Path Traversal in Signed URLs, Which Even Existed in the Official AWS SDK (English translation)

AWS SDK URL - Matsui, Eui Chul Chung, GMO Flatt Security Blog.

- Title in English: Path Traversal in Signed URLs, Which Even Existed in the Official AWS SDK
- Published: 2026-03-10
- Original: <https://blog.flatt.tech/entry/signed_url_path_traversal>
- Preserved from: https://blog.flatt.tech/entry/signed_url_path_traversal (stored) on 2026-08-14
- Licence: unknown

Content (translated into English)

Machine translation of `2026-gmo-flatt-security-blog-awssdkurl.md`, which holds the source's own words. Code, payloads, type names, URLs and CVE identifiers were masked before translating and restored after, so they are byte-identical to the original.

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

Hello. We are Matsui (@ryotaromosao) and Chung (Eui Chul Chung), security engineers at GMO Flatt Security. Have you heard of the “path traversal in presigned URLs” vulnerability? When implementing presigned URLs in web applications, developers often use an official AWS SDK. In the past, path traversal vulnerabilities have been found in the official SDKs themselves. On the other hand, there are also cases where the official SDK has implemented the correct countermeasures, but path traversal is caused by mistakes in the application developer’s implementation.

In this article, we will take an in-depth, code-based look at path traversal vulnerabilities in presigned URLs, including examples of vulnerabilities actually found in AWS SDKs. In the latter half, we will also introduce patterns of implementation mistakes made by application developers who use the SDKs. This article contains information that developers implementing presigned URLs can immediately apply in practice, so please read on.

The content of this blog was also presented at JAWS DAYS 2026. The slides are publicly available as well, so please refer to them alongside this article.

Disclaimer

This article was written for the purpose of broadly sharing security knowledge and does not encourage attacks such as the exploitation of vulnerabilities. Attacking a product without permission may constitute a crime. We accept no responsibility whatsoever for actions taken by referring to or imitating the information provided by our company.

About presigned URLs

A presigned URL is a **URL that grants temporary access to an object on Amazon S3**. Normally, operating on an object in S3 requires using IAM credentials with the necessary permissions or temporary credentials obtained by assuming the relevant role through AssumeRole.

However, there are cases where you want to temporarily make an object in S3 public or temporarily allow someone without IAM credentials to download or upload an object. Presigned

URLs are used in such cases. Presigned URLs allow objects to be transferred directly between a client and S3 without passing through an application server, offering the benefit of significantly reducing server load. For this reason, they are used in many web applications.

An important characteristic of presigned URLs is that **access is permitted only when the issuer of the URL—the IAM permissions used for signing—has the permissions required for the target operation**. Therefore, for example, even if a general user receives a presigned URL for downloading an object in `my-flatt-bucket`, the download will succeed only if, as shown below, the application server that issued the URL has the permission (`s3:GetObject`) required to download the object in `my-flatt-bucket`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDownloadFromMyFlattBucket",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::my-flatt-bucket/*"
      ]
    }
  ]
}
```

Presigned URL formats

Presigned URLs issued for S3 come in the following two formats. The first is the **path-style format**, in which the bucket name is placed in the URL path. The second is the **virtual-hosted-style format**, in which the bucket name is placed as part of the subdomain. Regarding the path-style format, as stated in the official AWS blog, AWS originally planned to "discontinue support for path-style URLs for buckets created after September 30, 2020." However, in response to user feedback, including SSL/TLS certificate validation challenges in environments where bucket names contain dots (`.`), this complete discontinuation has now been postponed.

Therefore, although it is still technically possible to generate and use path-style URLs, note that this functionality is already considered legacy.

Format	Example
Path-style format	`https://s3.${region-code}.amazonaws.com/${bucket-name}/${object-key}`
Virtual-hosted-style format	`https://${bucket-name}.s3.${region-code}.amazonaws.com/${object-key}`

Path traversal in presigned URLs

S3's flat structure

First, let us briefly discuss S3's specifications. S3 does not have a directory structure like an operating system's file system; instead, it manages data in a **flat structure**.

The official documentation also explains this as follows. The folder-like display shown in the management console merely uses slashes (`/`) as separators to make objects easier for people to organize and understand.

>

In Amazon S3 general purpose buckets, objects are the primary resources, and objects are stored in buckets. Amazon S3 general purpose buckets have a flat structure instead of a hierarchy like you would see in a file system. However, for the sake of organizational simplicity, the Amazon S3 console supports the folder concept as a means of grouping objects. (In Amazon S3 general purpose buckets, objects are the primary resources, and objects are stored in buckets. Amazon S3 general purpose buckets have a flat structure and do not have the kind of hierarchy found in a file system. However, to make the structure easier to understand, the Amazon S3 console supports the concept of folders as a way to group objects.)

A typical path traversal attack occurs by using a relative path such as `../` to move up to a parent directory. However, in S3's flat structure, even if `../` is entered, S3 does not resolve it as a parent directory; it is merely treated as an "object name containing the string `../`." Therefore, due to S3's specifications, path traversal of the kind known at the operating-system level is not considered possible.

Path traversal in presigned URLs

So, what kind of vulnerability is path traversal in presigned URLs? As described above, S3 itself has a flat structure and treats object keys containing `../` as literal strings. However, if **path normalization** is performed by the SDK's internal processing or by application code while generating a presigned URL, `../` may be resolved, potentially resulting in a presigned URL being issued for an object other than the one originally intended.

As a concrete example, consider a multi-tenant web application that manages objects using a separate prefix in an S3 bucket for each tenant and issues presigned download URLs based on object keys supplied as user input.

```
func generatePresignedURL(tenantID, userInputFilename string) (string, error) {
  objectKey := fmt.Sprintf("tenants/%s/files/%s", tenantID, userInputFilename)

  req, _ := s3Client.GetObjectRequest(&s3.GetObjectInput{
    Bucket: aws.String("my-flatt-bucket"),
    Key:    aws.String(objectKey),
  })

  return req.Presign(15 * time.Minute)
}
```

In this implementation, user input is passed to `userInputFilename`. On the normal path, for example, if `userInputFilename = "report.pdf"` is provided as input, the object key becomes `tenants/my-tenant/files/report.pdf`, and a presigned URL is issued for an object in the user's own tenant as intended.

However, if **path normalization** is performed internally by the SDK or in the application code, when an attacker specifies `userInputFilename= "../../other-tenant/files/secret.txt"`, a presigned URL is issued for an unintended object under `/other-tenant/files`, as shown below.

Step	Object key state
Before normalization	`tenants/my-tenant/files/../../other-tenant/files/secret.txt`
After normalization	`tenants/other-tenant/files/secret.txt`

Because S3 itself does not resolve `../`, if normalization does not occur, the string `tenants/my-tenant/files/../../other-tenant/files/secret.txt` is treated as the object key as-is, and access fails unless a corresponding object exists. However, if normalization occurs, a valid presigned URL is issued for `tenants/other-tenant/files/secret.txt`, which may actually exist.

Thus, path traversal in presigned URLs is not a vulnerability in S3 itself, but is caused by path normalization occurring during the process of generating the presigned URL.

The example above uses tenant isolation by prefix within the same bucket, but in the AWS SDK cases introduced below, we will also present a case in which path traversal made it possible to **issue presigned URLs across bucket boundaries**.

Path Traversal in AWS SDKs

From here, we will examine specifically how path traversal can occur in actual AWS SDKs. This blog covers AWS SDK for Go (v1) and AWS SDK for JavaScript (v3).

Path Traversal in S3 Presigned URLs in Go (v1)

We will begin with AWS SDK for Go (v1). In Go (v1), when a presigned URL is issued, the final URL is assembled through several internal SDK steps. The scope of the impact of path traversal varies significantly depending on the URL format determined during this assembly process: path-style or virtual-hosted–style.

Codebase Details

Let us follow the internal process step by step.

1. Creation of the `Request` object

Generation of a presigned URL begins by creating the `Request` object for the target operation. For example, for GetObject, `GetObjectRequest()` in the following code is called. Here, the template `HTTPPath: "/{Bucket}/{Key+}"` is joined directly to the path portion of the endpoint URL (for example, https://s3.us-east-1.amazonaws.com)) by the internal processing of `c.newRequest()`. At this stage, actual values have not yet been assigned to `{Bucket}` or `{Key+}`.

`aws-sdk-go-main/service/s3/api.go` lines 4976–4990 (v1.55.8)

```
func (c *S3) GetObjectRequest(input *GetObjectInput) (req *request.Request, output *GetObjectOutput) {
  op := &request.Operation{
    Name:       opGetObject,
    HTTPMethod: "GET",
    HTTPPath:   "/{Bucket}/{Key+}",
  }

  if input == nil {
    input = &GetObjectInput{}
  }

  output = &GetObjectOutput{}
  req = c.newRequest(op, input, output)
  return
}
```

1. Determination of Path-Style / Virtual-Hosted–Style

Next, `Sign()` is called within `Presign()`, and the URL is assembled within `r.Build()`.

`aws-sdk-go-main/aws/request/request.go` lines 436–447

```
func (r *Request) Sign() error {
  r.Build()
  if r.Error != nil {
    debugLogReqError(r, "Build Request", notRetrying, r.Error)
    return r.Error
  }

  SanitizeHostForHeader(r.HTTPRequest)

  r.Handlers.Sign.Run(r)
  return r.Error
}
```

Within this process, it first determines whether the presigned URL should use path-style or virtual-hosted–style format. This is performed by `endpointHandler()` and is determined according to the following conditions.

Condition	URL format
`S3ForcePathStyle == true`	Path-style
`S3ForcePathStyle == false` and the bucket name is not DNS-compatible	Path-style
`S3ForcePathStyle == false` and the bucket name is DNS-compatible	Virtual-hosted–style

By default, `S3ForcePathStyle` is set to `false`, and when the bucket name is not DNS-compatible, the presigned URL is issued in path-style format.

Conditions that make a bucket name DNS-incompatible include the bucket name being in IP address format (for example, `192.0.2.1`), the bucket name containing `..`, or the scheme being HTTPS while the bucket name contains `.`. Regarding the third condition in particular, because the Go SDK (v1) issues presigned URLs over HTTPS by default, a presigned URL is issued in path-style format when the bucket name contains a dot.

`aws-sdk-go-main/service/s3/endpoint.go` lines 115–120

```
func endpointHandler(req *request.Request) {
  endpoint, ok := req.Params.(endpointARNGetter)
  if !ok || !endpoint.hasEndpointARN() {
    updateBucketEndpointFromParams(req)
    return
  }
```

`aws-sdk-go-main/service/s3/host_style_bucket.go` lines 36–49

```
func updateEndpointForS3Config(r *request.Request, bucketName string) {
   forceHostStyle := aws.BoolValue(r.Config.S3ForcePathStyle)
   accelerate := aws.BoolValue(r.Config.S3UseAccelerate)

   if accelerate && accelerateOpBlacklist.Continue(r) {
      if forceHostStyle {
         if r.Config.Logger != nil {
            r.Config.Logger.Log("ERROR: ...")
         }
      }
      updateEndpointForAccelerate(r, bucketName)
   } else if !forceHostStyle && r.Operation.Name != opGetBucketLocation {
      updateEndpointForHostStyle(r, bucketName)
   }
}
```

When virtual-hosted–style is selected, `moveBucketToHost()` moves the bucket name to the subdomain, and `removeBucketFromPath()` removes `/{Bucket}` from the path.

`aws-sdk-go-main/service/s3/host_style_bucket.go` lines 133–137

```
func moveBucketToHost(u *url.URL, bucket string) {
  u.Host = bucket + "." + u.Host
  removeBucketFromPath(u)
}
```

1. Parameter Expansion and `path.Clean()`

Finally, `rest.Build()` is executed. The `buildURI()` called in this process embeds the actual bucket name and object key into the template.

`aws-sdk-go-main/private/protocol/rest/build.go` lines 201–216

```
func buildURI(u *url.URL, v reflect.Value, name string, tag reflect.StructTag) error {
  value, err := convertType(v, tag)

  u.Path = strings.Replace(u.Path, "{"+name+"}", value, -1)
  u.Path = strings.Replace(u.Path, "{"+name+"+}", value, -1)

  u.RawPath = strings.Replace(u.RawPath, "{"+name+"}", EscapePath(value, true), -1)
  u.RawPath = strings.Replace(u.RawPath, "{"+name+"+}", EscapePath(value, false), -1)

  return nil
}
```

Immediately after parameter expansion, `cleanPath()` is called. `path.Clean()`, which is called within `cleanPath()`, is part of Go's standard library and performs path normalization. As a result, a presigned URL is issued after `../` contained in the bucket name or object key has been resolved.

`aws-sdk-go-main/private/protocol/rest/build.go` lines 137–139

```
  if !aws.BoolValue(r.Config.DisableRestProtocolURICleaning) {
    cleanPath(r.HTTPRequest.URL)
  }
```

`aws-sdk-go-main/private/protocol/rest/build.go` lines 247–258

```
func cleanPath(u *url.URL) {
  hasSlash := strings.HasSuffix(u.Path, "/")

  u.Path = path.Clean(u.Path)
  u.RawPath = path.Clean(u.RawPath)

  if hasSlash && !strings.HasSuffix(u.Path, "/") {
    u.Path += "/"
    u.RawPath += "/"
  }
}
```

1. Signing the URL

Finally, `r.Handlers.Sign.Run(r)` within `Sign()` is called, and the assembled URL is signed.

`aws-sdk-go-main/aws/request/request.go` lines 436–447

```
func (r *Request) Sign() error {
  r.Build()
  if r.Error != nil {
    debugLogReqError(r, "Build Request", notRetrying, r.Error)
    return r.Error
  }

  SanitizeHostForHeader(r.HTTPRequest)

  r.Handlers.Sign.Run(r)
  return r.Error
}
```

Specific Examples of Path Traversal

Based on the URL assembly and normalization by `path.Clean()` described in the preceding steps, let us examine how specific payloads cause path traversal.

- Path-style

In path-style format, if `foo/../bar` is specified as the object key, normalization by `path.Clean()` results in a presigned URL being issued for the object key `bar` within the same bucket. At this point, because arbitrary objects within the bucket can be operated on, this becomes a problem when tenants are managed using prefixes within the same bucket, as described earlier.

| Step | Path state | | | **After endpointHandler()** | `/{Bucket}/{Key+}` | | | **After buildURI()** Bucket=my-flatt-bucket, Key=foo/../bar | `/my-flatt-bucket/foo/../bar` | | | **After path.Clean()** | `/my-flatt-bucket/bar` | |

More notably, path traversal may make it possible to **operate on objects across bucket boundaries**. For example, if `../other-flatt-bucket/secret.txt` is specified as the object key, normalization changes the path as follows.

| Step | Path state | | | **After endpointHandler()** | `/{Bucket}/{Key+}` | | | **After buildURI()** Bucket=my-flatt-bucket, Key=../other-flatt-bucket/secret.txt | `/my-flatt-bucket/../other-flatt-bucket/secret.txt` | | | **After path.Clean()** | `/other-flatt-bucket/secret.txt` | |

It can be seen that after `path.Clean()`, a presigned URL can be issued for `secret.txt` in `other-flatt-bucket`. In other words, if a web application uses user input as the object key, an attacker can specify a payload like the one above as the object key to issue a presigned URL for an object in another bucket.

However, whether the generated URL can actually be used to download or upload an object depends on the policy used to create the URL. For example, if an IAM policy grants broad permissions spanning multiple buckets, as shown below, objects in other buckets that should not be accessible may be exposed, or it may be possible to upload objects to other buckets.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDownloadAndUploadFromMultipleBuckets",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::my-flatt-bucket/*",
        "arn:aws:s3:::other-flatt-bucket/*"
      ]
    }
  ]
}
```

- Virtual-hosted–style

For virtual-hosted–style URLs, in which the bucket name is treated as a subdomain, traversal into another bucket is not possible because the hostname portion is unaffected by `path.Clean()`. However, path traversal of object keys within the same bucket still occurs in the same way.

| Step | Host | Path | | | **After endpointHandler()** | `my-flatt-bucket.s3.amazonaws.com` | `/{Key+}` | | | **After buildURI()** (Key=foo/../bar) | `my-flatt-bucket.s3.amazonaws.com` | `/foo/../bar` | | | **After path.Clean()** | `my-flatt-bucket.s3.amazonaws.com` | `/bar` | |

Correspondence with the AWS Security Team

We reported this vulnerability to the AWS security team. The email we actually received in response is shown below. (We obtained permission from AWS to publish the contents of the email.)

>

We are aware of many customers who have built up an implicit dependency on this behavior of path normalization. For that reason, we have retained this behavior by default for AWS SDK for Go V1 to maintain backward compatibility for customers that have older or difficult to update solutions that depend on this behavior. (We recognize that many customers implicitly depend on this path-normalization behavior. Therefore, AWS SDK for Go V1 retains this behavior by default to maintain backward compatibility for customers using older solutions or solutions that are difficult to update and depend on this behavior.)

In short, some customers depend on paths being normalized by default, so AWS said that it would continue to retain this behavior in order to maintain backward compatibility.

Mitigation

As described above, the cause is unintended path normalization performed internally by the SDK through `path.Clean()`. AWS SDK for Go (v1) provides the `DisableRestProtocolURICleaning` configuration option, so when normalization is unnecessary, enabling this setting disables it. With this setting, even if user input contains `../`, it is not normalized and is instead treated as an ordinary string, making it possible to prevent path traversal.

`aws-sdk-go-main/aws/config.go` lines 513–516

```
func (c *Config) WithDisableRestProtocolURICleaning(t bool) *Config {
  c.DisableRestProtocolURICleaning = &t
  return c
}
```

Summary

So far, we have examined path traversal in AWS SDK for Go (v1). Go (v1) is currently archived, and migration to AWS SDK for Go (v2) is recommended, so it is less likely to be adopted for new development. However, many systems built in the past are still in operation. In situations where an attacker can manipulate the object key, if a path-style signed URL is issued under certain configuration and bucket-name conditions, this creates a security risk that could allow unintended operations on other buckets.

Path Traversal in CloudFront Signed URLs in JavaScript (v3)

The issuance of CloudFront signed URLs by the `@aws-sdk/cloudfront-signer` package (AWS SDK for JavaScript v3) was also vulnerable to path traversal due to unintended path normalization by JavaScript's `URL` constructor.

Codebase Details

We will trace the internal processing in order.

1. Determining `baseUrl`

Signed URL generation begins by determining `baseUrl`. If `url` is passed as an argument to `getSignedUrl()`, `url` is assigned directly to `baseUrl`; if `policy` is passed, the resource URL is extracted from that argument and assigned to `baseUrl`.

`aws-sdk-js-v3/packages/cloudfront-signer/src/sign.ts` lines 113–124 (before the fix)

```
let baseUrl: string | undefined;
if (url) {
 baseUrl = url;
} else if (policy) {
 const resources = getPolicyResources(policy!);
 if (!resources[0]) {
  throw new Error(
   "@aws-sdk/cloudfront-signer: No URL provided and unable to determine URL from first policy statement resource."
  );
 }
 baseUrl = resources[0].replace("*://", "https://");
}
```

1. URL Normalization and Parameter Expansion

Next, a `URL` object is created with `new URL(baseUrl!)`, and signing parameters are added to the resulting object.

`aws-sdk-js-v3/packages/cloudfront-signer/src/sign.ts` lines 126–131 (before the fix)

```
const newURL = new URL(baseUrl!);
newURL.search = Array.from(newURL.searchParams.entries())
 .concat(Object.entries(cloudfrontSignBuilder.createCloudfrontAttribute()))
 .filter(([, value]) => value !== undefined)
 .map(([key, value]) => `${encodeURIComponent(key)}=${encodeURIComponent(value)}`)
 .join("&");
```

During this flow, automatic path normalization is performed by the `URL` constructor. As a result, `../` within the path is interpreted as a relative path and resolved (normalized) as movement to the parent directory.

Concrete Example of Path Traversal

If `foo/../bar` is specified as the URL path, normalization by the `URL` constructor results in a signed URL being generated for the path `bar`.

Step	URL State
When getSignedUrl is called (url=https://flatt-distribution.cloudfront.net/foo/../bar	`https://flatt-distribution.cloudfront.net/foo/../bar`
After new URL()	`https://flatt-distribution.cloudfront.net/bar`

At first glance, normalizing a URL when signing the “URL” of a CloudFront distribution may seem like natural behavior. However, CloudFront is also intended for use with S3, in which case the URL path component is interpreted as an S3 object key. Meanwhile, as described above, S3 uses a flat data model, so object keys containing strings such as `../` are permitted.

This discrepancy in the interpretation of resource paths may permit access to unintended resources even during legitimate, non-malicious use of CloudFront. Although not a common

scenario, one possible example is that a signature intended for a valid S3 object whose key contains a relative path could, due to URL normalization, become a signature for a different object located in the parent folder.

Correspondence with the AWS Security Team

After we reported this issue to the AWS security team, we received the following response.

>

I would like to inform you that our CNA team has evaluated your reported issue for a CVE/GHSA assignment and determined that this does not qualify for a CVE under our program [1] as this issue requires overly permissive S3 bucket policies to be applied. The configuration of these policies fall under the customer side of the AWS Shared Responsibility Model [2]. (After evaluating the reported issue for a CVE/GHSA assignment, we determined that it does not qualify for a CVE. Exploiting this issue requires overly permissive S3 bucket policies to be applied, and configuring those policies falls within the customer's area of responsibility under the AWS Shared Responsibility Model.)

Because the matter falls within the developer's area of responsibility, it was not formally recognized as a vulnerability. Nevertheless, the report was accepted and led to the removal of the normalization behavior in `v3.858.0` of `@aws-sdk/cloudfront-signer`.

Mitigation

In `v3.858.0` of `@aws-sdk/cloudfront-signer`, the implementation was changed to discontinue use of the `URL` constructor and instead construct URLs through string operations.

`aws-sdk-js-v3/packages/cloudfront-signer/src/sign.ts` lines 139–146 (after the fix)

```
const startFlag = baseUrl!.includes("?") ? "&" : "?";
const params = Object.entries(cloudfrontSignBuilder.createCloudfrontAttribute())
 .filter(([, value]) => value !== undefined)
 .map(([key, value]) => `${encodeURIComponent(key)}=${encodeURIComponent(value)}`)
 .join("&");
const urlString = baseUrl + startFlag + params;

return getResource(urlString);
```

Therefore, when using `@aws-sdk/cloudfront-signer` earlier than `v3.858.0`, path traversal can be avoided by updating the package to a version containing the fix or later.

Summary

This concludes our explanation of the path traversal that occurred in `@aws-sdk/cloudfront-signer` in AWS SDK for JavaScript v3. The root cause of this issue can be described as a difference in specifications between services using different data models: CloudFront, which assumes URL-based access, and S3, which has a flat data structure. Vulnerabilities caused by this kind of

mismatch in specifications between services may also exist in other AWS service integrations, so they warrant closer attention in future research.

Patterns in Which Application Developers Normalize Paths by Mistake

So far, we have examined cases in which path normalization occurs within AWS SDKs. However, even when the SDK is implemented correctly, developers calling the SDK may themselves mistakenly perform normalization in their application code. Here, we introduce three representative patterns.

Pattern Involving Explicit Normalization

The first case is where a developer calls a function that explicitly normalizes the path. This is the same pattern as `path.Clean()` in AWS SDK Go (v1), discussed earlier. Because S3 has a flat structure, normalizing with functions such as those below can create a signed URL for an object other than the one originally intended, resulting in path traversal.

- `path.normalize()` (JavaScript)
- `os.path.normpath()` (Python)
- `java.nio.file.Path.normalize()` (Java)

Sample code in `path.normalize()`

```
const path = require("path");
const { S3Client, GetObjectCommand } = require("@aws-sdk/client-s3");
const { getSignedUrl } = require("@aws-sdk/s3-request-presigner");

const s3Client = new S3Client({ region: "ap-northeast-1" });

async function generatePresignedUrlWithNormalize(tenantId, userInputPath) {

 const rawPath = `tenants/${tenantId}/files/${userInputPath}`;

 const objectKey = path.normalize(rawPath);

 const command = new GetObjectCommand({
   Bucket: process.env.BUCKET_NAME,
   Key: objectKey,
});

return await getSignedUrl(s3Client, command, { expiresIn: 3600 });
}
```

Pattern Involving Normalization When Joining Paths

The second case is where a path-joining function is used when joining a directory and a file. These functions are designed to normalize paths internally at runtime in order to resolve redundant path notation (such as `//` and `./`). Therefore, if the user input contains `../`, the path will be normalized.

- `path.join()` (JavaScript)
- `path.Join()`, `filepath.Join()` (Go)

Sample code in `path.join()`

```
const path = require("path");
const { S3Client, GetObjectCommand } = require("@aws-sdk/client-s3");
const { getSignedUrl } = require("@aws-sdk/s3-request-presigner");

const s3Client = new S3Client({ region: "ap-northeast-1" });

async function generatePresignedUrlWithPathJoin(tenantId, userInputFilename) {

const objectKey = path.join("tenants", tenantId, "files", userInputFilename);

 const command = new GetObjectCommand({
   Bucket: process.env.BUCKET_NAME,
   Key: objectKey,
 });

 return await getSignedUrl(s3Client, command, { expiresIn: 3600 });
}
```

Pattern Involving Normalization When Constructing a URL

The third case is where, when generating a URL object by appending a user-supplied path to a base URL, relative paths such as `../` are interpreted internally and normalization is performed.

In particular, when issuing a CloudFront signed URL, the complete URL string must be passed as the signing target. Therefore, if a developer uses functions such as those below while constructing the URL, internal normalization may cause a signed URL to be issued for an unintended object key.

- `new URL()` (JavaScript)
- `java.net.URI.resolve()` (Java)
- `url.URL.ResolveReference()` (Go)

Sample code in `new URL()`

```
const { getSignedUrl } = require("@aws-sdk/cloudfront-signer");

function generateCloudFrontSignedUrl(userInputPath) {

 const rawUrl = "https://d111111abcdef8.cloudfront.net/tenants/my-tenant/files/" + userInputPath;

 const url = new URL(rawUrl);

 return getSignedUrl({
  url: url.toString(),
  keyPairId: "dummyKeyPairId",
  privateKey: dummyPrivateKey,
  dateLessThan: new Date(Date.now() + 3600 * 1000).toISOString(),
 });
}
```

Conclusion

This blog post covered path traversal vulnerabilities in signed URLs. We introduced cases where this vulnerability existed in the official AWS SDKs for Go (v1) and JavaScript (v3), as well as three patterns in which path traversal can be caused by implementation mistakes made by application developers even though mitigations have been implemented on the SDK side. We hope this encourages those who are about to implement functionality using signed URLs, or who already operate such functionality, to give some thought to path traversal and verify that their implementation is appropriate.

Security Services for Development Organizations from GMO Flatt Security