

Zaman Tabanlı İstemci-Sunucu Dinamik Kimlik Doğrulama

Gürkan Uçar

Özet : Bu çalışma, web tabanlı yazılımlarda kullanılan kimlik doğrulama ve yetkilendirme yöntemlerini incelemekte, bu yöntemlerin avantajlarını ve dezavantajlarını ele almaktadır. Ayrıca, daha güvenli bir sistemin geliştirilebileceğini göstermektedir. Steam Guard ve Symantec benzeri zaman tabanlı tek kullanımlık şifre ile giriş yönteminin istemci-sunucu isteklerine dinamik olarak uyarlanması ile güvenlik artırılmıştır. Bu yöntem, ağdaki paketleri izleyen kişilerin bilgileri ele geçirse bile, aynı bilgilerle yapılan ikinci bir isteğin zaman kısıtı yüzünden geçersiz olmasını sağlar.

Anahtar Kelimeler : Yetki ve Kimliklendirme, TOTP, JWT, Digest Auth

Abstract : *This study examines the authentication and authorization methods used in web-based software and discusses the advantages and disadvantages of these methods. It also shows that a more secure system can be developed. Security is increased by dynamically adapting the login method to client-server requests with a time-based one-time password similar to Steam Guard and Symantec. This method ensures that even if people monitoring the packets on the network obtain the information, a second request made with the same information will be invalid due to time constraints.*

Keywords : Authentication And Authorization, TOTP, JWT, Digest Auth

1 Giriş

Günümüzde bilgi teknolojileri dünyasında güvenlik her zaman en kritik konulardan biri olmuştur. Kimlik doğrulama için kullanılan bir çok yöntem vardır. Tüm bu yöntemlerin bazı avantaj ve dezavantajları vardır. Şu anda en popüler olan yöntemlerden bazıları Bearer JWT (JSON Web Token), Basic Authentication ve Digest Authentication'dır. Ayrıca kullanıcı giriş yaparken kullanılan TOTP (Time Based One Time Password) yöntemi bulunmaktadır. Bu yöntem de Steam Guard, Symantec gibi birçok uygulamada yer almaktadır. Benim yapmaya çalıştığım ise sadece giriş yaparken kullanılan bu yöntemi, sunucuya HTTP isteği atarken her an dinamik olarak gerçekleştirmektir. Bu sistem diğer yöntemlere kıyasla yüksek güvenli bir alternatif olarak karşımıza çıkmaktadır.

2 Yetki Ve Kimliklendirme Sistemlerinin Genel Yapısı

İstemci-sunucu mimarisinde kullanılan yetki ve kimliklendirme sistemlerinde genel olarak mantık; istemci başarılı olarak giriş yaptıktan sonra giriş yapılan kullanıcı adı ve parolayı bir yerde saklayıp her HTTP isteğinde, başlık (header) kısmına eklenmesi ile gerçekleşmektedir. Bir diğer yöntem ise; kullanıcının başarılı girişi ardından bir token oluşturup bunu geriye döndürmek ve her istek öncesi bu token'ı başlık kısmına eklemektir. Bu, ilk yöntemle göre daha güvenlidir.

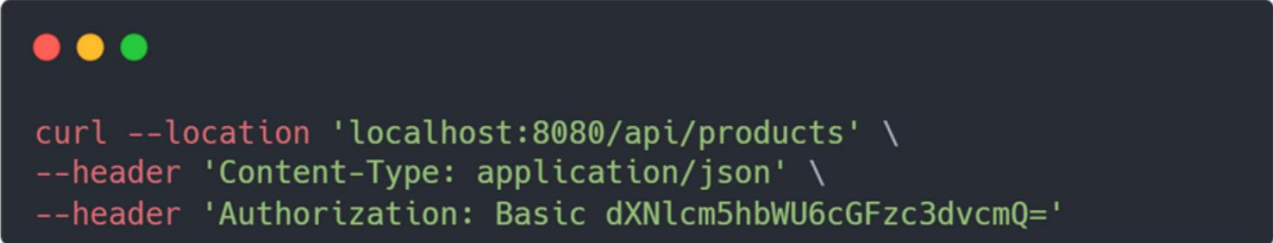
Sunucu tarafında ise kontroller, her istek geldiğinde başlık kısmından alınan kimliklendirme bilgileri eğer geçerli ise isteğin bir alt katmana erişmesine izin vermesi şeklinde gerçekleşir.

3 Yetki Ve Kimliklendirme Yöntemleri

3.1 Basic Authentication - Temel Kimliklendirme

Basic Authentication; istemcilerin sunucuya atacağı her isteğin "authorization" headerına kullanıcı adı ve parolasını eklemesi ve bunu sunucu tarafında kontrol etmesi ile gerçekleşir.

Detaylarına değinecek olursak; kullanıcı adı ve parola "username:password" şeklinde, arada ":" olacak şekilde birleştirilir. Ardından bu yeni metin base64 tabanında kodlanır (encode). Sunucu tarafında ise her istek öncesinde, authorization header da bulunan bu metnin öncelikle kodu çözülür (decode). Ardından dizi ayrıştırma (split) yöntemi ile ":" ayrıştırılarak, ilk kısım kullanıcı adı ikinci kısım ise parola olarak bir değişkene atanır. Akabinde ise veritabanına bu bilgiler ile gidilerek doğruluğu denetlenir. Bu yöntemin en büyük dezavantajlarından birisi her istekte giden bu kullanıcı adı ve şifrenin kolaylıkla kodunun çözülüp kullanıcı bilgilerinin ele geçirilebilmesidir. Güvenlik açısından zayıf kaldığı için genellikle HTTPS ile birlikte kullanılması önerilir.



```
curl --location 'localhost:8080/api/products' \
--header 'Content-Type: application/json' \
--header 'Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ='
```

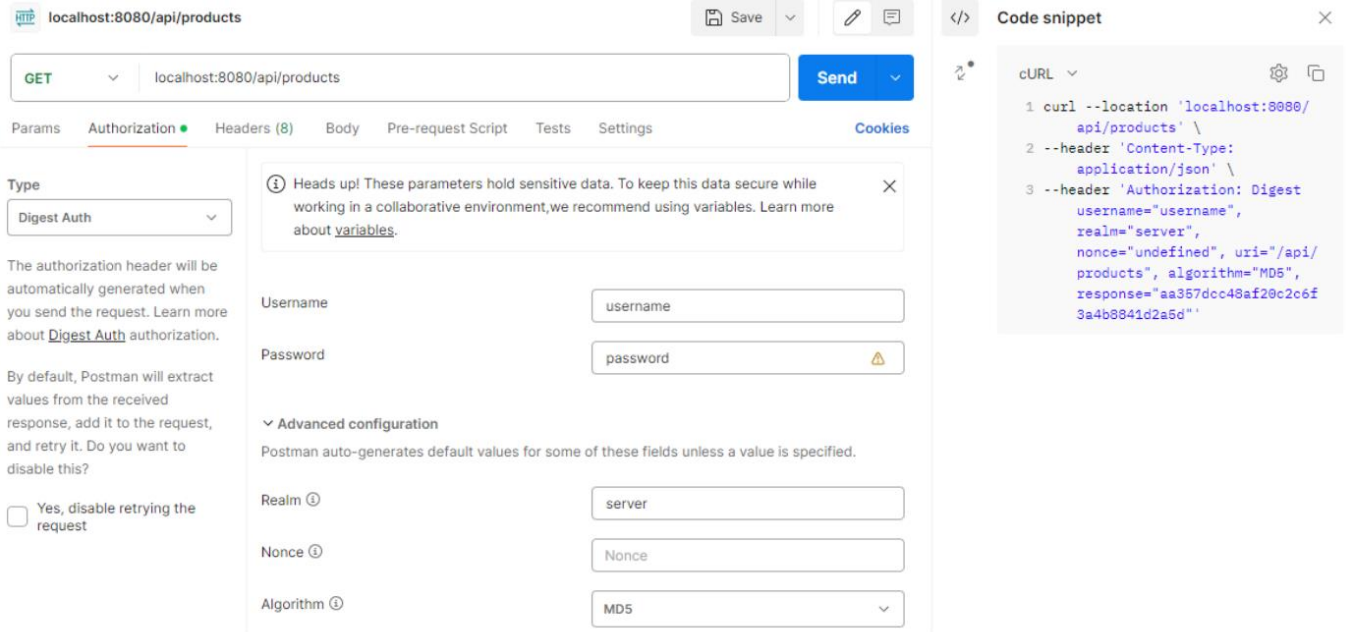
Şekil 1: Basic Authentication Curl Example

3.2 Digest Authentication

Digest Authentication, Basic Authentication'a göre daha güvenli bir seçenektir. Bu

yöntemde, kullanıcı adı ve parola gibi bilgiler doğrudan gönderilmez; bunun yerine, bu bilgilerin hash'lenmiş özeti ve her istek için benzersiz olan bir nonce (tek kullanımlık sayı) gönderilir. Bu işlem iki adımda gerçekleşir: Önce istemci sunucudan nonce ister, sonra bu nonce ile birlikte kullanıcı bilgilerinin hash'ini hesaplayıp sunucuya gönderir. Sunucu da bu hash'i doğrulayarak işlemi tamamlar.

Her istek için iki kez iletişim kurulmasını gerektirdiği için ağ trafiğini arttırır. Ancak, her isteğin benzersiz olması sayesinde replay saldırılarına karşı etkili bir koruma sağlar.



Şekil 2: Digest Authentication

3.3 JWT (JSON Web Token) Bearer Authentication

Bearer Authentication temel mantığı; kullanıcı giriş yaptıktan sonra bir token üretilmesi ve bu token'ın her istek öncesi header'a eklenmesi şeklindedir. Digest Auth'dan farklı olarak anahtar bir kez üretilir ve diğer isteklerde de aynı token kullanılır. Fakat bu token ele geçerse başkaları tarafından da istek atılabilir. Bu sebeple replay saldırılarına açıktır. Bu yöntemi daha güvenli hale getirmek adına tokenlar'a geçerli olma süresi eklenmiş ve ayrıca refresh-token (yenileme token'ı) ile beraber kullanımı önerilmiştir.

JWT ise; Bearer Auth'ı baz alır ve bir token üretme algoritmasıdır. Token içerisine kullanıcının rolü, email adresi gibi bilgiler de eklenebilmektedir. İçerisinde ek olarak token'ı düzenleyen bilgisi, son kullanma tarihi ve kullanılan hash algoritması gibi bilgiler yer alır. Header, Payload ve Signature'dan meydana gelir.

Header: Token'ın türünü ve kullanılan imza veya şifreleme algoritmasını (örneğin, HMAC, RSA) belirtir. Bu bölüm JSON formatında yazılır ve Base64 ile kodlanır.

Payload: Asıl verilerin saklandığı bölümdür. Bu kısım genellikle kullanıcı kimliği, yetki bilgileri gibi kullanıcıya özgü bilgileri ve token'ın geçerlilik süresi gibi bilgileri içerir. Base64 ile

kodlanır.

Signature: Bu bölüm, token'ın bütünlüğünü ve doğruluğunu korumak için kullanılır. Header ve Payload, belirtilen bir algoritma kullanılarak bir anahtar ile hashlenir (örneğin HMAC veya RSA ile). Sonuçta elde edilen imza, token'ın sonuna eklenir ve bu imza, token'ı alan tarafın, token'ın içeriğinin değiştirilmeden geldiğini doğrulamasını sağlar.

JWT'nin yapısı xxx.yyy.zzz şeklindedir, burada xxx Header'ı, yyy Payload'ı ve zzz ise Signature'ı temsil eder.

Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikd1cmthbiBVQ0FSIiwiaWF0IjoxNTEyMjM5MDIyfQ.JtE8VmeW7-LHZwv0cLg0IOE4i1RIYqw3B1mZ6EDyhOk

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

PAYLOAD: DATA

```
{  "name": "Gurkan UCAR",  "iat": 1516239022}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  your-256-bit-secret)
```

☐ secret base64 encoded

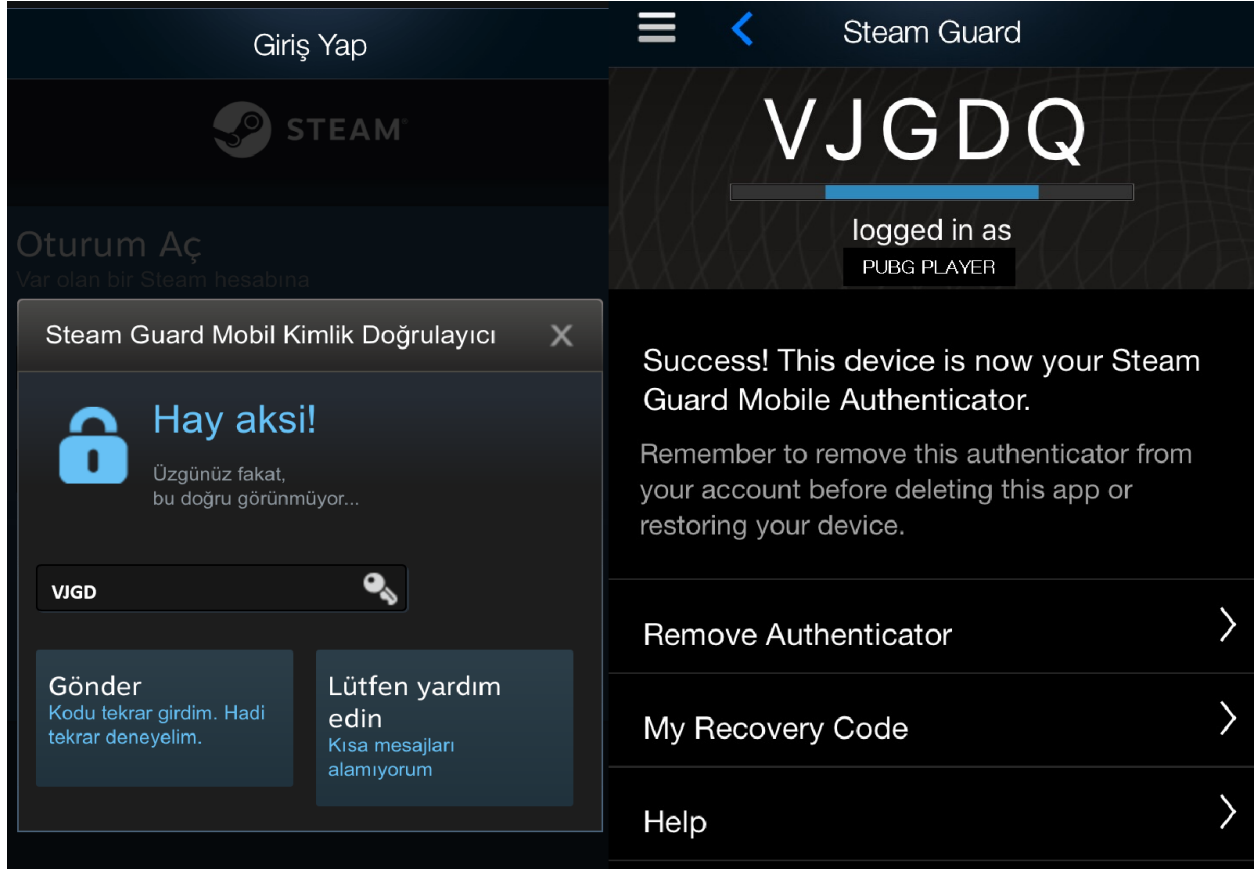
Signature Verified

SHARE JWT

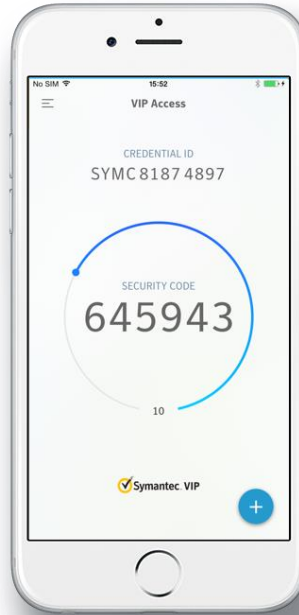
Şekil 3: JWT örneği

3.4 TOTP (Time-Based One-Time Password)

Zaman bazlı tek kullanımlık parola; diğer yöntemlerin aksine istek atılırken değil kullanıcı sisteme giriş yaparken tek bir defalık kullanılır. Örneğin 30 saniye gibi belirli bir süre boyunca geçerli olan tek kullanımlık şifreler üretir. Bu yöntem özellikle iki faktörlü doğrulama (2FA) süreçlerinde yaygın olarak kullanılır. Ayrıca bu yöntem, şifreyi üreten cihaz online olmasa bile çalışmaktadır. Önemli olan sunucu ile cihaz arasındaki zaman farkının tespit edilmesidir. Steam, Symantec gibi platformlar da bu yöntemi sıklıkla kullanmaktadır.



Şekil 4: Steam Guard Örneği



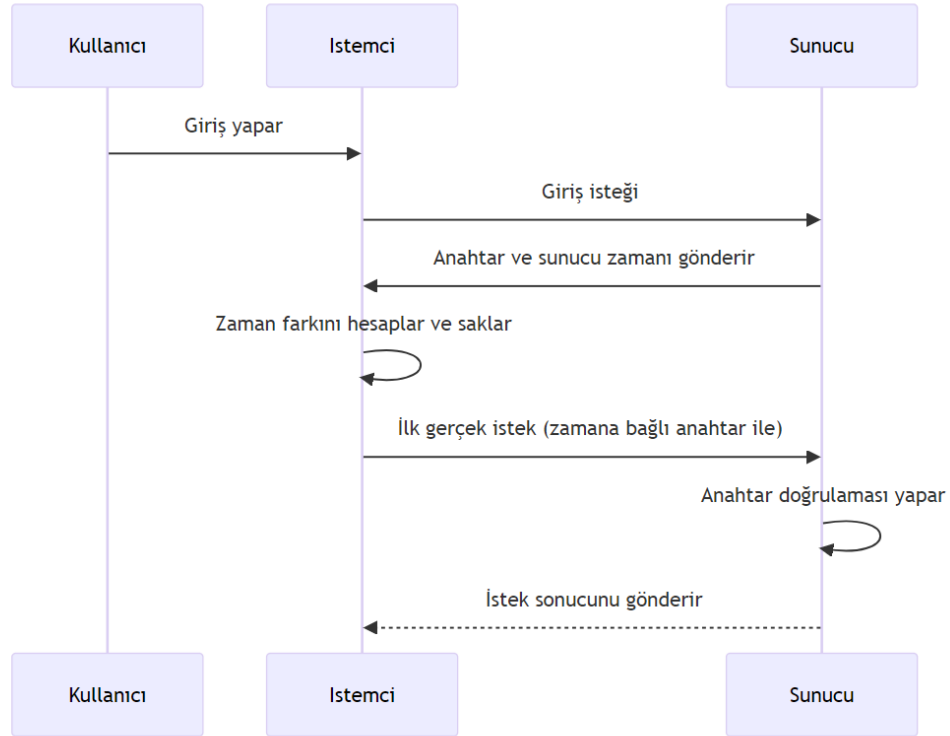
Şekil 5: Symantec Örneği

4 Zaman Tabanlı İstemci-Sunucu Dinamik Kimlik Doğrulama Yöntemi

Bu yöntemde; kullanıcı ilk kez sisteme giriş yaptığında, sunucu tarafından bir anahtar üretilir ve bu anahtar istemciye gönderilir. İstemci bu anahtarı saklar. İlk defa bir istek yapılacağı zaman, istemci sunucunun zaman bilgisini çekerek kendi zamanı ile arasındaki farkı hesaplar ve bu bilgiyi kaydeder. Bu işlem sadece ilk istekte yapılır. Daha sonra, her bir gerçek istek gönderilmeden önce, istemci bu zaman farkını kullanarak zamana bağlı yeni bir token hesaplar ve bu token'ı istekle birlikte sunucuya gönderir. Sunucu da, istek alındığı anda aynı token'ı üretir ve gelen token karşılaştırır. Bu yöntem, her iki tarafın da aynı algoritmayı kullanmasını gerektirir ve aynı anahtara sahip olmak çok önemlidir. Ayrıca, bu yöntemde anahtarın geçerlilik süresi sadece 5 saniye olduğundan (süre değiştirilebilir), bilgiler ele geçirilse bile replay saldırılarına karşı koruma sağlar.

Bu sistem, JWT Bearer ve Digest Authentication'ın bazı eksikliklerini kapatarak daha güvenli bir alternatif sunar. Örneğin JWT Bearer Auth yönteminde anahtar yaklaşık 1 gün boyunca geçerlidir. Eğer bu anahtar ele geçirilir ise 1 gün boyunca rızamız dışında bizim adımıza işlemler yapılabilir.

Digest Auth'un dez avantajı ise nonce değerinin hesaplanması için her seferinde sunucuya istek atılması gerekliliğidir. Bu da ağ kaynaklarının kullanımının artmasına sebep olmaktadır.



Şekil 6: Yöntem Sekans Diagramı

4.1 Avantajları

En büyük avantajı, ağı dinleyen birisi oluşturulan token'ı ele geçirse dahi ikinci bir istek attığında istek başarısız olacaktır. Ek olarak token, istemci tarafında üretildiğinden ve üretmek için her seferinde HTTP isteğine gerek olmadığından ağ kaynaklarını daha az ve verimli bir şekilde kullanmayı sağlamaktadır.

4.2 Dezavantajları

Kullanıcı giriş yaptığıında sunucu tarafından döndürülen anahtar değeri eğer ele geçirilir ise t anında üretilmesi gereken token tahmin edilebilir. Bu yüzden anahtar istemci tarafında güvenli bir şekilde saklanmalıdır. Fakat buna rağmen diğer yöntemlere göre avantajlıdır. Diğer yöntemlerde de aynı risk bulunmaktadır.

5 Yöntemin Test Edilmesi & İmplementasyonu - POC

Bu yöntemi test edebilmek adına bir istemci-sunucu projesi geliştirdim. Sunucu tarafında; Java 17, Spring Boot 3.0 Framework kullanılmaktadır. İstemci tarafında ise Javascript ve React framework kullanılmıştır.

5.1 Token Oluşturma Algoritması

Bu algoritma, zamana bağlı olarak tek kullanımlık şifreler (TOTP) üretir. Kullanıcıdan alınan bir anahtar ve unix sistem zamanına bağlı olarak 6 karakterli bir token üretilir. Anahtar ve zaman bilgisi, HMAC (Hash-based Message Authentication Code) kullanılarak birleştirilir. HmacSHA256 algoritması güvenilir ve yaygın olarak kullanılan bir hash fonksiyonudur.

Algoritma, verilen zamanı belirli bir yenileme süresine böler (örneğin her 5 saniyede bir) böylece zaman değeri daha az değişken hale gelir ve bu süreçte her kullanım için farklı bir değer üretilmiş olur. Bu zaman değeri, HMAC işlemi için anahtar ile birlikte kullanılır ve OTP üretilir. Bu yöntem, OTP'nin her defasında farklı ve tahmin edilemez olmasını sağlar. TOTP'nin geçerlilik süresi çok kısa tutulduğundan, bir saldırgan bu bilgiyi ele geçirse bile kullanımı sınırlı ve zordur. Bu, özellikle tekrar oynatma (replay) saldırılarına karşı ek güvenlik sağlar.

```
@Service
public class OTPService {

    private static final String HMAC_ALGO = "HmacSHA256";
    public static final int REFRESH_TIME = 5;

    public int generateOTP(String key, long time) throws NoSuchAlgorithmException, InvalidKeyException {
        Mac hmac = Mac.getInstance(HMAC_ALGO);
        SecretKeySpec keySpec = new SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), HMAC_ALGO);
        hmac.init(keySpec);

        byte[] timeBytes = ByteBuffer.allocate(8).putLong(time / REFRESH_TIME).array();
        byte[] hmacResult = hmac.doFinal(timeBytes);

        int offset = hmacResult[hmacResult.length - 1] & 0xf;
        int binary = ((hmacResult[offset] & 0x7f) << 24) | ((hmacResult[offset + 1] & 0xff) << 16)
            | ((hmacResult[offset + 2] & 0xff) << 8) | (hmacResult[offset + 3] & 0xff);

        return binary % 1000000;
    }

    public boolean validateOTP(String key, int otp) {
        long currentTime = Instant.now().getEpochSecond();
        try {
            int serverOTP = generateOTP(key, currentTime);
            return serverOTP == otp;
        } catch (NoSuchAlgorithmException | InvalidKeyException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Şekil 7: Java Kodu - Sunucu


```

import CryptoJS from "crypto-js";
import { axiosBasic } from "../axiosInterceptor";

export const REFRESH_TIME = 5;

export const generateOTP = async (key, time) => {
  // Create the HMAC hash
  const timeBytes = new ArrayBuffer(8);
  new DataView(timeBytes).setUint32(4, Math.floor(time / REFRESH_TIME), false);
  const timeWordArray = CryptoJS.lib.WordArray.create(
    new Uint8Array(timeBytes)
  );
  const keyWordArray = CryptoJS.enc.Utf8.parse(key);
  const hmacResult = CryptoJS.HmacSHA256(timeWordArray, keyWordArray);

  // Convert the HMAC hash to bytes
  const hmacBytes = CryptoJS.enc.Hex.parse(hmacResult.toString(CryptoJS.enc.Hex));
  const hmacArray = hmacBytes.words.map((word) => {return [
    (word >> 24) & 0xff, (word >> 16) & 0xff, (word >> 8) & 0xff, word & 0xff,
  ]}).flat();

  // Calculate the offset
  const offset = hmacArray[hmacArray.length - 1] & 0xf;

  // Extract the binary code
  const binary =
    ((hmacArray[offset] & 0x7f) << 24) |
    ((hmacArray[offset + 1] & 0xff) << 16) |
    ((hmacArray[offset + 2] & 0xff) << 8) |
    (hmacArray[offset + 3] & 0xff);

  // Return the OTP
  return binary % 1000000;
}

export const fetchServerTime = async () => {
  const response = await axiosBasic.get("/api/time");
  return parseInt(response.data, 10);
};

```

Şekil 8: Javascript Kodu - İstemci

5.2 Ağ Sebepli Gecikmelerden Dolayı Oluşabilecek Problemleri Önleme

Bazen ağ tabanlı, bazen ise belirlenen zaman diliminin sonuna denk gelinmesi sebebi ile token geçersi kılınabilmektedir. Bu gibi durumlarda ise istemci tarafından başarısız giden istekleri tekrarlayacak ve bu esnada yeniden token üretecek bir mekanizma geliştirilmiştir.

Aşağıdaki örnekte bir axios response yakalayıcı oluşturulup hata alındığında 3 defa istek tekrar yolanmaktadır.

```
axiosInstance.interceptors.response.use(
  (response) => response, // return the response if successful
  async (error) => {
    const config = error.config;
    // Initialize retry state if not already set
    if (!config._retry) {
      config._retry = true;
      config._retryCount = 0;
    }

    if (config._retryCount < 3) {
      config._retryCount++; // Increment retry count
      console.log(`Retry attempt #${config._retryCount}`); // Log retry attempt number

      try {
        await delay(1000); // Wait for 1000 milliseconds before retrying
        return await axiosInstance(config); // Retry the request
      } catch (err) {
        console.error("Error retrying the request:", err);
      }
    } else {
      console.error("Max retry attempts reached, redirecting to login.");
    }
    return Promise.reject(error);
  }
);
```

Şekil 9: Javascript retry kodu – İstemci

5.3 Uygulamadan Görseller

[Home](#) [Dashboard](#) [Login](#)

Login

Username:

Password:

[Login](#)

Şekil 10: Giriş Sayfası – İstemci

[Home](#) [Dashboard](#) [Otp Validator](#) [Log Out](#)

OTP Generator and Validator

Secret Key:

72723b19

fetch key from cookie

send request via OTP

OTP is valid!

generated header:

YWRtaW46NTU1OTQw

Generate otp header

Şekil 11: OTP-Validator Sayfası – İstemci

```
@Component
@RequiredArgsConstructor
@Slf4j
public class OtpFilter extends OncePerRequestFilter {

    private final UserDetailsServiceImpl userDetailsService;
    private final UserSecretRepository userSecretRepository;
    private final OTPService otpService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
    FilterChain filterChain) throws ServletException, IOException {
        String header = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (header == null) {
            filterChain.doFilter(request, response);
            return;
        }
        byte[] decodedBytes = Base64.getDecoder().decode(header);
        var decodedString = new String(decodedBytes);
        var username = decodedString.split(":")[0];
        var otp = Integer.parseInt(decodedString.split(":")[1]);
        var user = userDetailsService.loadUserByUsername(username);
        var userSecret = userSecretRepository.findByUserUsername(user.getUsername());
        if (userSecret.isEmpty()) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); return; }
        if (!otpService.validateOTP(userSecret.get().getSecret(), otp)) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); return; }

        log.info("successfully authenticated");
        filterChain.doFilter(request, response);
    }
}
```

Şekil 12: Request Filter - Sunucu

```

@RestController
@RequestMapping("/api")
@Slf4j
@RequiredArgsConstructor
public class OTPController {

    private final AuthenticationManager authenticationManager;
    private final UserRepository userRepository;
    private final UserSecretRepository userSecretRepository;

    @GetMapping("/say-hello")
    public ResponseEntity<Object> validateOTP(@RequestParam String name) {
        return ResponseEntity.ok("hello %s !".formatted(name));
    }

    @GetMapping("/time")
    public long serverTime() {
        var time = Instant.now().getEpochSecond();
        log.info("server time: {}", time);
        return time;
    }

    @PostMapping("/login")
    private ResponseEntity<Object> login(@RequestBody LoginRequest loginRequest) {
        var key = "";
        try {
            authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                    loginRequest.getUsername(), loginRequest.getPassword());
            log.info("Authentication successful {}", loginRequest.getUsername());

            key = UUID.randomUUID().toString().substring(0, 8);
            var user = userSecretRepository.findByUserUsername(loginRequest.getUsername());
            if (user.isPresent()) {
                user.get().setSecret(key);
                userSecretRepository.save(user.get());
            } else {
                UserSecret userSecret = new UserSecret();
                userSecret.setUser(userRepository.findByUsername(loginRequest.getUsername()).get());
                userSecret.setSecret(key);
                userSecretRepository.save(userSecret);
            }
        } catch (Exception e) {
            log.error("Authentication failed", e);
            return ResponseEntity.badRequest().body(Boolean.FALSE);
        }
        return ResponseEntity.ok(new LoginResponse(key));
    }
}

```

Şekil 13: Controller – Sunucu

Kaynak:

Jones, M. (2016). JSON Web Signature (JWS) Unencoded Payload Option.

<https://doi.org/10.17487/rfc7797>

Mahindrakar, P., & Pujeri, U. (2020). Insights of JSON Web Token. International Journal of Recent Technology and Engineering, 8(6), 1707–1710.

<https://doi.org/10.35940/ijrte.f7689.038620>

<https://medium.com/@jeevithajayakumar10/dive-into-http-authentication-schemes-basic-and-digest-85a0cf210240>

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., & Stewart, L. (1999). HTTP authentication: basic and Digest access authentication.

<https://doi.org/10.17487/rfc2617>

<https://medium.com/@raykipkorir/api-authentication-and-authorization-basic-authentication-jwt-oauth2-0-and-openid-connect-20aaeb5bf28b>