PRESENTS

Dapr security audit

In collaboration with the Dapr maintainers, Open Source Technology Improvement Fund and The Linux Foundation

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 6th September 2023

This report is licensed under Creative Commons 4.0 (CC BY 4.0)
## Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of contents</td>
<td>1</td>
</tr>
<tr>
<td>Executive summary</td>
<td>2</td>
</tr>
<tr>
<td>Project Summary</td>
<td>3</td>
</tr>
<tr>
<td>Audit Scope</td>
<td>4</td>
</tr>
<tr>
<td>Threat model</td>
<td>5</td>
</tr>
<tr>
<td>Fuzzing</td>
<td>15</td>
</tr>
<tr>
<td>Issues found</td>
<td>17</td>
</tr>
<tr>
<td>SLSA</td>
<td>43</td>
</tr>
<tr>
<td>Supply-chain mitigations</td>
<td>45</td>
</tr>
</tbody>
</table>
Executive summary

In May and June 2023, Ada Logics carried out a security audit for the Dapr project. The high-level goal was to complete a holistic audit drawing on several different security disciplines. The audit was split into the following goals:

1. Formalise a threat model of the code assets in scope.
2. Do a manual code audit of the code assets in scope.
3. Evaluate Daprs fuzzing suite against the formalised threat model.
4. Perform a SLSA review of Dapr.

Our overall assessment of Dapr is highly positive. Dapr follows security best practices in both design and implementation. Dapr performed well in this audit demonstrating a strong security posture.

The audit found 7 issues, of which 4 are umbrella issues covering multiple cases of similar issues across different components in the same Dapr building blocks. None of the issues were of critical or high severity. We found a vulnerability in a 3rd-party dependency which was assigned a CVE\(^1\) of high severity, however it did not impact Dapr in a critical or high severity manner, and affects only a small group of Dapr users in a component that is not enabled by default. The vulnerability had the potential to crash a Dapr sidecar with an out-of-memory denial of service attack vector. We found the vulnerability after performing the threat modelling goal and understanding the flow of untrusted data through a Dapr deployment, and then adding a fuzzer for the affected component.

We added a total of five fuzzers to Daprs OSS-Fuzz integration. These will continue to run continuously after the conclusion of the audit.

An area for future work on Daprs security posture is its software supply-chain. The SLSA review showed that Dapr is lacking a compliant provenance attestation alongside release artifacts but performs well with regards to its build and maintenance processes. We have included recommendations on generating provenance with releases using SLSA-provided builders. We also included recommendations on how Dapr can ensure the quality and integrity of its own supply-chain via its dependency tree.

\(^1\) CVE-2023-37475
Results summarised

7 security issues found
All issues except for 1 have been fixed
Five fuzzers added to Daprs fuzzing suite
1 CVE assigned
Threat model included in report
SLSA compliance review included in report
Supply-chain threat mitigation advice included in report

Project Summary

The auditors of Ada Logics were:

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam Korczynski</td>
<td>Security Engineer, Ada Logics</td>
<td><a href="mailto:Adam@adalogics.com">Adam@adalogics.com</a></td>
</tr>
<tr>
<td>David Korczynski</td>
<td>Security Researcher, Ada Logics</td>
<td><a href="mailto:David@adalogics.com">David@adalogics.com</a></td>
</tr>
</tbody>
</table>

The Dapr community members involved in audit were:

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yaron Schneider</td>
<td>Maintainer and steering committee member</td>
<td><a href="mailto:yaron@diagrid.io">yaron@diagrid.io</a></td>
</tr>
<tr>
<td>Alessandro Segala</td>
<td>Dapr maintainer</td>
<td><a href="mailto:asegalaw@microso.com">asegalaw@microso.com</a></td>
</tr>
<tr>
<td>Artur Souza</td>
<td>Dapr maintainer</td>
<td><a href="mailto:artur@diagrid.io">artur@diagrid.io</a></td>
</tr>
<tr>
<td>Bernd Verst</td>
<td>Dapr maintainer</td>
<td><a href="mailto:bernd.verst@microso.com">bernd.verst@microso.com</a></td>
</tr>
</tbody>
</table>

The following facilitators of OSTIF were engaged in the audit:

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Email</th>
</tr>
</thead>
</table>

ADA Logics
Audit Scope

The following assets were in scope of the audit.

<table>
<thead>
<tr>
<th>Repository</th>
<th><a href="https://github.com/dapr/dapr">https://github.com/dapr/dapr</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Go</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repository</th>
<th><a href="https://github.com/dapr/components-contrib">https://github.com/dapr/components-contrib</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Go</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repository</th>
<th><a href="https://github.com/dapr/kit">https://github.com/dapr/kit</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Go</td>
</tr>
</tbody>
</table>
**Threat model**

Dapr is a framework for building cloud-native applications. It consists of a runtime and a set of building blocks that allow users to move infrastructure-related tasks out of their applications into cloud infrastructure. A benefit of this is that users can spend time and resources on the value-proposition of their applications without having to develop infrastructure.

Users can deploy Dapr either in 1) self-hosted mode directly on a virtual machine or 2) on Kubernetes. When using Dapr with Kubernetes, Dapr is deployed as a sidecar container in the same pod as the user’s application. When running Dapr on a virtual machine, Dapr runs as a separate sidecar process. In both cases, the application and Dapr interact through HTTP or gRPC calls:

If the user has multiple applications running with Dapr, each has a sidecar next to it:

Dapr comes with a set of built-in components - a form of cloud-native primitives - that each enables common infrastructure-related functionality necessary when building microservice-based applications. The user application interacts with these components via HTTP or gRPC API endpoints. Dapr groups components into building blocks; A building block is a high-level category of a common cloud-native problem, and it consists of a
series of components that solve that problem in different ways. Dapr has ten building blocks:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Service-to-service invocation</td>
<td>A series of HTTP or gRPC endpoints for applications to communicate with each other.</td>
</tr>
<tr>
<td>2</td>
<td>State management</td>
<td>A key/value-based state and query API for managing information in long-running stateful services.</td>
</tr>
<tr>
<td>3</td>
<td>Publish and subscribe</td>
<td>A messaging platform to send (publish) messages to a topic. Subscribers subscribe to a topic to receive messages.</td>
</tr>
<tr>
<td>4</td>
<td>Bindings</td>
<td>A bi-directional connection to an external service.</td>
</tr>
<tr>
<td>5</td>
<td>Actors</td>
<td>An independent unit of computation.</td>
</tr>
<tr>
<td>6</td>
<td>Observability</td>
<td>A series of components for logging, tracing, monitoring, emission of metrics.</td>
</tr>
<tr>
<td>7</td>
<td>Secrets</td>
<td>For managing secrets.</td>
</tr>
<tr>
<td>8</td>
<td>Configuration</td>
<td>For retrieving application configurations from configuration stores.</td>
</tr>
<tr>
<td>9</td>
<td>Distributed Lock</td>
<td>An API used to lock a resource, so multiple instances of an application can access the resource with guaranteed consistency.</td>
</tr>
<tr>
<td>10</td>
<td>Workflows</td>
<td>Used to define long-running processes across multiple microservices.</td>
</tr>
</tbody>
</table>

Illustrated high-level overview

Having outlined the main parts of Dapr, the runtime and the components, we can look at a high-level view of Dapr:
At the top is the user’s application. This is the application that the user deploys Dapr with. The application communicates with the Dapr sidecar via either HTTP or gRPC. The Dapr sidecar invokes the Dapr components which interact with cloud services which are illustrated at the very bottom of the diagram.

**daprd**

The Dapr sidecar process is called **daprd** and is implemented in [github.com/dapr/dapr/cmd/daprd](https://github.com/dapr/dapr/tree/ddd11bcc07ddf61bf5edd835a4b621a3ef1d395a/cmd/daprd). We now do a quick code walk through of how Dapr starts the runtime and sets up the HTTP and gRPC endpoints and the components. The purpose of this brief section is to map the high-level view from the previous section to the Dapr codebase.

The **daprd** main function has two main purposes: 1) To instantiate a new DaprRuntime and 2) start it. DaprRuntime is the service that the users application communicates with. **daprd** creates the DaprRuntime with a call to [github.com/dapr/dapr/pkg/runtime.FromFlags()](https://github.com/dapr/dapr/blob/ddd11bcc07ddf61bf5edd835a4b621a3ef1d395a/cmd/daprd/main.go#L48)

```go
func main() {
    // set GOMAXPROCS
    _, _ = maxprocs.Set()
    rt, err := runtime.FromFlags(os.Args[1:]);
    if err != nil {
```

2 [https://github.com/dapr/dapr/tree/ddd11bcc07ddf61bf5edd835a4b621a3ef1d395a/cmd/daprd](https://github.com/dapr/dapr/tree/ddd11bcc07ddf61bf5edd835a4b621a3ef1d395a/cmd/daprd)
log.Fatal(err)
}

Which returns a DaprRuntime:
https://github.com/dapr/dapr/blob/d5f9625cf94e3b032759d7ef35a5256287c183cd/pkg/runtime/cli.go#L445

    return NewDaprRuntime(runtimeConfig, globalConfig, accessControlList, resiliencyProvider), nil
}

daprd then starts the DaprRuntime with all the available building blocks:
https://github.com/dapr/dapr/blob/ddd11bcc07ddf61bf5edd835a4b621a3ef1d395a/cmd/daprd/main.go#L6

err = rt.Run(
    runtime.WithSecretStores(secretstoresLoader.DefaultRegistry),
    runtime.WithStates(stateLoader.DefaultRegistry),
    runtime.WithConfigurations(configurationLoader.DefaultRegistry),
    runtime.WithLocks(lockLoader.DefaultRegistry),
    runtime.WithPubSubs(pubsubLoader.DefaultRegistry),
    runtime.WithNameResolutions(nrLoader.DefaultRegistry),
    runtime.WithBindings(bindingsLoader.DefaultRegistry),
    runtime.WithCryptoProviders(cryptoLoader.DefaultRegistry),
    runtime.WithHTTPMiddlewares(httpMiddlewareLoader.DefaultRegistry),
    runtime.WithWorkflowComponents(workflowsLoader.DefaultRegistry),
)

DaprRuntime starts a gRPC server and an HTTP server which the user application communicates with.

The endpoints are implemented here:
- gRPC:
  https://github.com/dapr/dapr/blob/89a93c9516da56e03b74316ecfcf95ae4c23f488/pkg/grpc/api.go
- HTTP:
  https://github.com/dapr/dapr/blob/5aba3c9aa4ea9b3f388df125f9c66495b43c5c9e/pkg/http/api.go

Threat actors

A threat actor is an individual or group that intentionally attempts to exploit vulnerabilities, deploy malicious code, or compromise or disrupt a Dapr deployment, often for financial gain, espionage, or sabotage.
We identify the following threat actors below; For example, a fully untrusted user can also be a contributor to a 3rd-party library used by Dapr. actors for Dapr. A threat actor can assume multiple profiles from the tab

<table>
<thead>
<tr>
<th>Actor</th>
<th>Description</th>
<th>Level of trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>External attacker</td>
<td>Users that have not been granted any privileges and are unauthenticated.</td>
<td>Fully untrusted</td>
</tr>
<tr>
<td>Cluster operator</td>
<td>A user that has permissions to manage the Kubernetes cluster for deployments of Dapr in Kubernetes mode.</td>
<td>Fully trusted</td>
</tr>
<tr>
<td>Application user</td>
<td>A user of the application that the Dapr sidecar has been deployed alongside.</td>
<td>Fully untrusted</td>
</tr>
<tr>
<td>Contributors to 3rd-party dependencies</td>
<td>Contributors to dependencies used by Dapr.</td>
<td>Fully untrusted</td>
</tr>
<tr>
<td>Well-funded criminal groups</td>
<td>Organized criminal groups that often have either political or economic goals. These groups typically have large resources available and specific goals to achieve.</td>
<td>Fully untrusted</td>
</tr>
<tr>
<td>Cloud service users</td>
<td>Users managing the cloud services that a Dapr deployment interacts with.</td>
<td>Partially untrusted</td>
</tr>
</tbody>
</table>

**Trustflow**

Trustflow describes how trust flows through a software system. We identity four trust zones in Daprs threat model:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Internet</td>
<td>Dapr users will often expose their application to the internet. For later reference, we call users in the Internet trust zone “Application Users” as they are meant to interact with the application.</td>
</tr>
<tr>
<td>2</td>
<td>User application</td>
<td>The application that users are deploying Dapr with.</td>
</tr>
</tbody>
</table>
Below we illustrate the trustflow across Dapr’s trust zone and draw the trust boundaries between each zone.

This diagram presents the trustflow between applications users, the application, the Dapr sidecar and the remote services. There are three trust boundaries in this workflow:

1: Trust boundary between untrusted users and the application
The first trust boundary sits between the internet and the application. Here, trust flows from low to high in the direction from the internet to the application. This trust boundary exemplifies use cases where the application accepts untrusted data from the internet.

2: Trust boundary between the application and the Dapr sidecar
When the user application interacts with the Dapr sidecar, trust flows from low to high to low in the direction from the application to the Dapr sidecar, and high to low in the opposite direction. Dapr should guarantee its security posture from malicious requests from the user application through supported Dapr endpoints: An application sending malicious requests to Dapr should not be able to compromise Daprs security posture; however, Dapr should not guarantee that the application’s security remains uncompromised in the case of malicious requests from Dapr to the application - this is the responsibility of the application to ensure. For example, if Dapr sends a request to a NodeJS application that triggers a remote code execution vulnerability in the NodeJS
application³, this is entirely the responsibility of the application; Dapr should not be prevented from sending such requests to the application. The same principle applies in the opposite direction: The application should be able to send any request to the Dapr sidecar without compromising Dapr; It is Dapr’s responsibility to take adequate measures to harden itself against any potentially harmful request.

The trust boundary between the application and the Dapr sidecar exists for that reason: Dapr cannot guarantee that the application takes adequate measures to sanitize, normalize and check all incoming requests before the user application passes these onto Dapr. As such, from the perspective of Dapr, requests from the application are untrusted and Dapr should ensure that the sidecar is not compromised from them. Importantly, Dapr should ensure its own security posture, but it does not need to harden against compromises later in the dataflow; For example, an untrusted user may send a request to the user application that compromises neither the user application nor the Dapr sidecar nor a particular Dapr component but does trigger a vulnerability in a remote service. The request could also trigger a vulnerability that returns sensitive information from the remote service all the way back through the Dapr sidecar, through the user application to the untrusted user. This is not Dapr’s responsibility to defend against.

3: Trust boundary between the Dapr sidecar and remote services
When Dapr components interact with remote cloud services, traffic crosses the cluster’s trust boundary. When data leaves the cluster, trust flows from high to low in the direction from the cluster to the cloud service. This trust boundary has a few security implications for Dapr: First, cloud providers offering these remote services can be compromised, and an attacker can obtain control over the data being sent to Dapr. An attacker who controls the data sent to Dapr can most likely do severe damage to other customers of the remote service besides Dapr users. As such, the attacker could seek much more rewarding outcomes than sending malicious data to a Dapr deployment. However, some criminal groups could consider compromising a particular, targeted Dapr deployment the highest reward for their objectives; for example, groups that specifically target political dissidents would find it more rewarding to compromise a single high-profile political dissident than 100 regular internet users. Google has been the victim of persistent attacks from state-backed groups that attempted to breach Google’s infrastructure to access private emails of human activists⁴. As such, we consider this an attack vector that well-funded threat actors are willing to attempt. Breaching the cloud providers’ infrastructure is not the only route to control the data that Dapr users receive from these remote services; Threat actors can instead breach the accounts managing these remote services. A user may have user accounts registered with limited permissions that are sufficient to cause harm to Dapr

³ For example CVE-2022-24760
⁴ EP000: Operation Aurora | HACKING GOOGLE; https://www.youtube.com/watch?v=przDcQe6n5o
- for example write permissions to an S3 bucket. A threat actor could leverage this by writing a legitimate and valid object to the bucket that would cause Dapr to run slowly or exhaust its resources because of a vulnerability in the Dapr sidecar.

The main reason for this trust boundary is that Dapr cannot control what happens in the remote services and, therefore, cannot trust that they return data that never compromises the core security posture of Dapr. Above we highlighted how targeted attacks could compromise a Dapr deployment, but even unexpected changes in remote services can cause reliability issues in Dapr. A scenario could be that a remote storage service suddenly increases the allowed size of returned data blobs, and Dapr neither enforces time outs on long connections or processes the incoming blobs in such a manner that Dapr would exhaust its resources. In that case, a malicious threat actor could add large objects to a storage bucket and wait until Dapr would fetch it and cause denial of service of Dapr.

Components Contrib

The Components Contrib repository is a collection of community-maintained components that are either shipped with Dapr per default or can be plugged in.

This design is prone to a large attack surface from the 3rd-party dependency contributor threat actor. As such, the Components Contrib subproject should enforce measures to limit the attack surface. In essence, any contributor can make a contribution to a component and leave the project afterwards. If the contributor introduces bugs or security vulnerabilities, they are not required to follow up and fix these. This is not only relevant for Components Contrib’s direct dependencies, but also for its transitive dependencies.

An attacker can commit malicious PRs to a library in Component-contribs dependency tree or perform a dependency confusion attack - which is a manoeuvre where an attacker takes over a library to harm a user of the library.

Another important part of the trust flow to and from Components Contrib is the question of sanitization of user input. If the application does not properly sanitize user input, the Dapr user exposes themselves to a wide range of vulnerabilities. An example from our manual code review are SQL Injections: All components that receive SQL queries from the application and pass them to the database service are vulnerable to SQL injections if the user does not properly sanitize these. In most cases, the full SQL query comes from the request which gives the attacker full control over the query⁵.

⁵ We have tracked this issue under “Issues found” with ID ADA-DAPR-23-3.
Because of this attack surface from untrusted input, it is important that Dapr clearly communicates the security boundaries of the Components Contrib sub-project. We found the documentation to not communicate that sufficiently, and we found that several components were ambiguous about the trust from the data coming from the user application. For example, we found two components that do not trust the incoming request. The first component is the localstorage binding which is hardened against arbitrary file writes:


```go
func (ls *LocalStorage) create(filename string, req *bindings.InvokeRequest) (*bindings.InvokeResponse, error) {
    d, err := strconv.Unquote(string(req.Data))
    if err == nil {
        req.Data = []byte(d)
    }

    decoded, err := base64.StdEncoding.DecodeString(string(req.Data))
    if err == nil {
        req.Data = decoded
    }

    absPath, relPath, err := getSecureAbsRelPath(ls.metadata.RootPath, filename)
    if err != nil {
        return nil, fmt.Errorf("error getting absolute path for file %s: %w", filename, err)
    }

    dir := filepath.Dir(absPath)
    err = os.MkdirAll(dir, 0777)
    if err != nil {
        return nil, fmt.Errorf("error creating directory %s: %w", dir, err)
    }

    f, err := os.Create(absPath)
    if err != nil {
        return nil, fmt.Errorf("error creating file %s: %w", absPath, err)
    }
}
```

getSecureAbsRelPath() defends against path traversal attacks:


```go
func getSecureAbsRelPath(rootPath string, filename string) (absPath string, relPath string, err error) {
    absPath, err = securejoin.SecureJoin(rootPath, filename)
    if err != nil {
        return
    }
    relPath, err = filepath.Rel(rootPath, absPath)
```

ADAlogics
The check for path traversal attacks is unnecessary if the input is trusted.

The second example we found was another defense against path traversal in the HTTP binding:

https://github.com/dapr/components-contrib/blob/e46130ad74ebd9871cfe0ad7914d3a168a914cc7/bindings/http/http.go#L229-L241

```go
func (h *HTTPSource) Invoke(parentCtx context.Context, req *bindings.InvokeRequest) (*bindings.InvokeResponse, error) {
    u := h.metadata.URL
    errorIfNot2XX := h.errorIfNot2XX // Default to the component config (default is true)

    if req.Metadata != nil {
        if path, ok := req.Metadata["path”]; ok {
            // Simplicity and no "../" type exploits.
            u = fmt.Sprintf("%s/%s", strings.TrimRight(u, "/"),
            strings.TrimLeft(path, "/"))
            if strings.Contains(u, "../") {
                return nil, fmt.Errorf("invalid path: %s", path)
            }
        }
    }
}
```

This check is also unnecessary in case the `InvokeRequest` is trusted.
Fuzzing

During the audit, Ada Logics wrote five new fuzzers for Dapr. We added the fuzzers to Daprs OSS-Fuzz integration so that they run continuously after the audit concluded. This allows the fuzzers to run for a longer time and explore more of the reachable code. It also allows the fuzzers to keep testing the latest master branch as it evolves to test whether new bugs get introduced. Short-term, OSS-Fuzz was of value, in that one of the fuzzers found a security vulnerability in a 3rd-party dependency to Components Contrib (ADA-DAPR-23-7).

We added all fuzzers to Daprs integration at https://github.com/cncf/cncf-fuzzing/tree/main/projects/dapr. When OSS-Fuzz builds Daprs fuzzers, it pulls them from there and compiles them against the latest main/master branch.

The fuzzers added during this audit are:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Target code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FuzzRLTest</td>
<td>github.com/dapr/components-contrib/middleware/http/ratelimit</td>
</tr>
<tr>
<td>2</td>
<td>FuzzAzureEventGridTest</td>
<td>github.com/dapr/components-contrib/bindings/azure/eventgrid</td>
</tr>
<tr>
<td>3</td>
<td>FuzzGraphqlRETest</td>
<td>github.com/dapr/components-contrib/bindings/graphql</td>
</tr>
<tr>
<td>4</td>
<td>FuzzAvroTest</td>
<td>github.com/dapr/components-contrib/pubsub/pulsar</td>
</tr>
<tr>
<td>5</td>
<td>FuzzPurellTest</td>
<td>github.com/dapr/pkg/acl</td>
</tr>
</tbody>
</table>

**FuzzRLTest**
Tests whether well-crafted requests to the `ratelimit` middleware component can cause harm.

URL: https://github.com/cncf/cncf-fuzzing/blob/7ed5200c931ff9277d0cd7f587d8792295cd597d/projects/dapr/fuzz_components_contrib_ratelimiter_test.go

**FuzzAzureEventGridTest**
Tests the validation routine for the authorization header of incoming requests.
FuzzGraphqlRETest
Tests whether the regular expression that processes incoming requests to the GraphQL component is safe.

URL: https://github.com/cncf/cncf-fuzzing/blob/d9711dcf18a17cb8671b0b80023eabf2b557a9f5/projects/dapr/fuzz_components_contrib_graphql_test.go

FuzzAvroTest
Tests the parsing routine for incoming requests to the Pulsar pubsub component.

URL: https://github.com/cncf/cncf-fuzzing/blob/ec91e3af6a8de485e207bfabc91442a39c23b01/projects/dapr/fuzz_components_contrib_pubsub_pulsar_test.go

FuzzPurellTest
Tests a string processing routine that Dapr’s acl package uses to normalize the URLs of incoming requests.

URL: https://github.com/cncf/cncf-fuzzing/blob/ec91e3af6a8de485e207bfabc91442a39c23b01/projects/dapr/fuzz_acl_test.go#L43
Issues found

In this section we present the findings from goal #2 of the security audit, “Perform a manual code audit of the code assets in scope.” We found 7 security issues during this goal, one of which was a security vulnerability in a 3rd-party library which was assigned CVE-2023-37475. Issue 1, 2, 3, and 4 are umbrella issues of a specific class of vulnerabilities that affect several components in the same building block in a similar manner.

<table>
<thead>
<tr>
<th>#</th>
<th>ID</th>
<th>Title</th>
<th>Severity</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADA-DAPR-23-1</td>
<td>Lack of warning when skipping certificate verification</td>
<td>Informational</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>ADA-DAPR-23-2</td>
<td>DoS from large responses from cloud services</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>ADA-DAPR-23-3</td>
<td>SQL strings are not sanitized to defend against injections</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>ADA-DAPR-23-4</td>
<td>Denial of service of Dapr from bypassing limit of response size</td>
<td>Low</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>ADA-DAPR-23-5</td>
<td>Archived and deprecated 3rd-party dependencies</td>
<td>Low</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>ADA-DAPR-23-6</td>
<td>Possible DoS in HTTP binding</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>ADA-DAPR-23-7</td>
<td>OOM triggerable by malicious PubSub message</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
</tbody>
</table>

6 ADA-DAPR-23-7
Lack of warning when skipping certificate verification

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Components Contrib</td>
</tr>
<tr>
<td>Severity</td>
<td>Informational</td>
</tr>
<tr>
<td>Fixed in:</td>
<td><a href="https://github.com/dapr/components-contrib/pull/3090/files">Link</a></td>
</tr>
</tbody>
</table>

Some components allow the user to skip TLS verification which per default is disabled which is positive for Daprs security posture. Most modules also both name the option “insecure” and log a warning if the user has opted in for this setting which is best-practice for this type of setting. For example the kafka component labels the option insecure:

```go
tlsConfig := &tls.Config{
    MinVersion:   tls.VersionTLS12,
    InsecureSkipVerify: ts.skipCaVerify,
}
```

The Kafka component also logs a warning:

```go
if m.TLSSkipVerify {
    k.logger.Infos("kafka: you are using 'skipVerify' to skip server config verify which is unsafe!")
}
```

Not all components follow this practice. The Hashicorp Vault Secretstore component labels the option “Insecure” but does not log a warning. Other components do not log if certification verification is skipped.

We recommend following the standards for all components that allow users to skip certificate verification:

1. Always name the option “Insecure”
2. Always log a warning if the option is used.
3. Always disable certificate verification by default.
DoS from large responses from cloud services

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Components Contrib</td>
</tr>
<tr>
<td>Severity</td>
<td>Moderate</td>
</tr>
<tr>
<td>Fixed in:</td>
<td><a href="https://github.com/dapr/docs/pull/3684">https://github.com/dapr/docs/pull/3684</a></td>
</tr>
</tbody>
</table>

A common pattern of Daprs components is that it reads the response from cloud services entirely into memory without setting a limit to the size response from the cloud services. A large response has the potential to drain memory and cause denial of service of Dapr. This issue is an umbrella issue for all cases the Dapr components perform this operation.

This will not result in a system-wide resource exhaustion, but an attacker will be able to use enough memory for Go to perform a sigkill and crash Dapr causing a denial-of-service. This may result in a denial of service for other Go processes on the machine.

The issue can be triggered accidentally or purposefully; If a user simply misconfigures their Dapr deployment, they might at some point request a large object from a cloud service that exhausts memory of the Dapr sidecar. The attack vector of this umbrella issue is that a lower-privileged user can purposefully add a large object to a cloud store that will exhaust memory when Dapr requests it. The attacker is likely to be an insider who has certain privileges.

**Example 1: Vault**

If the Vault SecretStore component does not receive a successful response from the remote store, Dapr copies the response into a buffer and subsequently logs it: [https://github.com/dapr/components-contrib/blob/cfbac4d794b35e5da28d65a13369d33383fb6ad4/secretstores/hashicorp/vault/vault.go#L247](https://github.com/dapr/components-contrib/blob/cfbac4d794b35e5da28d65a13369d33383fb6ad4/secretstores/hashicorp/vault/vault.go#L247)
If the remote Secretstore returns a large response, these two lines will exhaust memory and crash the Dapr instance resulting in a denial-of-service of Dapr.

**Example 2: Vault**

If Dapr receives a http.StatusOK from the remote Secretstore and the vaultValueType is not a map type, then Dapr will read the entire response into memory without an upper bounds to the size of the body of the response:

https://github.com/dapr/components-contrib/blob/cfbac4d794b35e5da28d65a13369d33383fb6ad4/secretstores/hashicorp/vault/vault.go#L267

```go
if v.vaultValueType.isMapType() {
    // parse the secret value to map[string]string
    if err := json.NewDecoder(httpresp.Body).Decode(&d); err != nil {
        return nil, fmt.Errorf("couldn't decode response body: %s", err)
    }
} else {
    // treat the secret as string
    b, err := io.ReadAll(httpresp.Body)
    if err != nil {
        return nil, fmt.Errorf("couldn't read response: %s", err)
    }
    res := v.json.Get(b, DataStr, DataStr).ToString()
    d.Data.Data = map[string]string{
        secret: res,
    }
}
```

**Example 3: Wasm**

While not a cloud service, Daprs Wasm component’s request handler can cause excessive resource exhaustion on the host machine which can lead to DoS of Dapr. If a request causes the Wasm handler to generate a large stdout or stderr buffer, Dapr will it entirely into memory and exhaust memory of the machine:
**Recommendation**

Check the size of the `stdout` and `stderr` buffers before reading them.

**Example 4: Oracle Cloud Infrastructure**

Daprs OCI (Oracle Cloud Infrastructure) client fetches an object from the OCI storage in its `getObject()` method:

```go
func (c *ociObjectStorageClient) getObject(ctx context.Context, objectname string) (content []byte, etag *string, metadata map[string]string, err error) {
    c.logger.Debugf("read file %s from OCI ObjectStorage StateStore %s ",
    objectname, &c.objectStorageMetadata.BucketName)
    request := objectstorage.GetObjectRequest{
        NamespaceName: &c.objectStorageMetadata.Namespace,
        BucketName: &c.objectStorageMetadata.BucketName,
        ObjectName: &objectname,
    }
    response, err := c.objectStorageMetadata.OCIObjectStorageClient.GetObject(ctx, request)
    if err != nil {
        c.logger.Debugf("Issue in OCI ObjectStorage with retrieving object %s, error: %s", objectname, err)
            return nil, nil, nil, nil
        }
        return nil, nil, nil, fmt.Errorf("failed to retrieve object : %w", err)
    }
    buf := new(bytes.Buffer)
    buf.ReadFrom(response.Content)
    return buf.Bytes(), response.ETag, response.OpcMeta, nil
}
```
This method reads the contents of the retrieved object entirely into memory on its last line:

```go
buf := new(bytes.Buffer)
buf.ReadFrom(response.Content)
return buf.Bytes(), response.ETag, response.OpcMeta, nil
```

This could create a resource exhaustion scenario given that the limit of an objects size is 10TiB in Oracle Cloud Infrastructure:

**Limits on Object Storage Resources**

See [Service Limits](https://docs.oracle.com/en-us/iaas/Content/Object/Concepts/objectstorageoverview.htm) for a list of applicable limits and instructions for requesting a limit increase.

To set tenancy or compartment-specific storage limits, administrators can use [object storage quotas](https://docs.oracle.com/en-us/iaas/Content/Object/Concepts/objectstorageoverview.htm).

Other limits include:

- Number of Object Storage namespaces per root compartment: 1
- Maximum object size: 10 TiB
- Maximum object part size in a multipart upload: 50 GiB
- Maximum number of parts in a multipart upload: 10,000
- Maximum object size allowed by PutObject API: 50 GiB
- The total size of all the metadata assigned to an object is limited to 4000 bytes.

We found the same issue in the following components:

- Alibaba Cloud DingTalk binding (Alpha component*)
- Apple Push Notification Service binding
- Azure Blob Storage state store
- Azure Signalr binding (Alpha component*)
- GCP bucket binding (Alpha component*)
- Hashicorp Vault binding
- Huawei OBS binding (Alpha component*)
- OCI binding (Beta component*)
- Twilio SMS binding (Alpha component*)

* Not recommended to be used in production.
**SQL strings are not sanitized to defend against injections**

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Components Contrib</td>
</tr>
<tr>
<td>Severity</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

**Fixed in:**
- [https://github.com/dapr/components-contrib/pull/2975](https://github.com/dapr/components-contrib/pull/2975)
- [https://github.com/dapr/components-contrib/pull/2972](https://github.com/dapr/components-contrib/pull/2972)

Daprs bindings dealing with SQL in Components Contrib do not sanitize the queries before executing them which could lead to sql injection attacks in case the user passes untrusted input from the application to Dapr. In fact, if an attacker can get a malicious SQL query to the MySQL binding, [https://github.com/dapr/components-contrib/blob/cfbac4d794b35e5da28d65a13369d33383fb6ad4/bindings/mysql/mysql.go#L136](https://github.com/dapr/components-contrib/blob/cfbac4d794b35e5da28d65a13369d33383fb6ad4/bindings/mysql/mysql.go#L136), they have essentially succeeded in executing an SQL injection, since the SQL string is not sanitized:

```go
func (m *Mysql) Invoke(ctx context.Context, req *bindings.InvokeRequest) (*bindings.InvokeResponse, error) {
    if req == nil {
        return nil, errors.New("invoke request required")
    }

    if req.Operation == closeOperation {
        return nil, m.db.Close()
    }

    if req.Metadata == nil {
        return nil, errors.New("metadata required")
    }

    m.logger.Debugf("operation: %v", req.Operation)

    s, ok := req.Metadata[commandSQLKey]
    if !ok || s == "" {
        return nil, fmt.Errorf("required metadata not set: %s",
            commandSQLKey)
    }

    startTime := time.Now()
```
```go
dapr

```dapr

```

Recommendation

We recommend one of the following:

- Properly sanitize SQL strings before passing them onto the service.
- Document that users should never pass unsanitized, untrusted SQL to bindings.
Denial of service of Dapr from bypassing limit of response size in AppChannel

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Local and external HTTP channels</td>
</tr>
<tr>
<td>Severity</td>
<td>Low</td>
</tr>
<tr>
<td>Fixed</td>
<td>No</td>
</tr>
</tbody>
</table>

Dapr has two cases of possible bypasses of a size checks of HTTP responses from untrusted sources:

1. Daprs external AppChannel
2. Daprs local AppChannel

The vulnerable methods limit the size of a response from a user application, however, an attacker can trigger an OOM panic before Dapr performs the size check.

**External AppChannel**

The issue is triggerable from Dapr Runtimes (`*HTTPEndpointAppChannel.InvokeMethod`). Below we demonstrate how the data flows from `*HTTPEndpointAppChannel.InvokeMethod` to the point of failure:

https://github.com/dapr/dapr/blob/20d56527e94d451e014be27a0b778e87b0b9556a/pkg/channel/external/http_channel.go#L90

```go
func (h *HTTPEndpointAppChannel) InvokeMethod(ctx context.Context, req *invokev1.InvokeMethodRequest, appID string) (rsp *invokev1.InvokeMethodResponse, err error)
```

invokes

https://github.com/dapr/dapr/blob/20d56527e94d451e014be27a0b778e87b0b9556a/pkg/channel/external/http_channel.go#L104

```go
func (h *HTTPEndpointAppChannel) invokeMethodV1(ctx context.Context, req *invokev1.InvokeMethodRequest, appID string) (*invokev1.InvokeMethodResponse, error)
```

Which sends the request to the user application and parses the response. When `invokeMethodV1()` parses the response, it limits the size of the response. In the below
code example, the lines highlighted with blue send the request to the user application, and the line highlighted with green invokes the parsing API of the response:

https://github.com/dapr/dapr/blob/20d56527e94d451e014be27a0b778e87b0b9556a/pkg/channel/external/http_channel.go#L104

```go
func (h *HTTPEndpointAppChannel) invokeMethodV1(ctx context.Context, req *invokev1.InvokeMethodRequest, appID string) (*invokev1.InvokeMethodResponse, error) {
    channelReq, err := h.constructRequest(ctx, req, appID)
    if err != nil {
        return nil, err
    }

    if h.ch != nil {
        h.ch <- struct{}{}
    }
    defer func() {
        if h.ch != nil {
            <-h.ch
        }
    }()

    // Emit metric when request is sent
    startRequest := time.Now()

    var resp *http.Response
    if len(h.pipeline.Handlers) > 0 {
        // Exec pipeline only if at least one handler is specified
        rw := &httpChannel.RWRecorder{
            W: &bytes.Buffer{},
        }
        execPipeline := h.pipeline.Apply(http.HandlerFunc(func(wr http.ResponseWriter, r *http.Request) {
            // Send request to user application
            // (Body is closed below, but linter isn't detecting that)
            //nolint:bodyclose
            clientResp, clientErr := h.client.Do(r)
            if clientResp != nil {
                copyHeader(wr.Header(), clientResp.Header)
                wr.WriteHeader(clientResp.StatusCode)
                _, _ = io.Copy(wr, clientResp.Body)
            }
            if clientErr != nil {
                err = clientErr
            }
        }))
        execPipeline.ServeHTTP(rw, channelReq)
        resp = rw.Result() //nolint:bodyclose
    } else {
        // Send request to user application
        // (Body is closed below, but linter isn't detecting that)
        //nolint:bodyclose
        resp, err = h.client.Do(channelReq)
    }

```
elapsedMs := float64(time.Since(startRequest) / time.Millisecond)

var contentLength int64
if resp != nil {
    if resp.Header != nil {
        contentLength, _ = strconv.ParseInt(resp.Header.Get("content-length"), 10, 64)
    }
}

if err != nil {
    return nil, err
}

rsp, err := h.parseChannelResponse(req, resp)
if err != nil {
    return nil, err
}

return rsp, nil

(h *HTTPEndpointAppChannel).parseChannelResponse checks the size of the http response before converting it to an invoke method response:

https://github.com/dapr/dapr/blob/20d56527e94d451e014be27a0b778e87b0b9556a/pkg/channel/external/http_channel.go#L245-L264

func (h *HTTPEndpointAppChannel) parseChannelResponse(req *invokev1.InvokeMethodRequest, channelResp *http.Response) (*invokev1.InvokeMethodResponse, error) {
    contentType := channelResp.Header.Get("content-type")

    // Limit response body if needed
    var body io.ReadCloser
    if h.maxResponseBodySizeMB > 0 {
        body = streamutils.LimitReadCloser(channelResp.Body, int64(h.maxResponseBodySizeMB)<<20)
    } else {
        body = channelResp.Body
    }

    // Convert status code
    rsp := invokev1.
Dapr security audit 2023

WithHTTPHeaders(channelResp.Header).
WithRawData(body).
WithContentType(contentType)

return rsp, nil
}

The vulnerability
When Dapr copies the response from the response into the response writer in execPipeline(), a large body will crash Dapr with an out-of-memory panic. The issue are on the highlighted line:

https://github.com/dapr/dapr/blob/20d56527e94d451e014be27a0b778e87b0b9556a/pkg/channel/external/http_channel.go#L133-L138

clientResp, clientErr := h.client.Do(r)
if clientResp != nil {
    copyHeader(wr.Header(), clientResp.Header)
    wr.WriteHeader(clientResp.StatusCode)
    _, _ = io.Copy(wr, clientResp.Body)
}

PoC
The following PoC demonstrates the issue. To reproduce, run the following PoC with go run main.go. We include the expected stacktrace below.

package main

import {
    "bytes"
    "fmt"
    "io"
    "net/http"
    "strings"
}

type client struct{}

func (c *client) Do(req *http.Request) *http.Response {
    longString := strings.Repeat("Abc", 1000000000)
    r1 := strings.NewReader(longString)
    r2 := strings.NewReader(longString)
    r3 := strings.NewReader(longString)
    body := io.NopCloser(io.MultiReader(r1, r2, r3))
    resp := &http.Response{Body: body}
    return resp
}

func main() {
    c := &client{

bytes.NewReader([]byte("request body")))
    if err != nil {
        panic(err)
    }
    resp := c.Do(req)
    fmt.Println("Copying...")
    if _, err := io.Copy(io.Discard, resp.Body); err != nil {
    }

PoC - expected stacktrace

fatal error: runtime: out of memory

runtime stack:
runtime.throw({0x55962e?, 0x0?})
/usr/local/go/src/runtime/panic.go:1047 +0x5d fp=0x7fffb29cd648 pc=0x434a7d
runtime.sysMapOS(0xc000400000, 0x6fc400000?)
/usr/local/go/src/runtime/mem_linux.go:187 +0x11b fp=0x7fffb29cd690 pc=0x417f7b
runtime.sysMap(0x69dde0?, 0x3fffffff?, 0x6adf78?)
/usr/local/go/src/runtime/mem.go:142 +0x35 fp=0x7fffb29cd6c0 sp=0x7fffb29cd690 pc=0x417f95
runtime.(*mheap).grow(0x69dde0, 0x37e11e?)
/usr/local/go/src/runtime/mheap.go:1522 +0x245 fp=0x7fffb29cd738 pc=0x427ea5
runtime.(*mheap).allocSpan(0x69dde0, 0x37e11e, 0x0, 0xe9?)
/usr/local/go/src/runtime/mheap.go:1243 +0x1b7 fp=0x7fffb29cd7d0 sp=0x7fffb29cd738 pc=0x4275f7
runtime.(*mheap).alloc.func1() (/usr/local/go/src/runtime/mheap.go:961 +0x65 fp=0x7fffb29cd818 sp=0x7fffb29cd7d0 pc=0x4270a5
runtime.systemstack() (/usr/local/go/src/runtime/asm_amd64.s:496 +0x49 fp=0x7fffb29cd820 sp=0x7fffb29cd818 pc=0x462329
goroutine 1 [running]:
runtime.systemstack_switch() (/usr/local/go/src/runtime/asm_amd64.s:463 +0x45 fp=0x7fffb29cd818 sp=0x7fffb29cd820 pc=0x426fe5
runtime.(*mcache).allocLarge(0x3f?, 0x6fc23ac00, 0x1)
/usr/local/go/src/runtime/mcache.go:234 +0x85 fp=0x7fffb29cd927 sp=0x7fffb29cd918 pc=0x44b972
strings.(*Builder).grow(...) (/usr/local/go/src/strings/builder.go:68
strings.(*Builder).Grow(...) (/usr/local/go/src/strings/builder.go:68
strings.Repeat(0x554ee4?, 0xc000007860?)
/usr/local/go/src/strings/strings.go:569 +0x11a fp=0xc00011be30 sp=0xc00011bd88 pc=0x44a63a
main.(*client).Do(0x59fa68?, 0xc00001a00?)
/tmp/gopoc/main.go:14 +0x45 fp=0xc00011bef8 sp=0xc00011be30 pc=0x518b05
Local AppChannel

Daprs local AppChannel has a response size check that can be bypassed. The vulnerable method limits the size of a response from a user application, however, an attacker can trigger an OOM panic before Dapr performs the size check.

Dataflow

https://github.com/dapr/dapr/blob/7a0fdf9f016b0c3a3f48447bc4169838e6026ec2/pkg/channel/http/http_channel.go#L136

```go
func (h *Channel) InvokeMethod(ctx context.Context, req *invokev1.InvokeMethodRequest) (rsp *invokev1.InvokeMethodResponse, err error)
```

invokes

https://github.com/dapr/dapr/blob/7a0fdf9f016b0c3a3f48447bc4169838e6026ec2/pkg/channel/http/http_channel.go#L205

```go
func (h *Channel) invokeMethodV1(ctx context.Context, req *invokev1.InvokeMethodRequest) (*invokev1.InvokeMethodResponse, error)
```

Which sends the request to the user application and parses the response. When `invokeMethodV1()` parses the response, it limits the size of the response. Below the lines highlighted with blue send the request to the user application, and the line highlighted with green invokes the parsing API of the response:

https://github.com/dapr/dapr/blob/7a0fdf9f016b0c3a3f48447bc4169838e6026ec2/pkg/channel/http/http_channel.go#L205

```go
func (h *Channel) invokeMethodV1(ctx context.Context, req *invokev1.InvokeMethodRequest) (*invokev1.InvokeMethodResponse, error) {
    channelReq, err := h.constructRequest(ctx, req)
    if err != nil {
        return nil, err
    }

    if h.ch != nil {
        // Invokes parsing API of the response
    }
```
h.ch <- struct{}{}
}

defer func() {
    if h.ch != nil {
        <-h.ch
    }
}()

// Emit metric when request is sent
startRequest := time.Now()

var resp *http.Response
if len(h.pipeline.Handlers) > 0 {
    // Exec pipeline only if at least one handler is specified
    rw := &RWRecorder{
        W: &bytes.Buffer{},
    }
    execPipeline = h.pipeline.Apply(http.HandlerFunc(func(wr http.ResponseWriter, r *http.Request) {
        // Send request to user application
        // (Body is closed below, but linter isn't detecting that)
        //nolint:bodyclose
        clientResp, clientErr := h.client.Do(r)
        if clientResp != nil {
            copyHeader(wr.Header(), clientResp.Header)
            wr.WriteHeader(clientResp.StatusCode)
            _, _ = io.Copy(wr, clientResp.Body)
        }
        if clientErr != nil {
            err = clientErr
        }
    }))
    execPipeline.ServeHTTP(rw, channelReq)
    resp = rw.Result() //nolint:bodyclose
} else {
    // Send request to user application
    // (Body is closed below, but linter isn't detecting that)
    //nolint:bodyclose
    resp, err = h.client.Do(channelReq)
}

elapsedMs := float64(time.Since(startRequest) / time.Millisecond)

var contentLength int64
if resp != nil {
    if resp.Header != nil {
        contentLengthString := resp.Header.Get("content-length")
        contentLength, _ = strconv.ParseInt(contentLengthString, 10, 64)
    }
}

if err != nil {
parseChannelResponse() checks the size of the http response before converting it to an invoke method response:

https://github.com/dapr/dapr/blob/7a0fdf9f016b0c3a3f48447bc4169838e6026ec2/pkg/channel/http_channel.go#L327

```go
func (h *Channel) parseChannelResponse(req *invokev1.InvokeMethodRequest, channelResp *http.Response) (*invokev1.InvokeMethodResponse, error) {
    contentType := channelResp.Header.Get("content-type")

    // Limit response body if needed
    var body io.ReadCloser
    if h.maxResponseBodySizeMB > 0 {
        body = streamutils.LimitReadCloser(channelResp.Body, int64(h.maxResponseBodySizeMB)<<20)
    } else {
        body = channelResp.Body
    }

    // Convert status code
    rsp := invokev1.NewInvokeMethodResponse(int32(channelResp.StatusCode), "", nil).WithHTTPHeaders(channelResp.Header).WithRawData(body).WithContentType(contentType)

    return rsp, nil
}
```

**The vulnerability**

When Dapr copies the response from the response into the response writer in `execPipeline()`, a large body will crash Dapr with an out-of-memory panic. The issue are on the highlighted line:
https://github.com/dapr/dapr/blob/7a0fdf9f016b0c3a3f48447bc4169838e6026ec2/pkg/channel/ht
tp/http_channel.go#L234-L239

```go
clientResp, clientErr := h.client.Do(r)
if clientResp != nil {
    copyHeader(wr.Header(), clientResp.Header)
    wr.WriteHeader(clientResp.StatusCode)
    _, _ = io.Copy(wr, clientResp.Body)
}
```
Archived and deprecated 3rd-party dependencies

<table>
<thead>
<tr>
<th>ID</th>
<th>Component</th>
<th>Severity</th>
<th>Fixed</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADA-DAPR-23-5</td>
<td>Components Contrib</td>
<td>Low</td>
<td>Yes</td>
<td>ADA-DAPR-23-5</td>
</tr>
</tbody>
</table>

Dapr core and Components Contrib has 5 archived or deprecated libraries in its direct dependency tree. They are:

<table>
<thead>
<tr>
<th>#</th>
<th>Issue</th>
<th>Dependency</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Archived</td>
<td>github.com/gorilla/mux</td>
<td>Has been un-archived after completion of audit. No change in code.</td>
</tr>
<tr>
<td>2</td>
<td>Archived</td>
<td>github.com/benbjohnson-clock</td>
<td>Removed in <a href="https://github.com/dapr/kit/pull/55">https://github.com/dapr/kit/pull/55</a></td>
</tr>
<tr>
<td>3</td>
<td>Archived</td>
<td>github.com/minio/blake2b-simd</td>
<td>Removed in <a href="https://github.com/dapr/dapr/pull/6763">https://github.com/dapr/dapr/pull/6763</a></td>
</tr>
<tr>
<td>4</td>
<td>Deprecated</td>
<td>github.com/nats-io/stan.go</td>
<td>Planned for removal in Dapr 1.13</td>
</tr>
<tr>
<td>5</td>
<td>Deprecated</td>
<td>github.com/golang/protobuf</td>
<td>Assessed by Dapr team. No change in code.</td>
</tr>
</tbody>
</table>

Archived or deprecated projects are unlikely to fix issues - both reliability issues and security vulnerabilities and that they are unlikely to even accept and triage security disclosures. Furthermore, the project are unlikely to do its own ongoing security work; For example, Ada Logics attempted to involve the project in integrating continuous fuzzing by way of OSS-Fuzz in 2020: [https://github.com/gorilla/mux/pull/575](https://github.com/gorilla/mux/pull/575) via a pull request that has still not been merged⁷.

---

⁷ This PR was closed after the Dapr audit had concluded.
Possible DoS in HTTP binding

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Components Contrib</td>
</tr>
<tr>
<td>Severity</td>
<td>Moderate</td>
</tr>
<tr>
<td>Fixed in:</td>
<td><a href="https://github.com/dapr/components-contrib/pull/3040">https://github.com/dapr/components-contrib/pull/3040</a></td>
</tr>
</tbody>
</table>

Dapr’s HTTP binding (https://github.com/dapr/components-contrib/blob/e46130ad74ebd9871cfe0ad7914d3a168a914cc7/bindings/http/http.go) reads the body of the response from a non-Dapr service entirely into memory. This could be a problem if the non-Dapr service returns a large buffer; Dapr can exhaust memory which will crash the Dapr sidecar and cause a limited denial of service of the host machine.

The HTTP binding reads the response into memory on the following line:

https://github.com/dapr/components-contrib/blob/e46130ad74ebd9871cfe0ad7914d3a168a914cc7/bindings/http/http.go#L320-L331

```go
resp, err := h.client.Do(request)
if err != nil {
    return nil, err
}
defer resp.Body.Close()

// Read the response body. For empty responses (e.g. 204 No Content)
// `b` will be an empty slice.
 b, err := io.ReadAll(resp.Body)
if err != nil {
    return nil, err
}
```

This is a case where a vulnerable remote service can crash the Dapr sidecar. Instead of allowing this, we recommend that Dapr hardens itself against vulnerable remote services.
# OOM triggerable by malicious PubSub message

<table>
<thead>
<tr>
<th>ID</th>
<th>ADA-DAPR-23-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Components Contrib</td>
</tr>
<tr>
<td>Severity</td>
<td>Moderate</td>
</tr>
<tr>
<td>Fixed in</td>
<td><a href="https://github.com/dapr/components-contrib/pull/2994">https://github.com/dapr/components-contrib/pull/2994</a></td>
</tr>
</tbody>
</table>

This issue had its root cause in a 3rd-party dependency which assigned the following CVE:

<table>
<thead>
<tr>
<th>Title</th>
<th>GHSA</th>
<th>CVE</th>
<th>CVSS score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker-controlled parameter can cause DoS of avro</td>
<td>GHSA-9x44-9pgq-cf 45</td>
<td>CVE-2023-374 75</td>
<td>7.5</td>
</tr>
</tbody>
</table>

The Pulsar PubSub component is susceptible to an unrecoverable OOM panic that can be controlled by the data in a `PublishRequest`. This allows a user who can send a pubsub message to the Pulsar component to crash the Dapr sidecar. The following PoC demonstrates the issue. Add the unit test to `components-contrib/pubsub/pulsar/pulsar_test.go` and run it with `go test -run=TestParsePublishMetadata2`.

```go
func TestParsePublishMetadata2(t *testing.T) {
    m := &pubsub.PublishRequest{
        Data: []byte{246, 255, 255, 255, 10, 255, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32}  //, _ = parsePublishMetadata(m, schemaMetadata{protocol: avroProtocol, value: "bytes"})
}
```

The test outputs the following stacktrace:

```
fatal error: runtime: out of memory
runtime stack:
runtime.throw({0xc32c27?, 0x8000?})
    /usr/local/go/src/runtime/panic.go:1047 +0x5d fp=0x7ffe44bca48
sp=0x7ffe44bca5c8 pc=0x445a9d
runtime.sysMapOS(0xc000400000, 0x2c00000000?)
    /usr/local/go/src/runtime/mem_linux.go:187 +0x11b fp=0x7ffe44bc6d30
sp=0x7ffe44bc6ce8 pc=0x424d4f
```
Dapr security audit 2023

runtime.sysMap(0x11aa240?, 0x7fffffff)?, 0x11ba3d8?)
  /usr/local/go/src/runtime/mem.go:142 +0x35 fp=0x7ffe4bbc6d60 sp=0x7ffe4bbc6d30 pc=0x4247d5
runtime.(*mheap).grow(0x11aa240, 0x1600000)
  /usr/local/go/src/runtime/mheap.go:1522 +0x252 fp=0x7ffe4bbc6d8 sp=0x436832
runtime.(*mheap).allocSpan(0x11aa240, 0x1600000, 0x0, 0xae)
  /usr/local/go/src/runtime/mheap.go:1243 +0x1b7 fp=0x7ffe4bbc6e70 sp=0x435f77
runtime.(*mheap).alloc.func1()
  /usr/local/go/src/runtime/mheap.go:961 +0x65 fp=0x7ffe4bbc6eb8 sp=0x435a25
runtime.systemstack()
  /usr/local/go/src/runtime/asm_amd64.s:496 +0x49 fp=0x7ffe4bbc6ec0 sp=0x47a400
goroutine 34 [running]:
runtime.systemstack_switch()
  /usr/local/go/src/runtime/asm_amd64.s:463 +0x85 fp=0xc0003929e0 sp=0xc0003929a48 pc=0x43556f
github.com/hamba/avro/v2.(*Reader).ReadBytes(0x41a68a?)
  /home/adam/go/pkg/mod/github.com/hamba/avro/v2@v2.5.0/reader.go:208 +0x74 fp=0xc000392a48 sp=0xc000392a70 pc=0x43556f
github.com/hamba/avro/v2.(*Reader).ReadNext(0x7f210ffccae0?, 0xc0003929e0)
  /home/adam/go/pkg/mod/github.com/hamba/avro/v2@v2.5.0/reader_generic.go:63 +0x41f fp=0xc000392ac0 sp=0xc000392a48 pc=0x43556f
github.com/hamba/avro/v2.(*efaceDecoder).Decode(0xc0001188f0?, 0xc0002b0060, 0xc0002b0060, 0xc0002b0060, 0xc0002b0060)
  /home/adam/go/pkg/mod/github.com/hamba/avro/v2@v2.5.0/decoder.go:49 fp=0xc000392d88 sp=0xc000392d98 pc=0x83394e
github.com/hamba/avro/v2.Unmarshal(...)
Dapr security audit 2023

```
sp=0xc000392de8 pc=0xa3b8868
github.com/dapr/components-contrib/pubsub/pulsar.TestParsePublishMetadata2(0x413239)
    /tmp/components-contrib/pubsub/pulsar/pulsar_test.go:155 +0xb0 fp=0xc000392f70
sp=0xc000392ef0 pc=0xa3c850
testing.tRunner(0xc0002b8ea0, 0x78960)
    /usr/local/go/src/testing/testing.go:1576 +0x10b fp=0xc000392fc0 sp=0xc000392f70
pc=0x53632b
testing.(*T).Run.func1()
    /usr/local/go/src/testing/testing.go:1629 +0x2a fp=0xc000392fe0 sp=0xc000392fc0
pc=0x47c621
created by testing.(*T).Run
    /usr/local/go/src/testing/testing.go:1629 +0x3ea
goroutine 1 [chan receive]:
    runtime.gopark(0x1192640?, 0xc000294048?, 0x20?, 0xa8?, 0xc00019da28?)
    /usr/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc00019d9a8 sp=0xc00019d988
pc=0x4487f6
runtime.chanrecv(0xc0002e83f0, 0xc00019daa7, 0x1)
    /usr/local/go/src/runtime/chan.go:583 +0x49d fp=0xc00019da38 sp=0xc00019d9a8
pc=0x4137fd
runtime.chanrecv1(0x11916c0?, 0xb445a0?)
    /usr/local/go/src/runtime/chan.go:442 +0x18 fp=0xc00019d9a8 sp=0xc00019d988
pc=0x4487f6
runtime.chanrecv(0xc0002e83f0, 0xc00019daa7, 0x1)
    /usr/local/go/src/runtime/chan.go:583 +0x49d fp=0xc00019da38 sp=0xc00019d9a8
pc=0x4137fd
runtime.chanrecv1(0x11916c0?, 0xb445a0?)
    /usr/local/go/src/runtime/chan.go:442 +0x18 fp=0xc00019d9a8 sp=0xc00019d988
pc=0x4487f6
testing.(*T).Run(0xc000288d00, {0x11477c0?, 0xa?, 0xa?}, 0xc78960)
    /usr/local/go/src/testing/testing.go:1576 +0x10b fp=0xc00019d9a8 sp=0xc00019d988
pc=0x53632b
testing.runTests(0xc0002bcb60?, {0x11477c0?, 0xa, 0xa}, {0x0?, 0x100c0002989a8?, 0x1191d00?})
    /usr/local/go/src/testing/testing.go:2034 +0x489 fp=0xc00019dc8 sp=0xc00019dcb0
pc=0x539289
testing.(*M).Run(0xc0002bcb60)
    /usr/local/go/src/testing/testing.go:1906 +0x63a fp=0xc00019df00 sp=0xc00019dcb0
pc=0x537bfa
main.main()
    _testmain.go:65 +0x1aa fp=0xc00019df00 sp=0xc00019df00 pc=0xa3f08a
runtime.main()
    /usr/local/go/src/runtime/proc.go:250 +0x207 fp=0xc00019dfe0 sp=0xc00019df80
pc=0x4483c7
runtime.goexit()
    /usr/local/go/src/runtime/asm_amd64.s:1598 +0x1 fp=0xc00019dfe0 sp=0xc00019df80
pc=0x47c621
goroutine 2 [force gc (idle)]:
    runtime.gopark(0x0?, 0x0?, 0x0?, 0x0?, 0x0?)
    /usr/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc00006cfb0 sp=0xc00006cf90
pc=0x4487f6
runtime.goparkunlock(...)
```
/usr/local/go/src/runtime/proc.go:387
runtime.forcegchelper()
/ usr/local/go/src/runtime/proc.go:305 +0xb0 fp=0xc00006cfe0 sp=0xc00006cfb0
pc=0x448630
runtime.goexit()
/ usr/local/go/src/asm_amd64.s:1598 +0x1 fp=0xc00006cfe8 sp=0xc00006cfe0
pc=0x47c621
created by runtime.init.6
/ usr/local/go/src/runtime/proc.go:293 +0x25
goroutine  3  [GC sweep wait]:
runtime.gopark(0x0?, 0x0?, 0x0?, 0x0?, 0x0?)
/ usr/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc00006d780 sp=0xc00006d760
pc=0x4487f6
runtime.goexit(...)
/ us r/local/go/src/runtime/mgcsweep.go:278 +0x8e fp=0xc00006d7c8 sp=0xc00006d780
pc=0x43282e
runtime.gcenable.func1()
/ us r/local/go/src/runtime/mgc.go:178 +0x26 fp=0xc00006d7e0 sp=0xc00006d7e8
pc=0x427ae6
runtime.goexit() / us r/local/go/src/runtime/asm_amd64.s:1598 +0x1 fp=0xc00006d7e0 sp=0xc00006d7e0
pc=0x47c621
created by runtime.gcenable
/ us r/local/go/src/runtime/mgc.go:178 +0x6b
goroutine  4  [GC scavenger wait]:
runtime.gopark(0xc00003070?, 0xd19350?, 0x0?, 0x0?)
/ us r/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc00006d780 sp=0xc00006d760
pc=0x4487f6
runtime.goexit(...)
/ us r/local/go/src/runtime/mgc.go:178 +0x6b
goroutine 18  [finalizer wait]:
runtime.gopark(0x1a0?, 0x1192640?, 0xex0?, 0x024?, 0xc00006c770?)
/ us r/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc00006c628 sp=0xc00006c608
pc=0x4487f6
runtime.runfinq() / us r/local/go/src/runtime/mfinal.go:193 +0x107 fp=0xc00006c7e0 sp=0xc00006c628
pc=0x426b27
Dapr security audit 2023

runtime.goexit()
/usr/local/go/src/runtime/asm_amd64.s:1598 +0x1 fp=0xc00006c7e8 sp=0xc00006c7e0
pc=0x47c621
created by runtime.createfing
/usr/local/go/src/runtime/mfinal.go:163 +0x45

goroutine 19 [ID wait]:
runtime.gopark(0x0?, 0x0?, 0x0?, 0x0?, 0x0?)
/usr/local/go/src/runtime/proc.go:381 +0xd6 fp=0xc000181a78 sp=0xc000181a58
pc=0x4487f6
runtime.netpollblock(0x0?, 0x4100cf?, 0x0?)
/usr/local/go/src/runtime/netpoll.go:163 +0x45 fp=0xc000181d50 sp=0xc000181d00
pc=0xc048e17
internal/poll.runtime_pollWait(0x0, 0x419efe, 0xc00020e750, 0x7f2115855c38)
/usr/local/go/src/internal/poll/fd_unix.go:91 +0x14f fp=0xc000181d00 sp=0xc000181d00
pc=0x66a9bc
github.com/godbus/dbus.(*oobReader).Read(0xc0000a2000, 0xc000208470, 0xc000181e28, 0xc00000126a0)

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:21 +0x45 fp=0xc000181df0 sp=0xc000181d78 pc=0x8c1a405
io.ReadAtLeast(0x0d1d40, 0xc0000a2000, 0xc000208470, 0xc000181d50, 0xc0000a2000, 0xc000181d50, 0x4f9e3?

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:91 +0x11e fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405
io.ReadFull(...)

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:294 +0x3b fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:332 +0x9a fp=0xc000181e38 sp=0xc000181d78 pc=0x8c1a405

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:351 +0x351 github.com/godbus/dbus.(*unixTransport).ReadMessage(0xc000181d78)

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:381 +0x45 fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405
io.ReadAtLeast(0x0d1d40, 0xc0000a2000, 0xc000208470, 0xc000181d50, 0xc0000a2000, 0xc000181d50, 0x4f9e3?

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:91 +0x11e fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405
io.ReadFull(...)

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:294 +0x3b fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:332 +0x9a fp=0xc000181e38 sp=0xc000181d78 pc=0x8c1a405

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:351 +0x351 github.com/godbus/dbus.(*unixTransport).ReadMessage(0xc000181d78)

/home/adam/go/pkg/mod/github.com/godbus/dbus@v0.0.0-20190726142602-4481c8c300e2/transpor
t_unix.go:91 +0x11e fp=0xc000181f8 fp=0xc000181f8 sp=0xc000181f8 pc=0x8c1a405
io.ReadAtLeast(0x0d1d40, 0xc0000a2000, 0xc000208470, 0xc000181d50, 0xc0000a2000, 0xc000181d50, 0x4f9e3?
Root cause

The root cause of the issue is that the pulsar component’s dependency, github.com/hamba/avro/v2, allocates memory of a size controllable by the payload:

https://github.com/hamba/avro/blob/v2.5.0/reader.go#L201C1-L211C2

```go
func (r *Reader) ReadBytes() []byte {
    size := r.ReadLong()
    if size < 0 {
        r.ReportError("ReadBytes", "invalid bytes length")
        return nil
    }

    buf := make([]byte, size)
    r.Read(buf)
    return buf
}
```

Exploitability

There is no evidence of malicious actors exploiting this in the wild, and the likelihood of Dapr users being affected by this is low; For users to be affected, they would need to use the Pulsar pubsub component and explicitly set a non-default Avro schema.
SLSA

SLSA is a framework for assessing the supply chain security posture of projects. The current version of SLSA is v1.0 which specifies a series of requirements related to the build platform for software releases as well as the provenance attestation. SLSA evaluates a project based on four security levels. Level 0 has no requirements as we do not include that in the table below.

The SLSA framework is intended to protect against a series of supply chain attack vectors that we have seen in the wild, for example attacks like build platform compromisation which was the case with the SolarWinds attack, where attackers compromised the build platform of a software vendor - SolarWinds - and injected malicious code that the vendor then distributed to its customers. The provenance statement helps users defend against typosquatting attacks or attacks as well as consuming packages from mirrors instead of the main and intended packages. These are known attack vectors; recently researchers found 1,652 malicious packages disguised as legitimate packages⁸. SLSA compliance is therefore an important factor of Daprs overall security posture and should be seen as an ongoing effort to achieve and maintain a solid integration with SLSAs specification. This will help Dapr defend against a series of well-known - by users and malicious actors - supply chain attack vectors.

Our overall assessment is that Dapr performs well against requirements for the build platform but is lacking the provenance statement. The provenance is a large and important part of the SLSA framework. Dapr is currently lacking a compliant provenance statement which brings compliance to a low level. Dapr performs well in other areas such as the build platform for release artifacts. We recommend adding the provenance generation via SLSA’s official Github Actions workflows: https://github.com/slsa-framework/slsa-github-generator. The SLSA community is currently working on a framework, Bring Your Own Builder (BYOB), which includes level 3 compliance out of the box.

Our assessment for each criteria:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provenance generation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provenance Exists</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Provenance is Authentic</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Provenance is Unforgeable</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hosted</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Isolated</strong></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
**Supply-chain mitigations**

During this audit we’ve found that the Dapr codebase is written to a high security standard and follows best security practices. This is reflected in the low number of issues we have found in the Dapr code assets that were in scope of this audit. We consider the supply-chain risk to be an area where Dapr faces a security risk, and in this section we recommend that Dapr adds Scorecard to their dependencies to mitigate this risk.

During the manual auditing and fuzzing goals of the audit, we found several issues that relate to Daprs supply-chain. We consider the threat from a malicious dependency to be the most substantial of these. Dapr has a wide attack surface from malicious dependencies with a large dependency tree:

<table>
<thead>
<tr>
<th>Repository</th>
<th># of 3rd-party dependencies*</th>
<th># of 3rd-party nodes in callgraph*</th>
</tr>
</thead>
<tbody>
<tr>
<td>github.com/dapr/components-contrib</td>
<td>~ 400</td>
<td>~ 250,000</td>
</tr>
<tr>
<td>github.com/dapr/dapr</td>
<td>~ 340</td>
<td>~ 105,000</td>
</tr>
<tr>
<td>github.com/dapr/kit</td>
<td>~ 20</td>
<td>~ 2600</td>
</tr>
</tbody>
</table>

*Includes direct and transitive dependencies used for production and test code. Does not include calls to the standard library but includes calls to standard library addon libraries (goLang.org/x/...). This data has been generated by way of Class Hierarchy Analysis.

A dependency can become malicious from a code change by either a contributor or a maintainer. A contributor can trick a maintainer into merging a pull request by obfuscating the malicious code, or they can assume the role of maintainer by obtaining control over the repository. Both maintainers and contributors can make malicious commits intentionally and unintentionally with both having been exercised in the wild. A case of an intentional malicious commit made by a maintainer is the node-ipc⁹ library, to which the maintainer purposefully added a vulnerability that specifically targeted Russian users of the library¹⁰. Alternatively, a threat actor can become a maintainer and then add vulnerabilities to the software package. There are examples of real-world cases where a person - or group of people - sent an email to an open source project maintainer and asked if they could carry forward their archived project and succeeded in achieving maintainer status. To increase their chances, threat actors can build up trust by making a long line of legitimate code contributions before making a malicious one.

---

⁹ [https://github.com/RIAEvangelist/node-ipc](https://github.com/RIAEvangelist/node-ipc)

¹⁰ [https://www.theregister.com/2022/03/18/protestware_javascript_node_ipc/](https://www.theregister.com/2022/03/18/protestware_javascript_node_ipc/)
This type of risk applies to all open source projects that use other open source packages in their dependency trees. The Scorecard project\(^{11}\) aims to mitigate that risk by formalizing a set of security heuristics to evaluate the security practices of a software project. We recommend that Dapr long-term adds Scorecard to its dependencies to evaluate the ongoing improvements to the security of Daprs 3rd-party dependencies.

\(^{11}\) https://github.com/ossf/scorecard