**2-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.

This can be done by simply noting the url for the authenticated page and manually navigating to it after logging in but before completing the 2-factor authentication always try this type of access control bypass if 2-FA is used by a website.

\*\*We can also try to brute force the 2-FA code using the zap fuzzer and somehow try to bypass the race condition if one is present (which is very likely) \*\*

**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.

Basically what this is saying is that we can use repeater to send a request using an alternate username that will essentially generate a MFA(multifactor authentication) code for the username we supply. We can then brute force that users generated MFA code then login to our own account but submit the other users MFA code to get access to their account without knowing their password.

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.

\*\* most 2FA codes are 4 or 6 digits therefore we need to create a script to add the padded 0s we can easily do this with a simple python script

The zfill() method in Python is used to pad a numeric string on the left with zeros, like print(num. zfill(3))

Ex : for i in range (0,10000): print(i.zfill(4))

This makes it so that padded 0s are added such that the number is always 4 digits long