**Authentication vulnerabilities**

Conceptually, authentication vulnerabilities are easy to understand. However, they are usually critical because of the clear relationship between authentication and security.

Authentication vulnerabilities can allow attackers to gain access to sensitive data and functionality. They also expose additional attack surface for further exploits. For this reason, it's important to learn how to identify and exploit authentication vulnerabilities, and how to bypass common protection measures.

In this section, we explain:

* The most common authentication mechanisms used by websites.
* Potential vulnerabilities in these mechanisms.
* Inherent vulnerabilities in different authentication mechanisms.
* Typical vulnerabilities that are introduced by their improper implementation.
* How you can make your own authentication mechanisms as robust as possible.

**What is authentication?**

Authentication is the process of verifying the identity of a user or client. Websites are potentially exposed to anyone who is connected to the internet. This makes robust authentication mechanisms integral to effective web security.

There are three main types of authentication:

* Something you **know**, such as a password or the answer to a security question. These are sometimes called "knowledge factors".
* Something you **have**, This is a physical object such as a mobile phone or security token. These are sometimes called "possession factors".
* Something you **are** or do. For example, your biometrics or patterns of behavior. These are sometimes called "inherence factors".

Authentication mechanisms rely on a range of technologies to verify one or more of these factors.

**What is the difference between authentication and authorization?**

Authentication is the process of verifying that a user is who they claim to be. Authorization involves verifying whether a user is allowed to do something.

For example, authentication determines whether someone attempting to access a website with the username Carlos123 really is the same person who created the account.

Once Carlos123 is authenticated, their permissions determine what they are authorized to do. For example, they may be authorized to access personal information about other users, or perform actions such as deleting another user's account.

**How do authentication vulnerabilities arise?**

Most vulnerabilities in authentication mechanisms occur in one of two ways:

* The authentication mechanisms are weak because they fail to adequately protect against brute-force attacks.
* Logic flaws or poor coding in the implementation allow the authentication mechanisms to be bypassed entirely by an attacker. This is sometimes called "broken authentication".

In many areas of web development, [logic flaws](https://portswigger.net/web-security/logic-flaws) cause the website to behave unexpectedly, which may or may not be a security issue. However, as authentication is so critical to security, it's very likely that flawed authentication logic exposes the website to security issues.

**What is the impact of vulnerable authentication?**

The impact of authentication vulnerabilities can be severe. If an attacker bypasses authentication or brute-forces their way into another user's account, they have access to all the data and functionality that the compromised account has. If they are able to compromise a high-privileged account, such as a system administrator, they could take full control over the entire application and potentially gain access to internal infrastructure.

Even compromising a low-privileged account might still grant an attacker access to data that they otherwise shouldn't have, such as commercially sensitive business information. Even if the account does not have access to any sensitive data, it might still allow the attacker to access additional pages, which provide a further attack surface. Often, high-severity attacks are not possible from publicly accessible pages, but they may be possible from an internal page.

**Vulnerabilities in authentication mechanisms**

A website's authentication system usually consists of several distinct mechanisms where vulnerabilities may occur. Some vulnerabilities are applicable across all of these contexts. Others are more specific to the functionality provided.

We will look more closely at some of the most common vulnerabilities in the following areas:

* Vulnerabilities in password-based login
* Vulnerabilities in multi-factor authentication
* Vulnerabilities in other authentication mechanisms

**Vulnerabilities in password-based login**

In this section, we'll look more closely at some of the most common vulnerabilities that occur in password-based login mechanisms. We'll also suggest ways that these can potentially be exploited. There are even some interactive labs so that you can try and exploit these vulnerabilities yourself.

For websites that adopt a password-based login process, users either register for an account themselves or they are assigned an account by an administrator. This account is associated with a unique username and a secret password, which the user enters in a login form to authenticate themselves.

In this scenario, the mere fact that they know the secret password is taken as sufficient proof of the user's identity. Consequently, the security of the website would be compromised if an attacker is able to either obtain or guess the login credentials of another user.

This can be achieved in a variety of ways, as we'll explore below.

**Brute-force attacks**

A brute-force attack is when an attacker uses a system of trial and error to guess valid user credentials. These attacks are typically automated using wordlists of usernames and passwords. Automating this process, especially using dedicated tools, potentially enables an attacker to make vast numbers of login attempts at high speed.

Brute-forcing is not always just a case of making completely random guesses at usernames and passwords. By also using basic logic or publicly available knowledge, attackers can fine-tune brute-force attacks to make much more educated guesses. This considerably increases the efficiency of such attacks. Websites that rely on password-based login as their sole method of authenticating users can be highly vulnerable if they do not implement sufficient brute-force protection.

**Brute-forcing usernames**

Usernames are especially easy to guess if they conform to a recognizable pattern, such as an email address. For example, it is very common to see business logins in the format firstname.lastname@somecompany.com. However, even if there is no obvious pattern, sometimes even high-privileged accounts are created using predictable usernames, such as admin or administrator.

During auditing, check whether the website discloses potential usernames publicly. For example, are you able to access user profiles without logging in? Even if the actual content of the profiles is hidden, the name used in the profile is sometimes the same as the login username. You should also check HTTP responses to see if any email addresses are disclosed. Occasionally, responses contain email addresses of high-privileged users, such as administrators or IT support.

**Brute-forcing passwords**

Passwords can similarly be brute-forced, with the difficulty varying based on the strength of the password. Many websites adopt some form of password policy, which forces users to create high-entropy passwords that are, theoretically at least, harder to crack using brute-force alone. This typically involves enforcing passwords with:

* A minimum number of characters
* A mixture of lower and uppercase letters
* At least one special character

However, while high-entropy passwords are difficult for computers alone to crack, we can use a basic knowledge of human behavior to exploit the vulnerabilities that users unwittingly introduce to this system. Rather than creating a strong password with a random combination of characters, users often take a password that they can remember and try to crowbar it into fitting the password policy. For example, if mypassword is not allowed, users may try something like Mypassword1! or Myp4$$w0rd instead.

In cases where the policy requires users to change their passwords on a regular basis, it is also common for users to just make minor, predictable changes to their preferred password. For example, Mypassword1! becomes Mypassword1? or Mypassword2!.

This knowledge of likely credentials and predictable patterns means that brute-force attacks can often be much more sophisticated, and therefore effective, than simply iterating through every possible combination of characters.

**Username enumeration**

Username enumeration is when an attacker is able to observe changes in the website's behavior in order to identify whether a given username is valid.

Username enumeration typically occurs either on the login page, for example, when you enter a valid username but an incorrect password, or on registration forms when you enter a username that is already taken. This greatly reduces the time and effort required to brute-force a login because the attacker is able to quickly generate a shortlist of valid usernames.

While attempting to brute-force a login page, you should pay particular attention to any differences in:

* **Status codes**: During a brute-force attack, the returned HTTP status code is likely to be the same for the vast majority of guesses because most of them will be wrong. If a guess returns a different status code, this is a strong indication that the username was correct. It is best practice for websites to always return the same status code regardless of the outcome, but this practice is not always followed.
* **Error messages**: Sometimes the returned error message is different depending on whether both the username AND password are incorrect or only the password was incorrect. It is best practice for websites to use identical, generic messages in both cases, but small typing errors sometimes creep in. Just one character out of place makes the two messages distinct, even in cases where the character is not visible on the rendered page.
* **Response times**: If most of the requests were handled with a similar response time, any that deviate from this suggest that something different was happening behind the scenes. This is another indication that the guessed username might be correct. For example, a website might only check whether the password is correct if the username is valid. This extra step might cause a slight increase in the response time. This may be subtle, but an attacker can make this delay more obvious by entering an excessively long password that the website takes noticeably longer to handle.

**Flawed brute-force protection**

It is highly likely that a brute-force attack will involve many failed guesses before the attacker successfully compromises an account. Logically, brute-force protection revolves around trying to make it as tricky as possible to automate the process and slow down the rate at which an attacker can attempt logins. The two most common ways of preventing brute-force attacks are:

* Locking the account that the remote user is trying to access if they make too many failed login attempts
* Blocking the remote user's IP address if they make too many login attempts in quick succession

Both approaches offer varying degrees of protection, but neither is invulnerable, especially if implemented using flawed logic.

For example, you might sometimes find that your IP is blocked if you fail to log in too many times. In some implementations, the counter for the number of failed attempts resets if the IP owner logs in successfully. This means an attacker would simply have to log in to their own account every few attempts to prevent this limit from ever being reached.

In this case, merely including your own login credentials at regular intervals throughout the wordlist is enough to render this defense virtually useless.

**Account locking**

One way in which websites try to prevent brute-forcing is to lock the account if certain suspicious criteria are met, usually a set number of failed login attempts. Just as with normal login errors, responses from the server indicating that an account is locked can also help an attacker to enumerate usernames.

Locking an account offers a certain amount of protection against targeted brute-forcing of a specific account. However, this approach fails to adequately prevent brute-force attacks in which the attacker is just trying to gain access to any random account they can.

For example, the following method can be used to work around this kind of protection:

1. Establish a list of candidate usernames that are likely to be valid. This could be through username enumeration or simply based on a list of common usernames.
2. Decide on a very small shortlist of passwords that you think at least one user is likely to have. Crucially, the number of passwords you select must not exceed the number of login attempts allowed. For example, if you have worked out that limit is 3 attempts, you need to pick a maximum of 3 password guesses.
3. Using a tool such as Burp Intruder, try each of the selected passwords with each of the candidate usernames. This way, you can attempt to brute-force every account without triggering the account lock. You only need a single user to use one of the three passwords in order to compromise an account.

Account locking also fails to protect against credential stuffing attacks. This involves using a massive dictionary of username:password pairs, composed of genuine login credentials stolen in data breaches. Credential stuffing relies on the fact that many people reuse the same username and password on multiple websites and, therefore, there is a chance that some of the compromised credentials in the dictionary are also valid on the target website. Account locking does not protect against credential stuffing because each username is only being attempted once. Credential stuffing is particularly dangerous because it can sometimes result in the attacker compromising many different accounts with just a single automated attack.

**User rate limiting**

Another way websites try to prevent brute-force attacks is through user rate limiting. In this case, making too many login requests within a short period of time causes your IP address to be blocked. Typically, the IP can only be unblocked in one of the following ways:

* Automatically after a certain period of time has elapsed
* Manually by an administrator
* Manually by the user after successfully completing a CAPTCHA

User rate limiting is sometimes preferred to account locking due to being less prone to username enumeration and denial of service attacks. However, it is still not completely secure. As we saw an example of in an earlier lab, there are several ways an attacker can manipulate their apparent IP in order to bypass the block.

As the limit is based on the rate of HTTP requests sent from the user's IP address, it is sometimes also possible to bypass this defense if you can work out how to guess multiple passwords with a single request.

**HTTP basic authentication**

Although fairly old, its relative simplicity and ease of implementation means you might sometimes see HTTP basic authentication being used. In HTTP basic authentication, the client receives an authentication token from the server, which is constructed by concatenating the username and password, and encoding it in Base64. This token is stored and managed by the browser, which automatically adds it to the Authorization header of every subsequent request as follows:

Authorization: Basic base64(username:password)

For a number of reasons, this is generally not considered a secure authentication method. Firstly, it involves repeatedly sending the user's login credentials with every request. Unless the website also implements HSTS, user credentials are open to being captured in a man-in-the-middle attack.

In addition, implementations of HTTP basic authentication often don't support brute-force protection. As the token consists exclusively of static values, this can leave it vulnerable to being brute-forced.

HTTP basic authentication is also particularly vulnerable to session-related exploits, notably [CSRF](https://portswigger.net/web-security/csrf), against which it offers no protection on its own.

In some cases, exploiting vulnerable HTTP basic authentication might only grant an attacker access to a seemingly uninteresting page. However, in addition to providing a further attack surface, the credentials exposed in this way might be reused in other, more confidential contexts.

**Vulnerabilities in multi-factor authentication**

In this section, we'll look at some of the vulnerabilities that can occur in multi-factor authentication mechanisms. We've also provided several interactive labs to demonstrate how you can exploit these vulnerabilities in multi-factor authentication.

Many websites rely exclusively on single-factor authentication using a password to authenticate users. However, some require users to prove their identity using multiple authentication factors.

Verifying biometric factors is impractical for most websites. However, it is increasingly common to see both mandatory and optional two-factor authentication (2FA) based on **something you know** and **something you have**. This usually requires users to enter both a traditional password and a temporary verification code from an out-of-band physical device in their possession.

While it is sometimes possible for an attacker to obtain a single knowledge-based factor, such as a password, being able to simultaneously obtain another factor from an out-of-band source is considerably less likely. For this reason, two-factor authentication is demonstrably more secure than single-factor authentication. However, as with any security measure, it is only ever as secure as its implementation. Poorly implemented two-factor authentication can be beaten, or even bypassed entirely, just as single-factor authentication can.

It is also worth noting that the full benefits of multi-factor authentication are only achieved by verifying multiple **different** factors. Verifying the same factor in two different ways is not true two-factor authentication. Email-based 2FA is one such example. Although the user has to provide a password and a verification code, accessing the code only relies on them knowing the login credentials for their email account. Therefore, the knowledge authentication factor is simply being verified twice.

**Two-factor authentication tokens**

Verification codes are usually read by the user from a physical device of some kind. Many high-security websites now provide users with a dedicated device for this purpose, such as the RSA token or keypad device that you might use to access your online banking or work laptop. In addition to being purpose-built for security, these dedicated devices also have the advantage of generating the verification code directly. It is also common for websites to use a dedicated mobile app, such as Google Authenticator, for the same reason.

On the other hand, some websites send verification codes to a user's mobile phone as a text message. While this is technically still verifying the factor of "something you have", it is open to abuse. Firstly, the code is being transmitted via SMS rather than being generated by the device itself. This creates the potential for the code to be intercepted. There is also a risk of SIM swapping, whereby an attacker fraudulently obtains a SIM card with the victim's phone number. The attacker would then receive all SMS messages sent to the victim, including the one containing their verification code.

**Bypassing two-factor authentication**

At times, the implementation of two-factor authentication is flawed to the point where it can be bypassed entirely.

If the user is first prompted to enter a password, and then prompted to enter a verification code on a separate page, the user is effectively in a "logged in" state before they have entered the verification code. In this case, it is worth testing to see if you can directly skip to "logged-in only" pages after completing the first authentication step. Occasionally, you will find that a website doesn't actually check whether or not you completed the second step before loading the page.

**Flawed two-factor verification logic**

Sometimes flawed logic in two-factor authentication means that after a user has completed the initial login step, the website doesn't adequately verify that the same user is completing the second step.

For example, the user logs in with their normal credentials in the first step as follows:

POST /login-steps/first HTTP/1.1

Host: vulnerable-website.com

...

username=carlos&password=qwerty

They are then assigned a cookie that relates to their account, before being taken to the second step of the login process:

HTTP/1.1 200 OK

Set-Cookie: account=carlos

GET /login-steps/second HTTP/1.1

Cookie: account=carlos

When submitting the verification code, the request uses this cookie to determine which account the user is trying to access:

POST /login-steps/second HTTP/1.1

Host: vulnerable-website.com

Cookie: account=carlos

...

verification-code=123456

In this case, an attacker could log in using their own credentials but then change the value of the account cookie to any arbitrary username when submitting the verification code.

POST /login-steps/second HTTP/1.1

Host: vulnerable-website.com

Cookie: account=victim-user

...

verification-code=123456

This is extremely dangerous if the attacker is then able to brute-force the verification code as it would allow them to log in to arbitrary users' accounts based entirely on their username. They would never even need to know the user's password.

**Brute-forcing 2FA verification codes**

As with passwords, websites need to take steps to prevent brute-forcing of the 2FA verification code. This is especially important because the code is often a simple 4 or 6-digit number. Without adequate brute-force protection, cracking such a code is trivial.

Some websites attempt to prevent this by automatically logging a user out if they enter a certain number of incorrect verification codes. This is ineffective in practice because an advanced attacker can even automate this multi-step process by [creating macros](https://portswigger.net/burp/documentation/desktop/settings/sessions#macros) for Burp Intruder. The [Turbo Intruder](https://portswigger.net/bappstore/9abaa233088242e8be252cd4ff534988) extension can also be used for this purpose.

**Vulnerabilities in other authentication mechanisms**

In this section, we'll look at some of the supplementary functionality that is related to authentication and demonstrate how these can be vulnerable. We've also created several interactive labs that you can use to put what you've learned into practice.

In addition to the basic login functionality, most websites provide supplementary functionality to allow users to manage their account. For example, users can typically change their password or reset their password when they forget it. These mechanisms can also introduce vulnerabilities that can be exploited by an attacker.

Websites usually take care to avoid well-known vulnerabilities in their login pages. But it is easy to overlook the fact that you need to take similar steps to ensure that related functionality is equally as robust. This is especially important in cases where an attacker is able to create their own account and, consequently, has easy access to study these additional pages.

**Keeping users logged in**

A common feature is the option to stay logged in even after closing a browser session. This is usually a simple checkbox labeled something like "Remember me" or "Keep me logged in".

This functionality is often implemented by generating a "remember me" token of some kind, which is then stored in a persistent cookie. As possessing this cookie effectively allows you to bypass the entire login process, it is best practice for this cookie to be impractical to guess. However, some websites generate this cookie based on a predictable concatenation of static values, such as the username and a timestamp. Some even use the password as part of the cookie. This approach is particularly dangerous if an attacker is able to create their own account because they can study their own cookie and potentially deduce how it is generated. Once they work out the formula, they can try to brute-force other users' cookies to gain access to their accounts.

Some websites assume that if the cookie is encrypted in some way it will not be guessable even if it does use static values. While this may be true if done correctly, naively "encrypting" the cookie using a simple two-way encoding like Base64 offers no protection whatsoever. Even using proper encryption with a one-way hash function is not completely bulletproof. If the attacker is able to easily identify the hashing algorithm, and no salt is used, they can potentially brute-force the cookie by simply hashing their wordlists. This method can be used to bypass login attempt limits if a similar limit isn't applied to cookie guesses.

Even if the attacker is not able to create their own account, they may still be able to exploit this vulnerability. Using the usual techniques, such as [XSS](https://portswigger.net/web-security/cross-site-scripting), an attacker could steal another user's "remember me" cookie and deduce how the cookie is constructed from that. If the website was built using an open-source framework, the key details of the cookie construction may even be publicly documented.

In some rare cases, it may be possible to obtain a user's actual password in cleartext from a cookie, even if it is hashed. Hashed versions of well-known password lists are available online, so if the user's password appears in one of these lists, decrypting the hash can occasionally be as trivial as just pasting the hash into a search engine. This demonstrates the importance of salt in effective encryption.

**Resetting user passwords**

In practice, it is a given that some users will forget their password, so it is common to have a way for them to reset it. As the usual password-based authentication is obviously impossible in this scenario, websites have to rely on alternative methods to make sure that the real user is resetting their own password. For this reason, the password reset functionality is inherently dangerous and needs to be implemented securely.

There are a few different ways that this feature is commonly implemented, with varying degrees of vulnerability.

**Sending passwords by email**

It should go without saying that sending users their current password should never be possible if a website handles passwords securely in the first place. Instead, some websites generate a new password and send this to the user via email.

Generally speaking, sending persistent passwords over insecure channels is to be avoided. In this case, the security relies on either the generated password expiring after a very short period, or the user changing their password again immediately. Otherwise, this approach is highly susceptible to man-in-the-middle attacks.

Email is also generally not considered secure given that inboxes are both persistent and not really designed for secure storage of confidential information. Many users also automatically sync their inbox between multiple devices across insecure channels.

**Resetting passwords using a URL**

A more robust method of resetting passwords is to send a unique URL to users that takes them to a password reset page. Less secure implementations of this method use a URL with an easily guessable parameter to identify which account is being reset, for example:

http://vulnerable-website.com/reset-password?user=victim-user

In this example, an attacker could change the user parameter to refer to any username they have identified. They would then be taken straight to a page where they can potentially set a new password for this arbitrary user.

A better implementation of this process is to generate a high-entropy, hard-to-guess token and create the reset URL based on that. In the best case scenario, this URL should provide no hints about which user's password is being reset.

http://vulnerable-website.com/reset-password?token=a0ba0d1cb3b63d13822572fcff1a241895d893f659164d4cc550b421ebdd48a8

When the user visits this URL, the system should check whether this token exists on the back-end and, if so, which user's password it is supposed to reset. This token should expire after a short period of time and be destroyed immediately after the password has been reset.

However, some websites fail to also validate the token again when the reset form is submitted. In this case, an attacker could simply visit the reset form from their own account, delete the token, and leverage this page to reset an arbitrary user's password.

If the URL in the reset email is generated dynamically, this may also be vulnerable to password reset poisoning. In this case, an attacker can potentially steal another user's token and use it change their password.

**Password reset poisoning**

Password reset poisoning is a technique whereby an attacker manipulates a vulnerable website into generating a password reset link pointing to a domain under their control. This behavior can be leveraged to steal the secret tokens required to reset arbitrary users' passwords and, ultimately, compromise their accounts.

**How does a password reset work?**

Virtually all websites that require a login also implement functionality that allows users to reset their password if they forget it. There are several ways of doing this, with varying degrees of security and practicality. One of the most common approaches goes something like this:

1. The user enters their username or email address and submits a password reset request.
2. The website checks that this user exists and then generates a temporary, unique, high-entropy token, which it associates with the user's account on the back-end.
3. The website sends an email to the user that contains a link for resetting their password. The user's unique reset token is included as a query parameter in the corresponding URL:

https://normal-website.com/reset?token=0a1b2c3d4e5f6g7h8i9j

1. When the user visits this URL, the website checks whether the provided token is valid and uses it to determine which account is being reset. If everything is as expected, the user is given the option to enter a new password. Finally, the token is destroyed.

This process is simple enough and relatively secure in comparison to some other approaches. However, its security relies on the principle that only the intended user has access to their email inbox and, therefore, to their unique token. Password reset poisoning is a method of stealing this token in order to change another user's password.

**How to construct a password reset poisoning attack**

If the URL that is sent to the user is dynamically generated based on controllable input, such as the Host header, it may be possible to construct a password reset poisoning attack as follows:

1. The attacker obtains the victim's email address or username, as required, and submits a password reset request on their behalf. When submitting the form, they intercept the resulting HTTP request and modify the Host header so that it points to a domain that they control. For this example, we'll use evil-user.net.
2. The victim receives a genuine password reset email directly from the website. This seems to contain an ordinary link to reset their password and, crucially, contains a valid password reset token that is associated with their account. However, the domain name in the URL points to the attacker's server:

https://evil-user.net/reset?token=0a1b2c3d4e5f6g7h8i9j

1. If the victim clicks this link (or it is fetched in some other way, for example, by an antivirus scanner) the password reset token will be delivered to the attacker's server.
2. The attacker can now visit the real URL for the vulnerable website and supply the victim's stolen token via the corresponding parameter. They will then be able to reset the user's password to whatever they like and subsequently log in to their account.

In a real attack, the attacker may seek to increase the probability of the victim clicking the link by first warming them up with a fake breach notification, for example.

Even if you can't control the password reset link, you can sometimes use the Host header to inject HTML into sensitive emails. Note that email clients typically don't execute JavaScript, but other HTML injection techniques like [dangling markup attacks](https://portswigger.net/web-security/cross-site-scripting/dangling-markup) may still apply.

**Changing user passwords**

Typically, changing your password involves entering your current password and then the new password twice. These pages fundamentally rely on the same process for checking that usernames and current passwords match as a normal login page does. Therefore, these pages can be vulnerable to the same techniques.

Password change functionality can be particularly dangerous if it allows an attacker to access it directly without being logged in as the victim user. For example, if the username is provided in a hidden field, an attacker might be able to edit this value in the request to target arbitrary users. This can potentially be exploited to enumerate usernames and brute-force passwords.

# How to secure your authentication mechanisms

In this section, we'll talk about how you can prevent some of the vulnerabilities we've discussed from occurring in your authentication mechanisms.

Authentication is a complex topic and, as we have demonstrated, it is unfortunately all too easy for weaknesses and flaws to creep in. Outlining every possible measure you can take to protect your own websites is clearly not possible. However, there are several general principles that you should always follow.

## Take care with user credentials

Even the most robust authentication mechanisms are ineffective if you unwittingly disclose a valid set of login credentials to an attacker. It should go without saying that you should never send any login data over unencrypted connections. Although you may have implemented HTTPS for your login requests, make sure that you enforce this by redirecting any attempted HTTP requests to HTTPS as well.

You should also audit your website to make sure that no username or email addresses are disclosed either through publicly accessible profiles or reflected in HTTP responses, for example.

## Don't count on users for security

Strict authentication measures often require some additional effort from your users. Human nature makes it all but inevitable that some users will find ways to save themselves this effort. Therefore, you need to enforce secure behavior wherever possible.

The most obvious example is to implement an effective password policy. Some of the more traditional policies fall down because people crowbar their own predictable passwords into the policy. Instead, it can be more effective to implement a simple password checker of some kind, which allows users to experiment with passwords and provides feedback about their strength in real time. A popular example is the JavaScript library zxcvbn, which was developed by Dropbox. By only allowing passwords which are rated highly by the password checker, you can enforce the use of secure passwords more effectively than you can with traditional policies.

## Prevent username enumeration

It is considerably easier for an attacker to break your authentication mechanisms if you reveal that a user exists on the system. There are even certain situations where, due to the nature of the website, the knowledge that a particular person has an account is sensitive information in itself.

Regardless of whether an attempted username is valid, it is important to use identical, generic error messages, and make sure they really are identical. You should always return the same HTTP status code with each login request and, finally, make the response times in different scenarios as indistinguishable as possible.

## Implement robust brute-force protection

Given how simple constructing a brute-force attack can be, it is vital to ensure that you take steps to prevent, or at least disrupt, any attempts to brute-force logins.

One of the more effective methods is to implement strict, IP-based user rate limiting. This should involve measures to prevent attackers from manipulating their apparent IP address. Ideally, you should require the user to complete a CAPTCHA test with every login attempt after a certain limit is reached.

Keep in mind that this is not guaranteed to completely eliminate the threat of brute-forcing. However, making the process as tedious and manual as possible increases the likelihood that any would-be attacker gives up and goes in search of a softer target instead.

## Triple-check your verification logic

As demonstrated by our [labs](https://portswigger.net/web-security/all-labs#authentication), it is easy for simple [logic flaws](https://portswigger.net/web-security/logic-flaws) to creep into code which, in the case of authentication, have the potential to completely compromise your website and users. Auditing any verification or validation logic thoroughly to eliminate flaws is absolutely key to robust authentication. A check that can be bypassed is, ultimately, not much better than no check at all.

## Don't forget supplementary functionality

Be sure not to just focus on the central login pages and overlook additional functionality related to authentication. This is particularly important in cases where the attacker is free to register their own account and explore this functionality. Remember that a password reset or change is just as valid an attack surface as the main login mechanism and, consequently, must be equally as robust.

## Implement proper multi-factor authentication

While multi-factor authentication may not be practical for every website, when done properly it is much more secure than password-based login alone. Remember that verifying multiple instances of the same factor is not true multi-factor authentication. Sending verification codes via email is essentially just a more long-winded form of single-factor authentication.

SMS-based 2FA is technically verifying two factors (something you know and something you have). However, the potential for abuse through SIM swapping, for example, means that this system can be unreliable.

Ideally, 2FA should be implemented using a dedicated device or app that generates the verification code directly. As they are purpose-built to provide security, these are typically more secure.

Finally, just as with the main authentication logic, make sure that the logic in your 2FA checks is sound so that it cannot be easily bypassed.