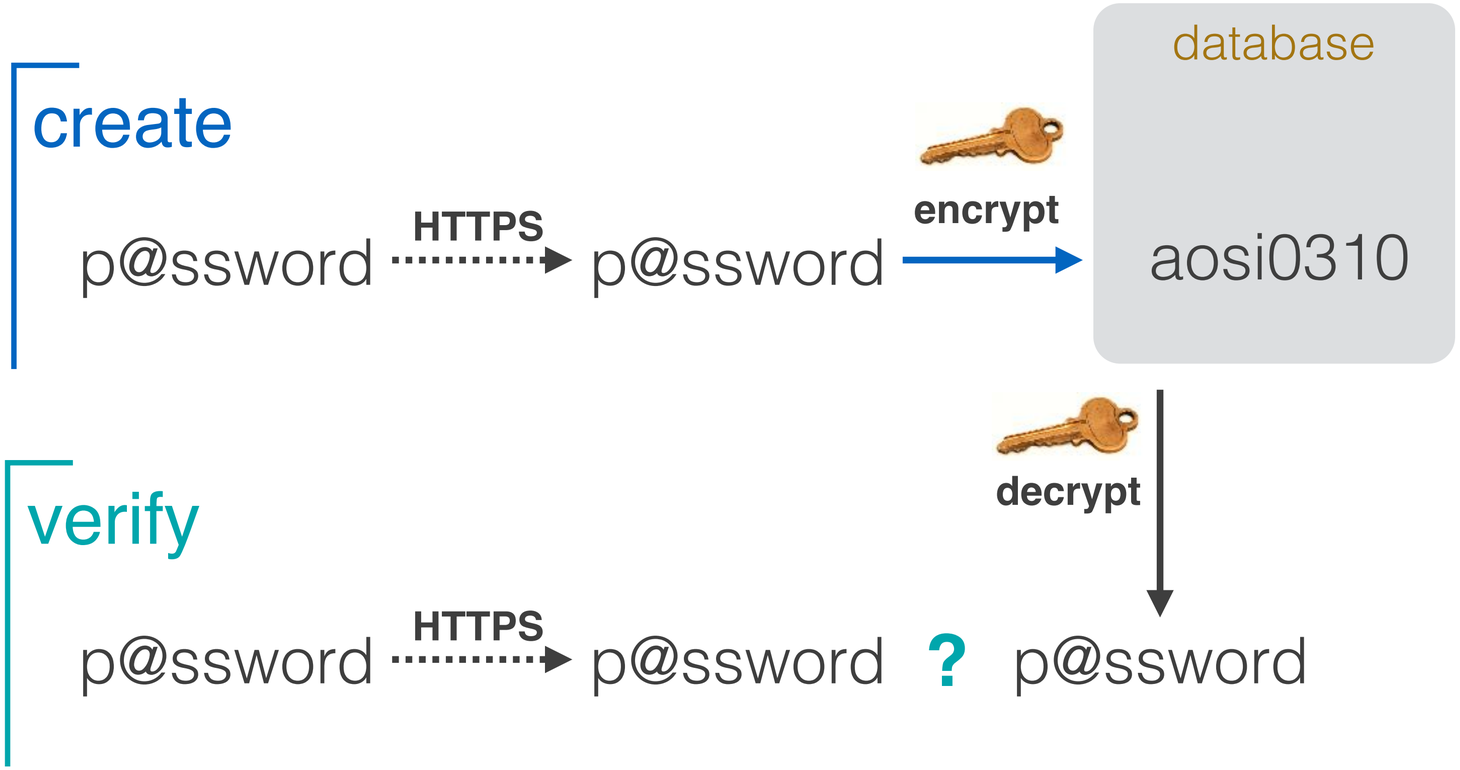
string **algorithm** **hash**

word 

jads82

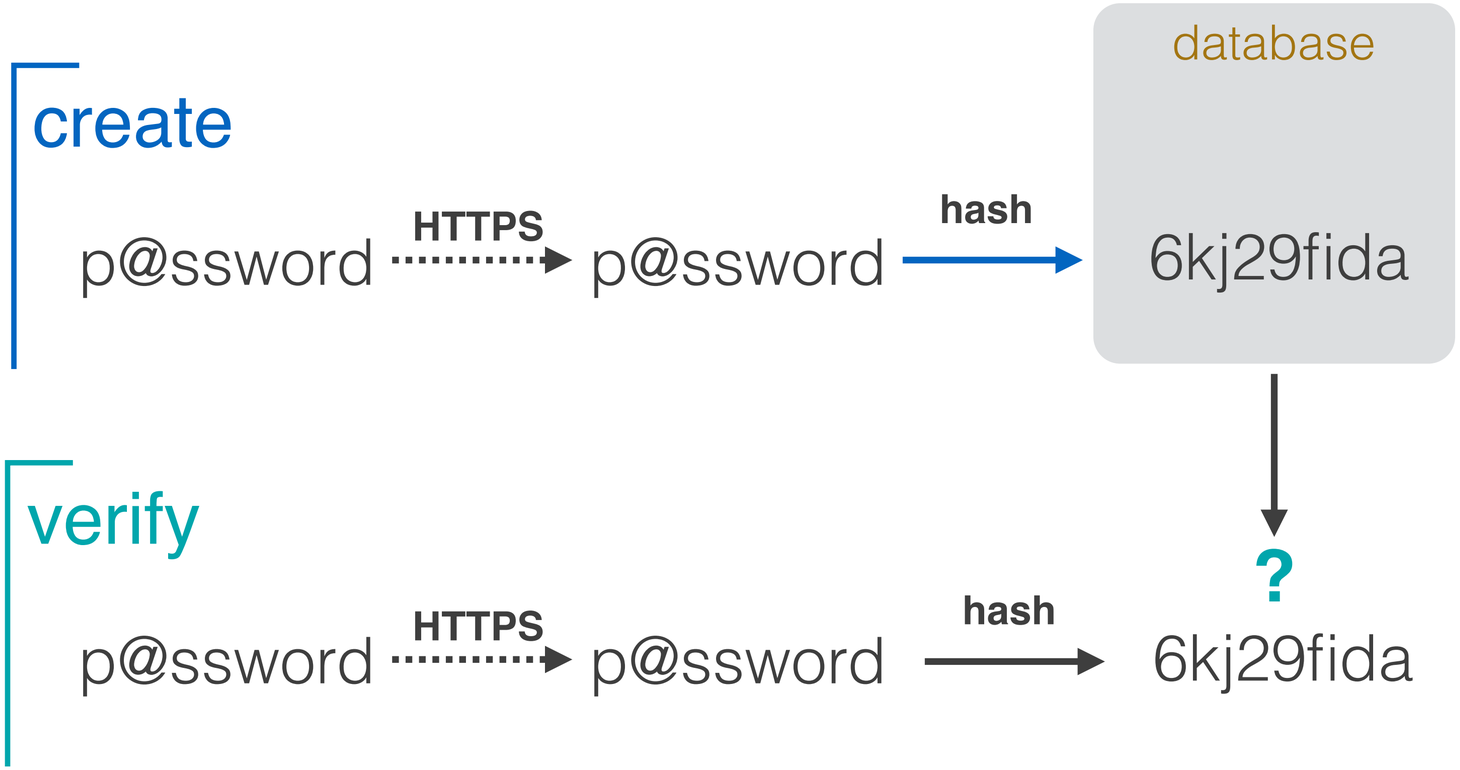
PASSWORD SECURITY

SECURE-ISH STORAGE



PASSWORD SECURITY

SECURE STORAGE



GOOD HASH KEY FUNCTION

Should be slow

Ultimately, this will hurt would-be attackers more than it will hurt you.

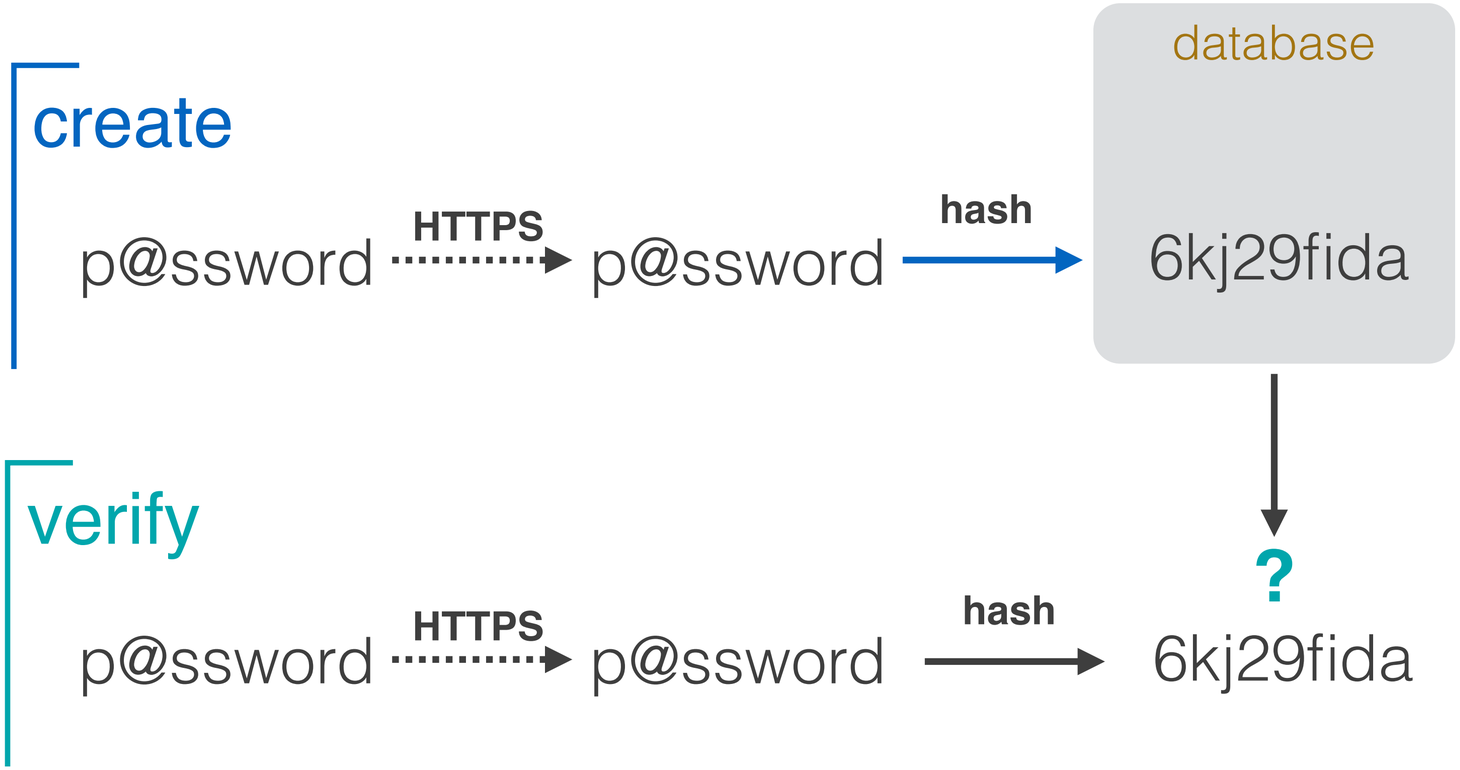
Should have a low collision frequency

Fewest possible duplicates. Or else two inputs could be indistinguishable.

PASSWORD SECURITY

Can we make it even better?

SECURE STORAGE

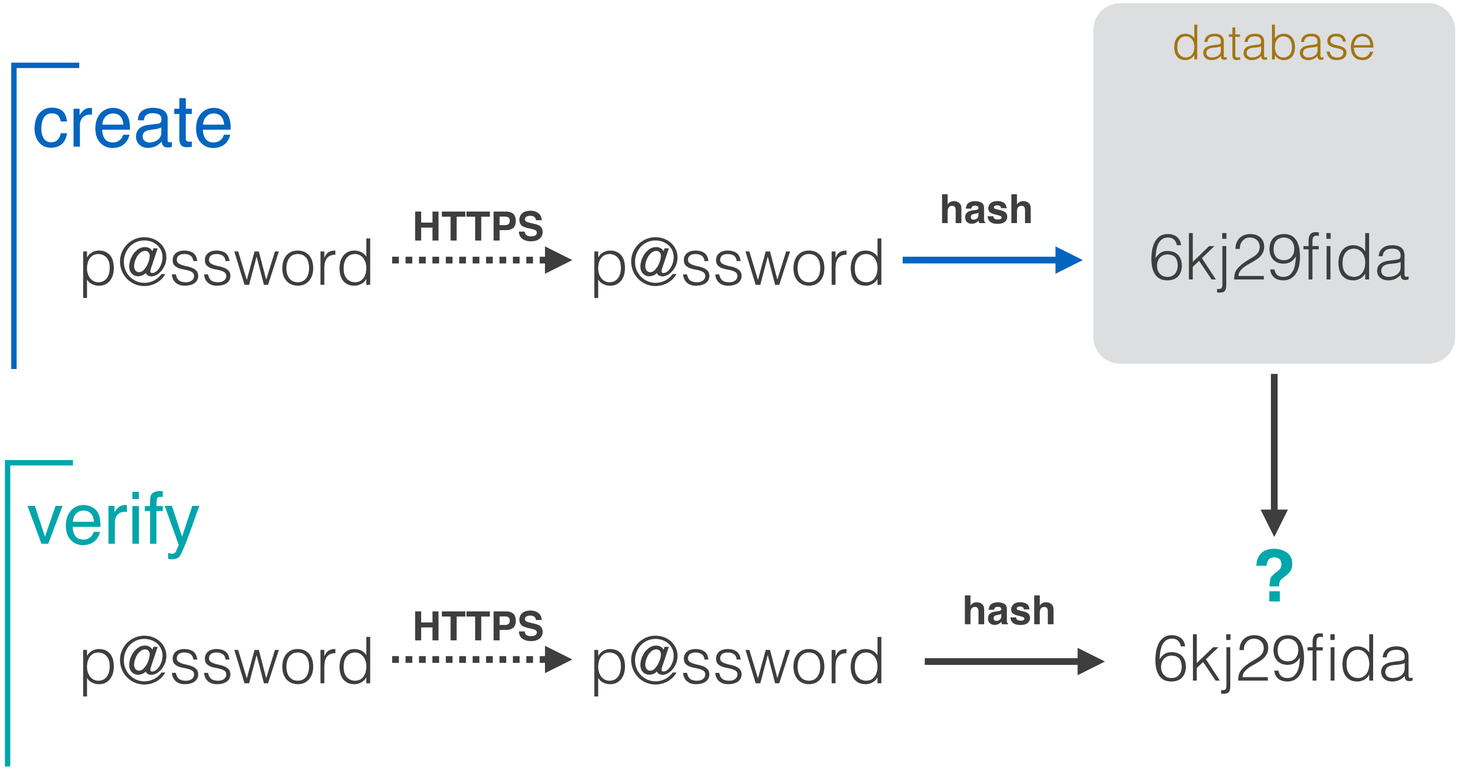


SALT

- Random string, unique to each user
- Added before hashing password
- Two users with the same password have different hashes
- Hashes are not computable ahead-of-time

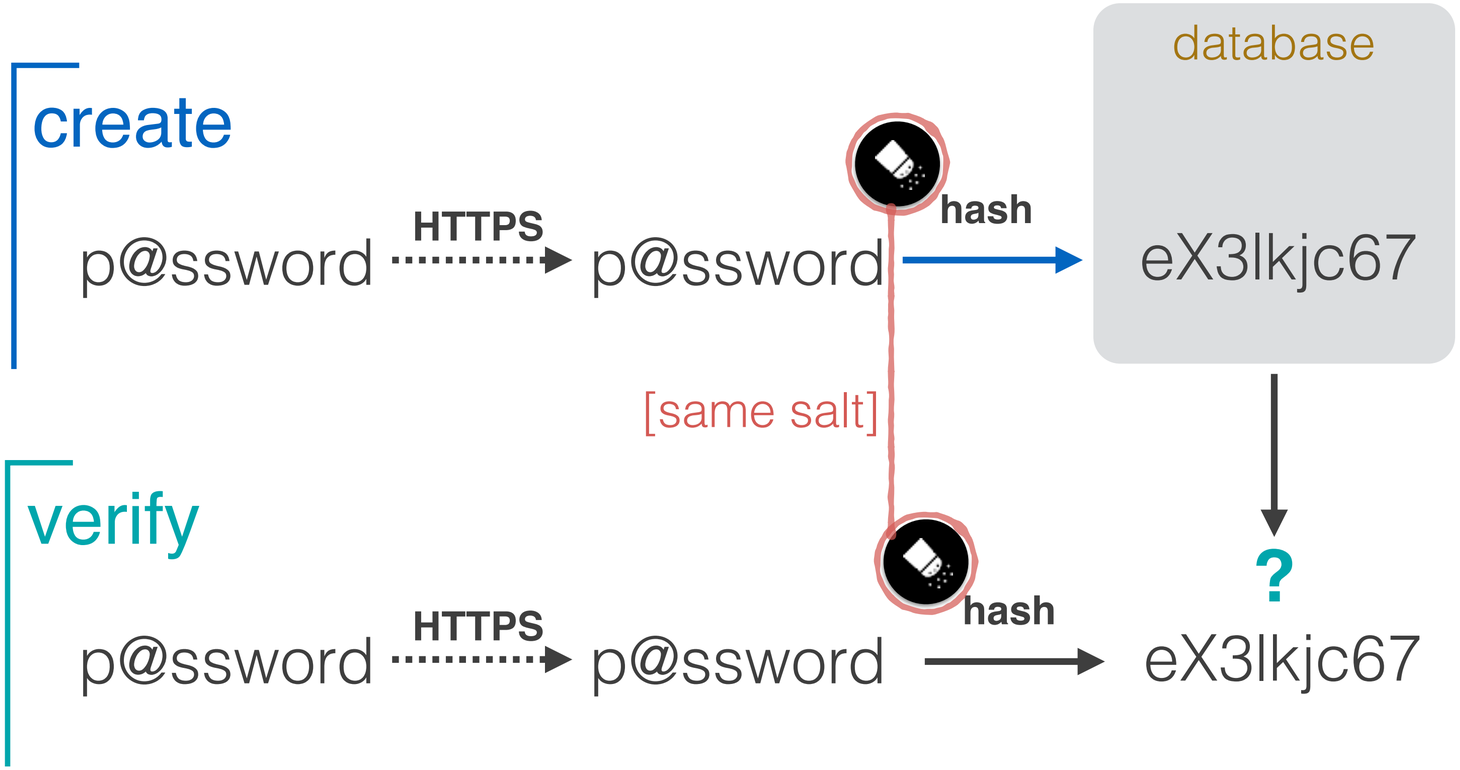
PASSWORD SECURITY

SECURE STORAGE



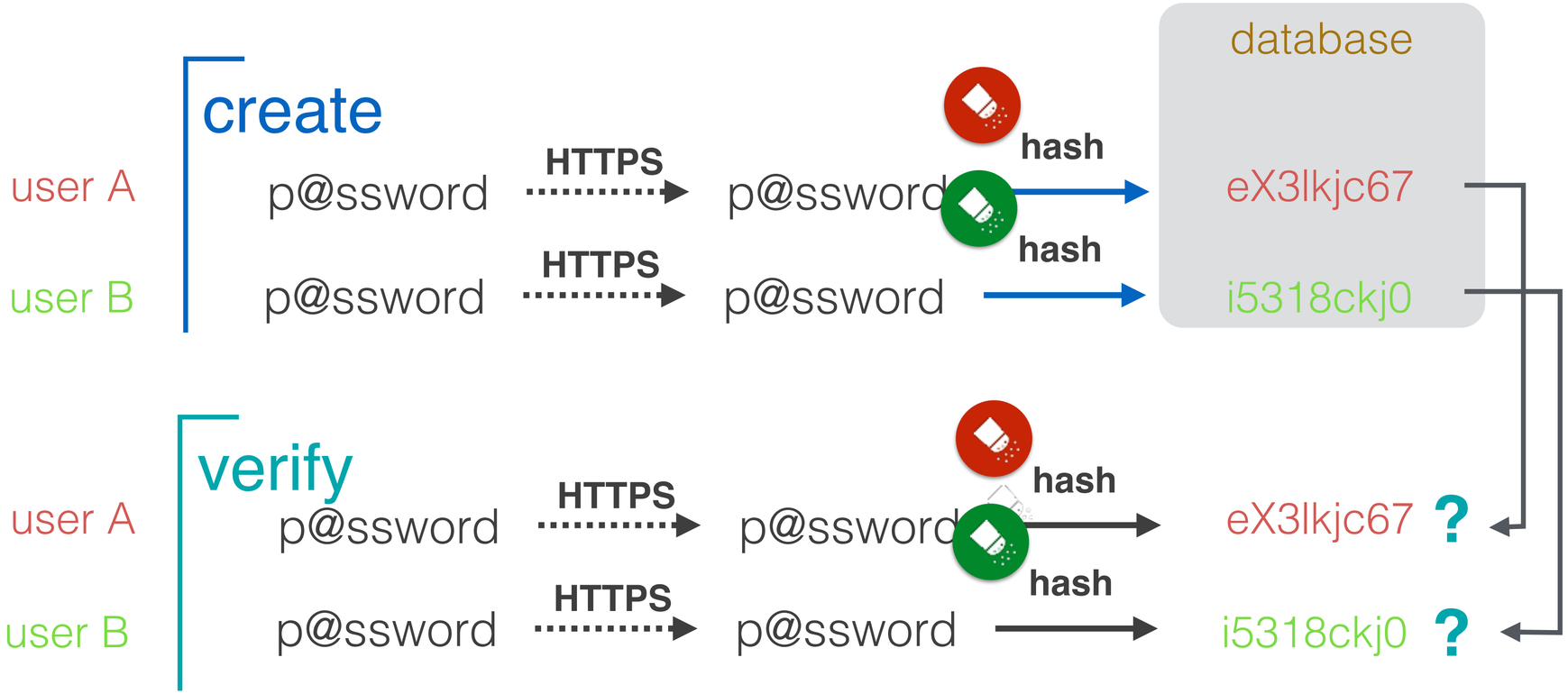
PASSWORD SECURITY

SECURE STORAGE



PASSWORD SECURITY

SECURE STORAGE



HASHING IN PRACTICE

```
// built-in node library
var crypto = require('crypto');
var salt = crypto.randomBytes(16);
var iterations = 1;
var bytes = 64;
var buffer = crypto.pbkdf2Sync('password', salt, iterations, bytes);
var hash = buffer.toString('base64');
// ...
```

```
User.generateSalt = function () {
  return crypto.randomBytes(16).toString('base64')
}

User.encryptPassword = function (plainText, salt) {
  return crypto
    .createHash('RSA-SHA256')
    .update(plainText)
    .update(salt)
    .digest('hex')
}

const setSaltAndPassword = user => {
  if (user.changed('password')) {
    user.salt = User.generateSalt()
    user.password = User.encryptPassword(user.password, user.salt)
  }
}

User.beforeCreate(setSaltAndPassword)
User.beforeUpdate(setSaltAndPassword)
```

OTHER CONSIDERATIONS

- Broken authentication flow
- Oversharing data

BROKEN AUTH

- Assume attacker is guest client
- Defense vulnerable if signup or login are improper
- Bad stuff: user impersonation, admin impersonation

OVERSHARING

- Assume attacker is non-admin client
- Defense vulnerable if attacker can see sensitive info of another
- Bad stuff: failed user privacy, NSA loves you

GOOD AUTH

- Communication is secure
- Storage is secure
- Cannot set privileges via signup
- Logging in *requires* username and password
- Data is not inadvertently shared

RECAP

FUNDAMENTAL CONCERNS

Authentication

Trusting that someone is who they say they are.

Communication

Transferring data through potentially unreliable middlemen.

Authorization

Giving resource access to the right people.

Control

Limiting or understanding the capabilities of agents.

OWASP TOP 10 (2013)

Injection	Server-side code execution
Broken Authentication	Allows for impersonation
XSS	Client-side code execution
Direct References	Access control can be circumvented
Security Misconfiguration	Vulnerable default/inherited settings
Data Exposure	Data is insecurely transmitted, stored, or simply overshared
Missing Access Control	Users can do things they shouldn't be allowed to do
XSRF	Abuse the target website's trust in the browser
Vulnerable Components	Third-party tools are vulnerable
Unvalidated Redirects	Abusable open-ended forwarding

source

weak **authentication** | poor **authorization** | insecure **communication** | breakdown of **control**

Injection	Server-side code execution
Broken Authentication	Allows for impersonation
XSS	Client-side code execution
Direct References	Access control can be circumvented
Security Misconfiguration	Vulnerable default/inherited settings
Data Exposure	Data is insecurely transmitted, stored, or simply overshared
Missing Access Control	Users can do things they shouldn't be allowed to do
XSRF	Abuse the target website's trust in the browser
Vulnerable Components	Third-party tools are vulnerable
Unvalidated Redirects	Abusable open-ended forwarding

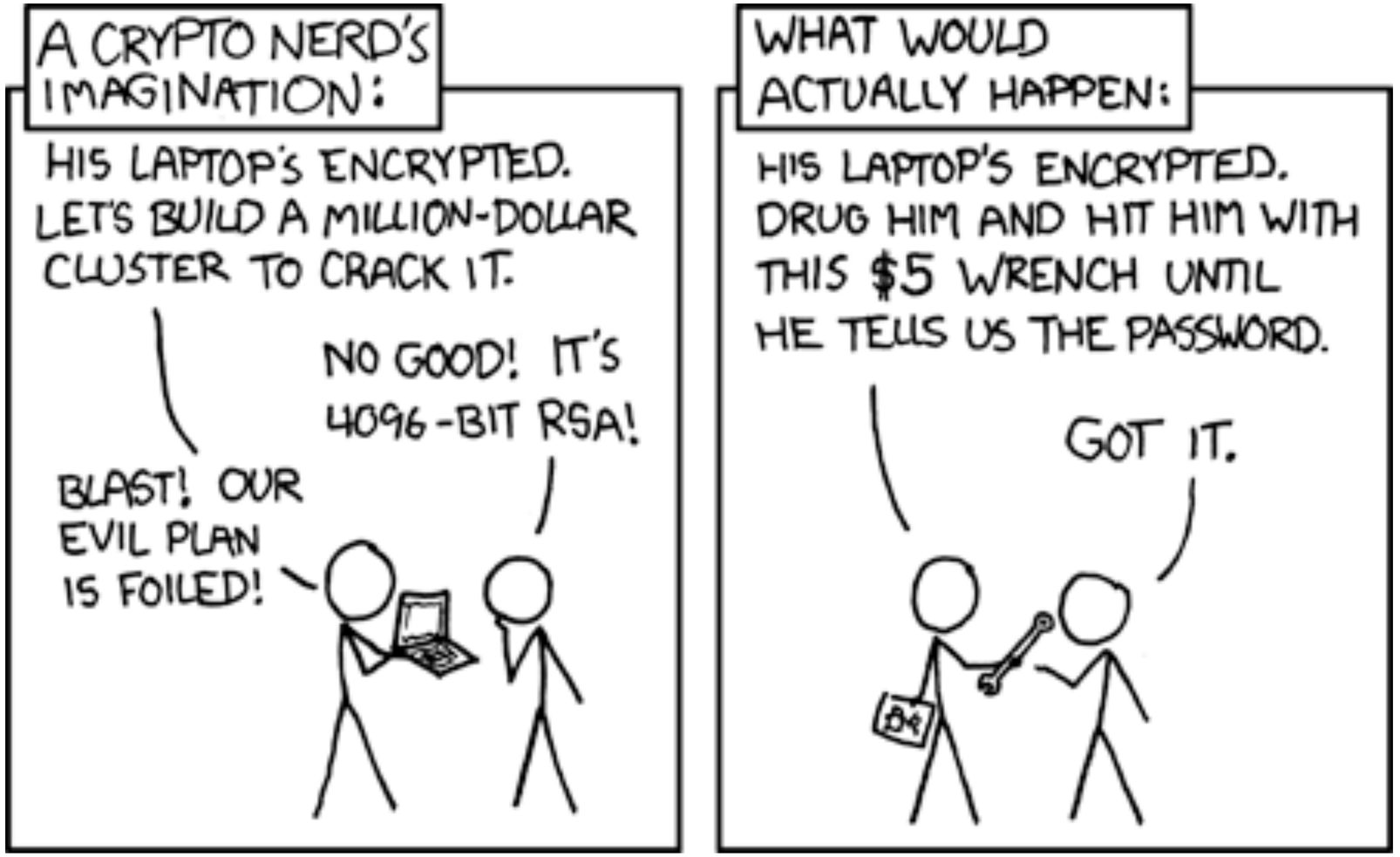
source

Injection	Server-side code execution
Broken Authentication	Allows for impersonation
XSS	Client-side code execution
Direct References	Access control can be circumvented
Security Misconfiguration	Vulnerable default/inherited settings
Data Exposure	Data is insecurely transmitted, stored, or simply overshared
Missing Access Control	Users can do things they shouldn't be allowed to do
XSRF	Abuse the target website's trust in the browser
Vulnerable Components	Third-party tools are vulnerable
Unvalidated Redirects	Abusable open-ended forwarding

[source]

Lesson: [illegible] through the server.

Injection	Server-side code execution	example
Broken Authentication	Allows for impersonation	example
XSS	Client-side code execution	example
Direct References	Access control can be circumvented	example
Security Misconfiguration	Vulnerable default/inherited settings	example
Data Exposure	Data is insecurely transmitted, stored, or simply overshared	example
Missing Access Control	Users can do things they shouldn't be allowed to do	
XSRF	Abuse the target website's trust in the browser	example
Vulnerable Components	Third-party tools are vulnerable	example
Unvalidated Redirects	Abusable open-ended forwarding	example



HTTPS://XKCD.COM/538/