S3 Security Best Practices for Cleanit

🔒 The Correct Architecture

```
User Request → Backend API → Private S3 Bucket → CloudFront CDN → User
                    ↓
              Presigned URLs / CloudFront URLs
```

⚠️ Never Do This (Public Bucket)

```javascript
// ❌ WRONG - DON'T DO THIS
const params = {
  Bucket: BUCKET_NAME,
  Key: filePath,
  Body: file.buffer,
  ACL: 'public-read'  // ❌ Makes file publicly accessible
};
```

✅ Always Do This (Private Bucket)

```javascript
// ✅ CORRECT - Private bucket
const params = {
  Bucket: BUCKET_NAME,
  Key: filePath,
  Body: file.buffer,
  ServerSideEncryption: 'AES256'  // ✅ Private + encrypted
};
```

🎯 Implementation Options

Option 1: Private S3 + CloudFront (RECOMMENDED)

Best for: Production apps, scalable solutions, cost efficiency

Advantages:

- ✅ Secure (bucket is private)
- ✅ Fast (CloudFront CDN caching)
- ✅ Cheaper at scale (CloudFront pricing < S3 pricing)

- ✅ DDoS protection
- ✅ HTTPS by default

Setup:

1. Create private S3 bucket
2. Create CloudFront distribution
3. Set Origin Access Identity (OAI) so only CloudFront can access S3
4. Backend returns CloudFront URLs

Implementation:

```javascript
// Upload returns CloudFront URL
const imageUrl = await uploadToS3(file);
// Returns: https://d123456.cloudfront.net/uploads/2024/01/abc123.jpg

// Store this URL in database - it's already accessible through CloudFront
await prisma.campaign.create({
  data: {
    imageUrl: imageUrl  // CloudFront URL
  }
});
```

Terraform setup: See updated `main.tf` artifact

Option 2: Private S3 + Presigned URLs

Best for: MVP, simpler setup, private content

Advantages:

- ✅ Secure (bucket is private)
- ✅ Time-limited access
- ✅ No CloudFront setup needed
- ✅ Good for MVP

Disadvantages:

- ⚠️ Higher costs at scale (S3 data transfer pricing)
- ⚠️ No caching (slower for users)

- ⚠️ URLs expire (need regeneration)

Implementation:

```javascript
// Upload returns S3 key (not full URL)
const s3Key = await uploadToS3(file);
// Returns: uploads/2024/01/abc123.jpg

// Store S3 key in database
await prisma.campaign.create({
  data: {
    imageUrl: s3Key  // Just the key, not full URL
  }
});

// When serving to frontend, generate presigned URL
const campaign = await prisma.campaign.findUnique({ where: { id } });
campaign.imageUrl = await generatePresignedUrl(campaign.imageUrl, 3600); // 1 hour
```

Modified Campaign Controller:

```javascript
exports.getCampaignById = async (req, res, next) => {
  try {
    const campaign = await prisma.campaign.findUnique({
      where: { id: req.params.id }
    });

    // Generate presigned URL for the image
    campaign.imageUrl = await storageService.generatePresignedUrl(
      campaign.imageUrl,
      3600 // URL valid for 1 hour
    );

    res.json({ campaign });
  } catch (error) {
    next(error);
  }
};
```

Option 3: Backend Proxy (Most Secure, Complex)

Best for: Highly sensitive content, full access control

Advantages:

- ✅ Complete control over who accesses what
- ✅ Can add authentication, logging, watermarks
- ✅ No presigned URL expiration issues

Disadvantages:

- ❌ Backend handles all traffic (expensive, slower)
- ❌ More complex to implement
- ❌ Backend becomes bottleneck

Implementation:

```javascript
// New route in backend
router.get('/images/:key', authenticateToken, async (req, res) => {
  const s3Stream = s3.getObject({
    Bucket: BUCKET_NAME,
    Key: req.params.key
  }).createReadStream();

  s3Stream.pipe(res);
});

// Frontend uses: /api/images/uploads/2024/01/abc123.jpg
```

📊 Comparison Table

Feature	Public S3	CloudFront + Private S3	Presigned URLs	Backend Proxy
Security	❌ Poor	✅ Excellent	✅ Excellent	✅ Excellent
Speed	⚠️ OK	✅ Fast	⚠️ OK	❌ Slow
Cost (high traffic)	❌ High	✅ Low	⚠️ Medium	❌ Very High
Setup Complexity	✅ Simple	⚠️ Medium	✅ Simple	❌ Complex
Scalability	✅ Good	✅ Excellent	✅ Good	❌ Poor
Recommended	**Never**	**Production**	**MVP**	**Rarely**

🚀 Recommended Approach for Cleanit

Phase 1: MVP (Presigned URLs)

Start simple with presigned URLs:

- Private S3 bucket
- Generate presigned URLs on-demand
- Good enough for initial launch
- Easy to implement

Phase 2: Production (CloudFront)

Scale with CloudFront:

- Add CloudFront distribution
- Point to private S3 bucket
- Update storage service to return CloudFront URLs
- Better performance + lower costs

🔧 Implementation Steps

Step 1: Create Private S3 Bucket

```bash
aws s3api create-bucket \
  --bucket cleanit-uploads \
  --region ap-south-1 \
  --create-bucket-configuration LocationConstraint=ap-south-1

# Block all public access
aws s3api put-public-access-block \
  --bucket cleanit-uploads \
  --public-access-block-configuration \
  "BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=true"

# Enable encryption
aws s3api put-bucket-encryption \
  --bucket cleanit-uploads \
  --server-side-encryption-configuration \
  '{"Rules":[{"ApplyServerSideEncryptionByDefault":{"SSEAlgorithm":"AES256"}}]}'
```

Step 2: Verify Bucket is Private

```bash
# Should return: Public Access: Blocked
aws s3api get-public-access-block --bucket cleanit-uploads
```

Step 3: Update Backend Code

Use the updated `storageService.js` provided above.

Step 4: Test

```javascript
// Upload test
const result = await uploadToS3(testFile);
console.log(result); // Should be S3 key or CloudFront URL

// Try accessing directly (should fail if private)
// https://cleanit-uploads.s3.ap-south-1.amazonaws.com/uploads/test.jpg
// Expected: Access Denied

// Generate presigned URL (should work)
const url = await generatePresignedUrl(result);
console.log(url); // Should work for 1 hour
```

📝 Checklist for Production

- [ ] S3 Block Public Access enabled
- [ ] S3 bucket encryption enabled
- [ ] S3 versioning enabled (for backup)
- [ ] CloudFront distribution created
- [ ] CloudFront uses HTTPS only
- [ ] Origin Access Identity configured
- [ ] S3 bucket policy allows only CloudFront
- [ ] Backend stores CloudFront URLs (not S3 URLs)
- [ ] Removed any `ACL: 'public-read'` from code
- [ ] Test that direct S3 URLs don't work
- [ ] Test that CloudFront URLs work
- [ ] Set up CloudWatch alarms for S3 access
- [ ] Enable S3 access logging
- [ ] Configure CORS properly

💰 Cost Comparison (Example: 1TB transfer/month)

Method	Approximate Cost
Public S3	~$90/month
Private S3 + Presigned URLs	~$90/month
CloudFront + Private S3	~$60/month

Winner: CloudFront (33% cheaper + faster)

🔐 Security Best Practices

1. **Never use public buckets** for user-uploaded content
2. **Always encrypt** data at rest (use S3 encryption)
3. **Use HTTPS only** (CloudFront enforces this)
4. **Enable versioning** for backup/recovery
5. **Monitor access** with CloudWatch and CloudTrail
6. **Rotate credentials** regularly
7. **Use IAM roles** instead of hardcoded keys when possible
8. **Scan uploads** for malware (add to future roadmap)
9. **Set size limits** (already done in multer config)
10. **Validate file types** (already done in upload middleware)

🆘 Troubleshooting

"Access Denied" when accessing images

If using presigned URLs:

- Check URL hasn't expired
- Verify IAM permissions for backend
- Check bucket policy

If using CloudFront:

- Verify OAI is configured correctly
- Check S3 bucket policy allows CloudFront
- Wait 15-30 minutes for CloudFront distribution to deploy

Images load slowly

- Enable CloudFront caching
- Set appropriate Cache-Control headers
- Use CloudFront over S3 direct access

High AWS bills

- Check if bucket accidentally became public
- Enable CloudFront (cheaper than S3 at scale)
- Set lifecycle policies to delete old files
- Monitor with AWS Cost Explorer

📚 Additional Resources

- AWS S3 Security Best Practices
- CloudFront with S3
- S3 Presigned URLs

✅ Summary

For Cleanit MVP:

1. Use **PRIVATE S3 bucket** (never public)
2. Start with **presigned URLs** (simpler)
3. Migrate to **CloudFront** before production (better performance + cost)
4. Never set `ACL: 'public-read'` in your code

The updated code provided above implements this correctly! 🎉