# The definitive guide to form-based website authentication [closed]

Asked 16 years, 4 months ago  Modified 2 years, 7 months ago

Viewed 694k times

5511

**Closed**. This question needs to be more focused. It is not currently accepting answers.

Closed 8 years ago.

The community reviewed whether to reopen this question 3 years ago and left it closed:

> Original close reason(s) were not resolved

**This question's answers are a community effort**. Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

> **Moderator note:**
>
> This question is not a good fit for our question and answer format with the topicality rules which currently apply for Stack Overflow. We normally use a "historical lock" for such questions where the content still has value. However, the answers on this question are actively maintained and a

> historical lock doesn't permit editing of the answers. As such, a "wiki answer" lock has been applied to allow the answers to be edited. You should assume the topicality issues which are normally handled by a historical lock are present (i.e. this question not a good example of an on-topic question for Stack Overflow).

## Form-based authentication for websites

We believe that Stack Overflow should not just be a resource for very specific technical questions, but also for general guidelines on how to solve variations on common problems. "Form based authentication for websites" should be a fine topic for such an experiment.

### It should include topics such as:

- How to log in
- How to log out
- How to remain logged in
- Managing cookies (including recommended settings)
- SSL/HTTPS encryption
- How to store passwords
- Using secret questions
- Forgotten username/password functionality

- Use of nonces to prevent cross-site request forgeries (CSRF)
- OpenID
- "Remember me" checkbox
- Browser autocompletion of usernames and passwords
- Secret URLs (public URL protected by digest)
- Checking password strength
- E-mail validation
- *and much more about* form based authentication...

## It should not include things like:

- Roles and authorization
- HTTP basic authentication

## Please help us by:

1. Suggesting subtopics
2. Submitting good articles about this subject
3. Editing the official answer

**forms** **http** **security** **authentication** **language-agnostic**

Share Follow

edited Nov 11, 2021 at 19:35

community wiki
37 revs, 21 users 25%
Michiel de Mare

54 Why exclude HTTP Basic Authentication? It can work in HTML Forms via Ajax: peej.co.uk/articles/http-auth-with-html-forms.html – system PAUSE Jun 25, 2009 at 22:50

56 HTTP Basic Auth has the property of being (comparatively) difficult to make a browser forget. It's also horribly insecure if you don't use it with SSL to secure the connection (i.e., HTTPS). – Donal Fellows Jun 15, 2010 at 10:53

24 I think it'd be worth talking about sessions (including fixation and hijacking) cookies (the secure and http only flags) HTTP based SSO – symcbean Jun 20, 2011 at 13:08

29 The super-useful `HttpOnly` cookie flag, which prevents JavaScript-based cookie theft (a subset of XSS attacks), should be mentioned somewhere too. – Alan H. Aug 8, 2011 at 20:41

81 Wow. Lengthy answers, dozens of upvotes for some of them, yet nobody mentions the common mistake of serving login forms over HTTP. I've even argued with people who said "but it submits to https://..." and only got blank stares when I asked if they were sure an attacker didn't rewrite the non-encrypted page the form was served over. – dzuelke Nov 8, 2012 at 22:34

Comments disabled on deleted / locked posts / reviews |

11 Answers Sorted by: Highest score (default)

# PART I: How To Log In

We'll assume you already know how to build a login+password HTML form which POSTs the values to a script on the server side for authentication. The sections below will deal with patterns for sound practical auth, and how to avoid the most common security pitfalls.

**To HTTPS or not to HTTPS?**

Unless the connection is already secure (that is, tunneled through HTTPS using SSL/TLS), your login form values will be sent in cleartext, which allows anyone eavesdropping on the line between browser and web server will be able to read logins as they pass through. This type of wiretapping is done routinely by governments, but in general, we won't address 'owned' wires other than to say this: Just use HTTPS.

In essence, the only **practical** way to protect against wiretapping/packet sniffing during login is by using HTTPS or another certificate-based encryption scheme (for example, TLS) or a proven & tested challenge-response scheme (for example, the Diffie-Hellman-based SRP). *Any other method can be easily circumvented* by an eavesdropping attacker.

Of course, if you are willing to get a little bit impractical, you could also employ some form of two-factor authentication scheme (e.g. the Google Authenticator app, a physical 'cold war style' codebook, or an RSA key generator dongle). If applied correctly, this could work even with an unsecured connection, but it's hard to

imagine that a dev would be willing to implement two-factor auth but not SSL.

### (Do not) Roll-your-own JavaScript encryption/hashing

Given the perceived (though now avoidable) cost and technical difficulty of setting up an SSL certificate on your website, some developers are tempted to roll their own in-browser hashing or encryption schemes in order to avoid passing cleartext logins over an unsecured wire.

While this is a noble thought, it is essentially useless (and can be a security flaw) unless it is combined with one of the above - that is, either securing the line with strong encryption or using a tried-and-tested challenge-response mechanism (if you don't know what that is, just know that it is one of the most difficult to prove, most difficult to design, and most difficult to implement concepts in digital security).

While it is true that hashing the password *can be* effective against **password disclosure**, it is vulnerable to replay attacks, Man-In-The-Middle attacks / hijackings (if an attacker can inject a few bytes into your unsecured HTML page before it reaches your browser, they can simply comment out the hashing in the JavaScript), or brute-force attacks (since you are handing the attacker both username, salt and hashed password).

### CAPTCHAS against humanity

CAPTCHA is meant to thwart one specific category of attack: automated dictionary/brute force trial-and-error with no human operator. There is no doubt that this is a real threat, however, there are ways of dealing with it seamlessly that don't require a CAPTCHA, specifically properly designed server-side login throttling schemes - we'll discuss those later.

Know that CAPTCHA implementations are not created alike; they often aren't human-solvable, most of them are actually ineffective against bots, all of them are ineffective against cheap third-world labor (according to OWASP, the current sweatshop rate is $12 per 500 tests), and some implementations may be technically illegal in some countries (see OWASP Authentication Cheat Sheet). If you must use a CAPTCHA, use Google's reCAPTCHA, since it is OCR-hard by definition (since it uses already OCR-misclassified book scans) and tries very hard to be user-friendly.

Personally, I tend to find CAPTCHAS annoying, and use them only as a last resort when a user has failed to log in a number of times and throttling delays are maxed out. This will happen rarely enough to be acceptable, and it strengthens the system as a whole.

**Storing Passwords / Verifying logins**

This may finally be common knowledge after all the highly-publicized hacks and user data leaks we've seen in recent years, but it has to be said: Do not store passwords in cleartext in your database. User databases

are routinely hacked, leaked or gleaned through SQL injection, and if you are storing raw, plaintext passwords, that is instant game over for your login security.

So if you can't store the password, how do you check that the login+password combination POSTed from the login form is correct? The answer is hashing using a key derivation function. Whenever a new user is created or a password is changed, you take the password and run it through a KDF, such as Argon2, bcrypt, scrypt or PBKDF2, turning the cleartext password ("correcthorsebatterystaple") into a long, random-looking string, which is a lot safer to store in your database. To verify a login, you run the same hash function on the entered password, this time passing in the salt and compare the resulting hash string to the value stored in your database. Argon2, bcrypt and scrypt store the salt with the hash already. Check out this article on sec.stackexchange for more detailed information.

The reason a salt is used is that hashing in itself is not sufficient -- you'll want to add a so-called 'salt' to protect the hash against rainbow tables. A salt effectively prevents two passwords that exactly match from being stored as the same hash value, preventing the whole database being scanned in one run if an attacker is executing a password guessing attack.

A cryptographic hash should not be used for password storage because user-selected passwords are not strong enough (i.e. do not usually contain enough entropy) and

a password guessing attack could be completed in a relatively short time by an attacker with access to the hashes. This is why KDFs are used - these effectively "stretch the key", which means that every password guess an attacker makes causes multiple repetitions of the hash algorithm, for example 10,000 times, which causes the attacker to guess the password 10,000 times slower.

**Session data - "You are logged in as Spiderman69"**

Once the server has verified the login and password against your user database and found a match, the system needs a way to remember that the browser has been authenticated. This fact should only ever be stored server side in the session data.

> If you are unfamiliar with session data, here's how it works: A single randomly-generated string is stored in an expiring cookie and used to reference a collection of data - the session data - which is stored on the server. If you are using an MVC framework, this is undoubtedly handled already.

If at all possible, make sure the session cookie has the secure and HTTP Only flags set when sent to the browser. The HttpOnly flag provides some protection against the cookie being read through XSS attack. The secure flag ensures that the cookie is only sent back via HTTPS, and therefore protects against network sniffing

attacks. The value of the cookie should not be predictable. Where a cookie referencing a non-existent session is presented, its value should be replaced immediately to prevent session fixation.

Session state can also be maintained on the client side. This is achieved by using techniques like JWT (JSON Web Token).

## PART II: How To Remain Logged In - The Infamous "Remember Me" Checkbox

Persistent Login Cookies ("remember me" functionality) are a danger zone; on the one hand, they are entirely as safe as conventional logins when users understand how to handle them; and on the other hand, they are an enormous security risk in the hands of careless users, who may use them on public computers and forget to log out, and who may not know what browser cookies are or how to delete them.

Personally, I like persistent logins for the websites I visit on a regular basis, but I know how to handle them safely. If you are positive that your users know the same, you can use persistent logins with a clean conscience. If not - well, then you may subscribe to the philosophy that users who are careless with their login credentials brought it upon themselves if they get hacked. It's not like we go to our user's houses and tear off all those facepalm-

inducing Post-It notes with passwords they have lined up on the edge of their monitors, either.

Of course, some systems can't afford to have *any* accounts hacked; for such systems, there is no way you can justify having persistent logins.

**If you DO decide to implement persistent login cookies, this is how you do it:**

1. First, take some time to read Paragon Initiative's article on the subject. You'll need to get a bunch of elements right, and the article does a great job of explaining each.
2. And just to reiterate one of the most common pitfalls, **DO NOT STORE THE PERSISTENT LOGIN COOKIE (TOKEN) IN YOUR DATABASE, ONLY A HASH OF IT!** The login token is Password Equivalent, so if an attacker got their hands on your database, they could use the tokens to log in to any account, just as if they were cleartext login-password combinations. Therefore, use hashing (according to https://security.stackexchange.com/a/63438/5002 a weak hash will do just fine for this purpose) when storing persistent login tokens.

## PART III: Using Secret Questions

**Don't implement 'secret questions'**. The 'secret questions' feature is a security anti-pattern. Read the paper from link number 4 from the MUST-READ list. You

can ask Sarah Palin about that one, after her Yahoo! email account got hacked during a previous presidential campaign because the answer to her security question was... "Wasilla High School"!

Even with user-specified questions, it is highly likely that most users will choose either:

- A 'standard' secret question like mother's maiden name or favorite pet
- A simple piece of trivia that anyone could lift from their blog, LinkedIn profile, or similar
- Any question that is easier to answer than guessing their password. Which, for any decent password, is every question you can imagine

**In conclusion, security questions are inherently insecure in virtually all their forms and variations, and should not be employed in an authentication scheme for any reason.**

The true reason why security questions even exist in the wild is that they conveniently save the cost of a few support calls from users who can't access their email to get to a reactivation code. This at the expense of security and Sarah Palin's reputation. Worth it? Probably not.

## PART IV: Forgotten Password Functionality

I already mentioned why you should **never use security questions** for handling forgotten/lost user passwords; it also goes without saying that you should never e-mail users their actual passwords. There are at least two more all-too-common pitfalls to avoid in this field:

1. Don't *reset* a forgotten password to an autogenerated strong password - such passwords are notoriously hard to remember, which means the user must either change it or write it down - say, on a bright yellow Post-It on the edge of their monitor. Instead of setting a new password, just let users pick a new one right away - which is what they want to do anyway. (An exception to this might be if the users are universally using a password manager to store/manage passwords that would normally be impossible to remember without writing it down).
2. Always hash the lost password code/token in the database. ***AGAIN***, this code is another example of a Password Equivalent, so it MUST be hashed in case an attacker got their hands on your database. When a lost password code is requested, send the plaintext code to the user's email address, then hash it, save the hash in your database -- and *throw away the original*. Just like a password or a persistent login token.

A final note: always make sure your interface for entering the 'lost password code' is at least as secure as your login form itself, or an attacker will simply use this to gain access instead. Making sure you generate very long 'lost

password codes' (for example, 16 case-sensitive alphanumeric characters) is a good start, but consider adding the same throttling scheme that you do for the login form itself.

## PART V: Checking Password Strength

First, you'll want to read this small article for a reality check: The 500 most common passwords

Okay, so maybe the list isn't the *canonical* list of most common passwords on *any* system *anywhere ever*, but it's a good indication of how poorly people will choose their passwords when there is no enforced policy in place. Plus, the list looks frighteningly close to home when you compare it to publicly available analyses of recently stolen passwords.

So: With no minimum password strength requirements, 2% of users use one of the top 20 most common passwords. Meaning: if an attacker gets just 20 attempts, 1 in 50 accounts on your website will be crackable.

Thwarting this requires calculating the entropy of a password and then applying a threshold. The National Institute of Standards and Technology (NIST) Special Publication 800-63 has a set of very good suggestions. That, when combined with a dictionary and keyboard layout analysis (for example, 'qwertyuiop' is a bad password), can reject 99% of all poorly selected passwords at a level of 18 bits of entropy. Simply

calculating password strength and showing a visual strength meter to a user is good, but insufficient. Unless it is enforced, a lot of users will most likely ignore it.

And for a refreshing take on user-friendliness of high-entropy passwords, Randall Munroe's Password Strength xkcd is highly recommended.

Utilize Troy Hunt's Have I Been Pwned API to check users passwords against passwords compromised in public data breaches.

## PART VI: Much More - Or: Preventing Rapid-Fire Login Attempts

First, have a look at the numbers: Password Recovery Speeds - How long will your password stand up

If you don't have the time to look through the tables in that link, here's the list of them:

1. It takes *virtually no time* to crack a weak password, even if you're cracking it with an abacus
2. It takes *virtually no time* to crack an alphanumeric 9-character password if it is **case insensitive**
3. It takes *virtually no time* to crack an intricate, symbols-and-letters-and-numbers, upper-and-lowercase password if it is **less than 8 characters long** (a desktop PC can search the entire keyspace up to 7 characters in a matter of days or even hours)

4. **It would, however, take an inordinate amount of time to crack even a 6-character password, *if you were limited to one attempt per second!***

So what can we learn from these numbers? Well, lots, but we can focus on the most important part: the fact that preventing large numbers of rapid-fire successive login attempts (ie. the *brute force* attack) really isn't that difficult. But preventing it *right* isn't as easy as it seems.

Generally speaking, you have three choices that are all effective against brute-force attacks *(and dictionary attacks, but since you are already employing a strong passwords policy, they shouldn't be an issue)*:

- Present a **CAPTCHA** after N failed attempts (annoying as hell and often ineffective -- but I'm repeating myself here)
- **Locking accounts** and requiring email verification after N failed attempts (this is a DoS attack waiting to happen)
- And finally, **login throttling**: that is, setting a time delay between attempts after N failed attempts (yes, DoS attacks are still possible, but at least they are far less likely and a lot more complicated to pull off).

**Best practice #1:** A short time delay that increases with the number of failed attempts, like:

- 1 failed attempt = no delay
- 2 failed attempts = 2 sec delay

- 3 failed attempts = 4 sec delay
- 4 failed attempts = 8 sec delay
- 5 failed attempts = 16 sec delay
- etc.

DoS attacking this scheme would be very impractical, since the resulting lockout time is slightly larger than the sum of the previous lockout times.

> To clarify: The delay is *not* a delay before returning the response to the browser. It is more like a timeout or refractory period during which login attempts to a specific account or from a specific IP address will not be accepted or evaluated at all. That is, correct credentials will not return in a successful login, and incorrect credentials will not trigger a delay increase.

**Best practice #2:** A medium length time delay that goes into effect after N failed attempts, like:

- 1-4 failed attempts = no delay
- 5 failed attempts = 15-30 min delay

DoS attacking this scheme would be quite impractical, but certainly doable. Also, it might be relevant to note that such a long delay can be very annoying for a legitimate user. Forgetful users will dislike you.

**Best practice #3:** Combining the two approaches - either a fixed, short time delay that goes into effect after N failed attempts, like:

- 1-4 failed attempts = no delay
- 5+ failed attempts = 20 sec delay

Or, an increasing delay with a fixed upper bound, like:

- 1 failed attempt = 5 sec delay
- 2 failed attempts = 15 sec delay
- 3+ failed attempts = 45 sec delay

This final scheme was taken from the OWASP best-practices suggestions (link 1 from the MUST-READ list) and should be considered best practice, even if it is admittedly on the restrictive side.

> *As a rule of thumb, however, I would say: the stronger your password policy is, the less you have to bug users with delays. If you require strong (case-sensitive alphanumerics + required numbers and symbols) 9+ character passwords, you could give the users 2-4 non-delayed password attempts before activating the throttling.*

DoS attacking this final login throttling scheme would be ***very*** impractical. And as a final touch, always allow persistent (cookie) logins (and/or a CAPTCHA-verified

login form) to pass through, so legitimate users won't even be delayed *while the attack is in progress*. That way, the very impractical DoS attack becomes an *extremely* impractical attack.

Additionally, it makes sense to do more aggressive throttling on admin accounts, since those are the most attractive entry points

## PART VII: Distributed Brute Force Attacks

Just as an aside, more advanced attackers will try to circumvent login throttling by 'spreading their activities':

- Distributing the attempts on a botnet to prevent IP address flagging
- Rather than picking one user and trying the 50.000 most common passwords (which they can't, because of our throttling), they will pick THE most common password and try it against 50.000 users instead. That way, not only do they get around maximum-attempts measures like CAPTCHAs and login throttling, their chance of success increases as well, since the number 1 most common password is far more likely than number 49.995
- Spacing the login requests for each user account, say, 30 seconds apart, to sneak under the radar

Here, the best practice would be **logging the number of failed logins, system-wide**, and using a running average of your site's bad-login frequency as the basis for an upper limit that you then impose on all users.

Too abstract? Let me rephrase:

Say your site has had an average of 120 bad logins per day over the past 3 months. Using that (running average), your system might set the global limit to 3 times that -- ie. 360 failed attempts over a 24 hour period. Then, if the total number of failed attempts across all accounts exceeds that number within one day (or even better, monitor the rate of acceleration and trigger on a calculated threshold), it activates system-wide login throttling - meaning short delays for ALL users (still, with the exception of cookie logins and/or backup CAPTCHA logins).

I also posted a question with more details and a really good discussion of how to avoid tricky pitfals in fending off distributed brute force attacks

## PART VIII: Two-Factor Authentication and Authentication Providers

Credentials can be compromised, whether by exploits, passwords being written down and lost, laptops with keys being stolen, or users entering logins into phishing sites. Logins can be further protected with two-factor authentication, which uses out-of-band factors such as

single-use codes received from a phone call, SMS message, app, or dongle. Several providers offer two-factor authentication services.

Authentication can be completely delegated to a single-sign-on service, where another provider handles collecting credentials. This pushes the problem to a trusted third party. Google and Twitter both provide standards-based SSO services, while Facebook provides a similar proprietary solution.

## MUST-READ LINKS About Web Authentication

1. OWASP Guide To Authentication / OWASP Authentication Cheat Sheet
2. Dos and Don'ts of Client Authentication on the Web (very readable MIT research paper)
3. Wikipedia: HTTP cookie
4. Personal knowledge questions for fallback authentication: Security questions in the era of Facebook (very readable Berkeley research paper)

Share Improve this answer Follow

edited Nov 11, 2021 at 12:03

community wiki
53 revs, 38 users 29%
Jens Roland

72 Well, I don't really agree with the Captcha part, yes Captchas are annoying and they can be broken (except recaptcha but this is barely solvable by humans!) but this is exactly like saying don't use a spam filter because it has less than 0.1% false negatives .. this very site uses Captchas, they are not perfect but they cut a considerable amount of spam and there's simply no good alternative to them – Waleed Eissa Apr 27, 2009 at 2:44

259 @Jeff: I'm sorry to hear that you have issues with my reply. I didn't know there was a debate on Meta about this answer, I would have gladly edited it myself if you'd asked me to. And deleting my posts just deleted 1200 reputation from my account, which hurts :( – Jens Roland Jun 19, 2011 at 17:00

14 "After sending the authentication tokens, the system needs a way to remember that you have been authenticated - this fact should only ever be stored serverside in the session data. A cookie can be used to reference the session data." Not quite. You can (and should, for stateless servers!) use a cryptographically signed cookie. That's impossible to forge, doesn't tie up server resources, and doesn't need sticky sessions or other shenanigans. – Martin Probst Aug 8, 2011 at 12:32

12 "a desktop PC can search the FULL KEYSPACE up to 7 characters in less than 90 days" A machine with a recent GPU can search the full 7 char keyspace in less than 1 day. A top of the line GPU can manage 1 billion hashes per second. golubev.com/hashgpu.htm This leads to some conclusions about password storage which aren't directly addressed. – Frank Farmer Aug 8, 2011 at 23:06

6 @MikeMike: "..and loop through them in php" -- why not just select the row in SQL? `SELECT * FROM LoginTokens WHERE UserID=[userid from cookie] AND HashedToken=[hash(token from cookie)]` should

work just fine (remember to use prepared statements / stored procedures for the SQL though) – Jens Roland Mar 11, 2013 at 20:39

435

# Definitive Article

## Sending credentials

The only practical way to send credentials 100% securely is by using SSL. Using JavaScript to hash the password is not safe. Common pitfalls for client-side password hashing:

- If the connection between the client and server is unencrypted, everything you do is vulnerable to man-in-the-middle attacks. An attacker could replace the incoming javascript to break the hashing or send all credentials to their server, they could listen to client responses and impersonate the users perfectly, etc. etc. SSL with trusted Certificate Authorities is designed to prevent MitM attacks.
- The hashed password received by the server is less secure if you don't do additional, redundant work on the server.

There's another secure method called **SRP**, but it's patented (although it is freely licensed) and there are few good implementations available.

## Storing passwords

Don't ever store passwords as plaintext in the database. Not even if you don't care about the security of your own site. Assume that some of your users will reuse the password of their online bank account. So, store the hashed password, and throw away the original. And make sure the password doesn't show up in access logs or application logs. OWASP recommends the use of Argon2 as your first choice for new applications. If this is not available, PBKDF2 or scrypt should be used instead. And finally if none of the above are available, use bcrypt.

Hashes by themselves are also insecure. For instance, identical passwords mean identical hashes--this makes hash lookup tables an effective way of cracking lots of passwords at once. Instead, store the **salted** hash. A salt is a string appended to the password prior to hashing - use a different (random) salt per user. The salt is a public value, so you can store them with the hash in the database. See here for more on this.

This means that you can't send the user their forgotten passwords (because you only have the hash). Don't reset the user's password unless you have authenticated the user (users must prove that they are able to read emails sent to the stored (and validated) email address.)

## Security questions

Security questions are insecure - avoid using them. Why? Anything a security question does, a password does

better. Read ***PART III: Using Secret Questions*** in @Jens Roland answer here in this wiki.

## Session cookies

After the user logs in, the server sends the user a session cookie. The server can retrieve the username or id from the cookie, but nobody else can generate such a cookie (TODO explain mechanisms).

Cookies can be hijacked: they are only as secure as the rest of the client's machine and other communications. They can be read from disk, sniffed in network traffic, lifted by a cross-site scripting attack, phished from a poisoned DNS so the client sends their cookies to the wrong servers. Don't send persistent cookies. Cookies should expire at the end of the client session (browser close or leaving your domain).

If you want to autologin your users, you can set a persistent cookie, but it should be distinct from a full-session cookie. You can set an additional flag that the user has auto-logged in, and needs to log in for real for sensitive operations. This is popular with shopping sites that want to provide you with a seamless, personalized shopping experience but still protect your financial details. For example, when you return to visit Amazon, they show you a page that looks like you're logged in, but when you go to place an order (or change your shipping address, credit card etc.), they ask you to confirm your password.

Financial websites such as banks and credit cards, on the other hand, only have sensitive data and should not allow auto-login or a low-security mode.

## List of external resources

- Dos and Don'ts of Client Authentication on the Web (PDF)
  21 page academic article with many great tips.
- Ask YC: Best Practices for User Authentication
  Forum discussion on the subject
- You're Probably Storing Passwords Incorrectly
  Introductory article about storing passwords
- Discussion: Coding Horror: You're Probably Storing Passwords Incorrectly
  Forum discussion about a Coding Horror article.
- Never store passwords in a database!
  Another warning about storing passwords in the database.
- Password cracking
  Wikipedia article on weaknesses of several password hashing schemes.
- Enough With The Rainbow Tables: What You Need To Know About Secure Password Schemes
  Discussion about rainbow tables and how to defend against them, and against other threads. Includes extensive discussion.

Share Improve this answer Follow

edited Nov 19, 2018 at 13:54

community wiki
21 revs, 14 users 35%
Michiel de Mare

1 Given the recent MITM vulnerability surrounding signed SSL certificates (blog.startcom.org/?p=145) so a combination of SSL and some kind of Challenge response authentication (There are alternatives to SRP) is probably a better solution. – Kevin Loney Jan 20, 2009 at 0:20

a lot of this stuff is situational. i tend not to use session cookies at all. cookies getting hijacked is almost always the servers fault. man in the middle / packet sniffing arent that common – Shawn Jan 28, 2009 at 5:26

BCrypt Nuget package : nuget.org/List/Packages/BCrypt – Fabian Vilers Aug 9, 2011 at 14:10

1 Note 1 about this answer: it is a draft, to be edited as a wiki. If you can edit this, you're welcome to. – Peter Mortensen Aug 18, 2012 at 11:23

SRP is specific to the presence of several parties if I understand well – Diagathe Josué Aug 27, 2018 at 1:28

170

First, a strong caveat that this answer is not the best fit for this exact question. It should definitely not be the top answer!

I will go ahead and mention Mozilla's proposed BrowserID (or perhaps more precisely, the Verified Email

Protocol) in the spirit of finding an upgrade path to better approaches to authentication in the future.

I'll summarize it this way:

1. Mozilla is a nonprofit with values that align well with finding good solutions to this problem.
2. The reality today is that most websites use form-based authentication
3. Form-based authentication has a big drawback, which is an increased risk of phishing. Users are asked to enter sensitive information into an area controlled by a remote entity, rather than an area controlled by their User Agent (browser).
4. Since browsers are implicitly trusted (the whole idea of a User Agent is to act on behalf of the User), they can help improve this situation.
5. The primary force holding back progress here is deployment deadlock. Solutions must be decomposed into steps which provide some incremental benefit on their own.
6. The simplest decentralized method for expressing an identity that is built into the internet infrastructure is the domain name.
7. As a second level of expressing identity, each domain manages its own set of accounts.
8. The form "account `@` domain" is concise and supported by a wide range of protocols and URI

schemes. Such an identifier is, of course, most universally recognized as an email address.

9. Email providers are already the de-facto primary identity providers online. Current password reset flows usually let you take control of an account if you can prove that you control that account’s associated email address.
10. The Verified Email Protocol was proposed to provide a secure method, based on public key cryptography, for streamlining the process of proving to domain B that you have an account on domain A.
11. For browsers that don’t support the Verified Email Protocol (currently all of them), Mozilla provides a shim which implements the protocol in client-side JavaScript code.
12. For email services that don’t support the Verified Email Protocol, the protocol allows third parties to act as a trusted intermediary, asserting that they’ve verified a user’s ownership of an account. It is not desirable to have a large number of such third parties; this capability is intended only to allow an upgrade path, and it is much preferred that email services provide these assertions themselves.
13. Mozilla offers their own service to act like such a trusted third party. Service Providers (that is, Relying Parties) implementing the Verified Email Protocol may choose to trust Mozilla's assertions or not. Mozilla’s service verifies users’ account ownership

using the conventional means of sending an email with a confirmation link.

14. Service Providers may, of course, offer this protocol as an option in addition to any other method(s) of authentication they might wish to offer.
15. A big user interface benefit being sought here is the “identity selector”. When a user visits a site and chooses to authenticate, their browser shows them a selection of email addresses (“personal”, “work”, “political activism”, etc.) they may use to identify themselves to the site.
16. Another big user interface benefit being sought as part of this effort is helping the browser know more about the user’s session – who they’re signed in as currently, primarily – so it may display that in the browser chrome.
17. Because of the distributed nature of this system, it avoids lock-in to major sites like Facebook, Twitter, Google, etc. Any individual can own their own domain and therefore act as their own identity provider.

This is not strictly “form-based authentication for websites”. But it is an effort to transition from the current norm of form-based authentication to something more secure: browser-supported authentication.

community wiki
9 revs, 4 users 75%
Charlie

3 BrowserID link is dead – Spoody Jan 28, 2018 at 20:10

The project seems to have been mothballed.... see en.wikipedia.org/wiki/Mozilla_Persona – Jeff Olson Mar 19, 2018 at 20:44

153

I just thought I'd share this solution that I found to be working just fine.

I call it the **Dummy Field** (though I haven't invented this so don't credit me). Others know this as a honey pot.

In short: you just have to insert this into your `<form>` and check for it to be empty at when validating:

```
<input type="text" name="email"
style="display:none" />
```

The trick is to fool a bot into thinking it has to insert data into a required field, that's why I named the input "email". If you already have a field called email that you're using you should try naming the dummy field something else like "company", "phone" or "emailaddress". Just pick something you know you don't need and what sounds like something people would normally find logical to fill in into a web form. Now hide the `input` field using CSS or

JavaScript/jQuery - whatever fits you best - just **don't** set the input `type` to `hidden` or else the bot won't fall for it.

When you are validating the form (either client or server side) check if your dummy field has been filled to determine if it was sent by a human or a bot.

Example:

**In case of a human:** The user will not see the dummy field (in my case named "email") and will not attempt to fill it. So the value of the dummy field should still be empty when the form has been sent.

**In case of a bot:** The bot will see a field whose type is `text` and a name `email` (or whatever it is you called it) and will logically attempt to fill it with appropriate data. It doesn't care if you styled the input form with some fancy CSS, web-developers do it all the time. Whatever the value in the dummy field is, we don't care as long as it's larger than `0` characters.

I used this method on a guestbook in combination with CAPTCHA, and I haven't seen a single spam post since. I had used a CAPTCHA-only solution before, but eventually, it resulted in about five spam posts every hour. Adding the dummy field in the form has stopped (at least until now) all the spam from appearing.

I believe this can also be used just fine with a login/authentication form.

**Warning**: Of course this method is not 100% foolproof. Bots can be programmed to ignore input fields with the style `display:none` applied to it. You also have to think about people who use some form of auto-completion (like most browsers have built-in!) to auto-fill all form fields for them. They might just as well pick up a dummy field.

You can also vary this up a little by leaving the dummy field visible but outside the boundaries of the screen, but this is totally up to you.

Be creative!

Share Improve this answer Follow

edited Apr 27, 2022 at 11:40

community wiki
7 revs, 6 users 34%
Pieter888

37 This is a useful anti-spam trick, but I would suggest using a field name other than 'email', or you may find that browser auto-fill's fill it in, inadvertently blocking genuine users of your site. – Nico Burns May 23, 2012 at 13:18

8 I also have several more of these using `visibility:hidden` and also `position:absolute;top:-9000px` you can also do `text-indent` and also `z-index` on a few of these elements and place them in compressed CSS file names with awkward names - since bots can detect 1display:none` and they now check for a range of combinations - I actually use these methods and they're old tricks of the trade. +1 – TheBlackBenzKid Sep 12, 2012 at 9:24

21 What happens when a user with a vision impairment is using a screenreader to navigate the form? – soycharliente Nov 23, 2012 at 22:09

9 This technique has a name: the honeypot en.wikipedia.org/wiki/Honeypot_(computing) – pixeline Nov 24, 2012 at 21:14

27 No need for inline styling. Just add a class to the field (maybe use a weird word that could never mean anything to a bot), and hide it via the site's CSS file. Like: `<input type="text" name="email" class="cucaracha">` and in your CSS: `.cucaracha { display:none; }`.
– Ricardo Zea Dec 4, 2012 at 15:17

88

I do not think the above answer is "wrong" but there are large areas of authentication that are not touched upon (or rather the emphasis is on "how to implement cookie sessions", not on "what options are available and what are the trade-offs".

My suggested edits/answers are

- The problem lies more in account setup than in password checking.
- The use of two-factor authentication is much more secure than more clever means of password encryption
- Do NOT try to implement your own login form or database storage of passwords, unless the data being stored is valueless at account creation and

self-generated (that is, web 2.0 style like Facebook, Flickr, etc.)

1. Digest Authentication is a standards-based approach supported in all major browsers and servers, that will not send a password even over a secure channel.

This avoids any need to have "sessions" or cookies as the browser itself will re-encrypt the communication each time. It is the most "lightweight" development approach.

However, I do not recommend this, except for public, low-value services. This is an issue with some of the other answers above - do not try an re-implement server-side authentication mechanisms - this problem has been solved and is supported by most major browsers. Do not use cookies. Do not store anything in your own hand-rolled database. Just ask, per request, if the request is authenticated. Everything else should be supported by configuration and third-party trusted software.

So ...

First, we are confusing the initial creation of an account (with a password) with the re-checking of the password subsequently. If I am Flickr and creating your site for the first time, the new user has access to zero value (blank web space). I truly do not care if the person creating the account is lying about their name. If I am creating an account of the hospital intranet/extranet, the value lies in

all the medical records, and so I *do* care about the identity (*) of the account creator.

This is the very very hard part. The *only* decent solution is a web of trust. For example, you join the hospital as a doctor. You create a web page hosted somewhere with your photo, your passport number, and a public key, and hash them all with the private key. You then visit the hospital and the system administrator looks at your passport, sees if the photo matches you, and then hashes the web page/photo hash with the hospital private key. From now on we can securely exchange keys and tokens. As can anyone who trusts the hospital (there is the secret sauce BTW). The system administrator can also give you an RSA dongle or other two-factor authentication.

But this is a *lot* of a hassle, and not very web 2.0. However, it is the only secure way to create new accounts that have access to valuable information that is not self-created.

1. Kerberos and SPNEGO - single sign-on mechanisms with a trusted third party - basically the user verifies against a trusted third party. (NB this is not in any way the not to be trusted OAuth)
2. SRP - sort of clever password authentication without a trusted third party. But here we are getting into the realms of "it's safer to use two-factor authentication, even if that's costlier"

3. SSL client side - give the clients a public key certificate (support in all major browsers - but raises questions over client machine security).

In the end, it's a tradeoff - what is the cost of a security breach vs the cost of implementing more secure approaches. One day, we may see a proper PKI widely accepted and so no more own rolled authentication forms and databases. One day...

Share Improve this answer Follow

edited Nov 19, 2018 at 14:04

community wiki
5 revs, 4 users 67%
you cad sir - take that

31 Hard to tell which answer you are talking about in 'I do not think the above answer is "wrong"' – Davorak Nov 8, 2012 at 0:02

60

When hashing, don't use fast hash algorithms such as MD5 (many hardware implementations exist). Use something like SHA-512. For passwords, slower hashes are better.

The faster you can create hashes, the faster any brute force checker can work. Slower hashes will therefore slow down brute forcing. A slow hash algorithm will make brute forcing impractical for longer passwords (8 digits +)

Share Improve this answer Follow

edited Aug 4, 2013 at 18:09

community wiki
2 revs, 2 users 50%
josh

6 SHA-512 is also fast, so you need thousands of iterations. – Seun Osewa Dec 6, 2011 at 9:51

7 More like something like bcrypt which is designed to hash slowly. – Fabian Nicollier Jul 30, 2014 at 14:03

6 As mentioned in another answer, "OWASP recommends the use of Argon2 as your first choice for new applications. If this is not available, PBKDF2 or scrypt should be used instead. And finally if none of the above are available, use bcrypt." Neither MD5 nor any of the SHA hashing functions should ever be used for hashing passwords. This answer is bad advice. – Mike Sep 12, 2018 at 4:50

57

My favourite rule in regards to authentication systems: use passphrases, not passwords. Easy to remember, hard to crack. More info: Coding Horror: Passwords vs. Pass Phrases

Share Improve this answer Follow

edited Nov 11, 2021 at 12:02

community wiki
4 revs, 4 users 76%
Someone_who_likes_SE

even better - hard to remember, hard to guess: Password managers ... linking to article from 2005 where that likely meant an excel spreadsheet :) – felickz Jan 29, 2021 at 16:18

33

I'd like to add one suggestion I've used, based on defense in depth. You don't need to have the same auth&auth system for admins as regular users. You can have a separate login form on a separate url executing separate code for requests that will grant high privileges. This one can make choices that would be a total pain to regular users. One such that I've used is to actually scramble the login URL for admin access and email the admin the new URL. Stops any brute force attack right away as your new URL can be arbitrarily difficult (very long random string) but your admin user's only inconvenience is following a link in their email. The attacker no longer knows where to even POST to.

Share Improve this answer Follow

answered Jul 18, 2015 at 1:18

community wiki
Iain Duncan

A simple link in an email isn't actually secure, since email is not secure. – David Spector Jun 24, 2018 at 14:01

It is as secure as any other token based password reset system that is not two-factor though. Which is almost all of

them. – Iain Duncan Jun 24, 2018 at 18:31

22

I dont't know whether it was best to answer this as an answer or as a comment. I opted for the first option.

Regarding the poing **PART IV: Forgotten Password Functionality** in the first answer, I would make a point about Timing Attacks.

In the **Remember your password** forms, an attacker could potentially check a full list of emails and detect which are registered to the system (see link below).

Regarding the Forgotten Password Form, I would add that it is a good idea to equal times between successful and unsucessful queries with some delay function.

https://crypto.stanford.edu/~dabo/papers/webtiming.pdf

Share Improve this answer Follow

answered Aug 16, 2015 at 17:31

community wiki
user9869932

20

I would like to add one very important comment: -

- "In a *corporate,* **intra-** net setting," most if not all of the foregoing might not apply!

Many corporations deploy "internal use only" websites which are, effectively, "corporate applications" that happen to have been implemented through URLs. These URLs can *(supposedly ...)* only be resolved within "the company's internal network." *(Which network magically includes all VPN-connected 'road warriors.')*

When a user is dutifully-connected to the aforesaid network, their identity *("authentication")* is [already ...] "conclusively known," as is their permission *("authorization")* to do certain things ... such as ... "to access this website."

This "authentication + authorization" service can be provided by several different technologies, such as LDAP *(Microsoft OpenDirectory)*, or Kerberos.

From your point-of-view, you simply know this: that *anyone* who legitimately winds-up at your website *must* be accompanied by [an environment-variable magically containing ...] a "token." (*i.e.* The absence of such a token must be immediate grounds for `404 Not Found`.)

The token's value makes no sense to you, *but,* should the need arise, "appropriate means exist" by which your website can "[authoritatively] ask someone who knows (LDAP... etc.)" about any *and every(!)* question that you may have. In other words, you do **not** avail yourself of *any* "home-grown logic." Instead, you inquire of The Authority and implicitly trust its verdict.

Uh huh ... it's *quite* a mental-switch from the "wild-and-wooly Internet."

Share Improve this answer Follow

edited Nov 19, 2018 at 14:05

community wiki
3 revs, 3 users 90%
Mike Robinson

9 Did you fell in the punctuation well as a child? :) I've read it three times and I am still lost at what point you are trying to make. But if you are saying "Sometimes you do not need form based authentication" then you are right. But considering we are discussing when we do need it, I dont see why this is very important to note? – Hugo Delsing Apr 29, 2015 at 6:47

2 My point is that the world *outside* a corporation is entirely different from the world *inside.* If you are building an app that is accessible to the "wooly wide web," and for general consumption by the public, then you have no choice but to roll your own authentication and authorization methods. But, inside a corporation, where the only way to get there is to be there or to use VPN, then it is very likely that the application will not have – *must not* have – "its own" methods for doing these things. The app *must* use these methods instead, to provide consistent, centralized management.
– Mike Robinson Apr 30, 2015 at 13:23

2 Even intranets require a minimum amount of security in the building. Sales has confidential profit and loss numbers, while engineering has confidential intellectual property. Many companies restrict data across departmental or divisional lines. – Sablefoste Jul 30, 2017 at 23:46

10

Use OpenID Connect or User-Managed Access.

As nothing is more efficient than not doing it at all.

Share Improve this answer Follow

answered Aug 10, 2016 at 13:27

community wiki
jwilleke