Fine penetration tests for fine websites

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

Pentest-Report Harbor 10.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. D. Weißer, J. Larsson,
BSc. J. Hector, MSc. N. Krein, M. Kinugawa

Index

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

Introduction

“Harbor is an open source cloud native registry that stores, signs, and scans container images for vulnerabilities.

Harbor solves common challenges by delivering trust, compliance, performance, and interoperability. It fills a gap for organizations and applications that cannot use a public or cloud-based registry, or want a consistent experience across clouds.”

From https://goharbor.io/

This report documents the findings of a security assessment targeting the Harbor software compound. Carried out by Cure53 in October 2019, this project entailed a penetration test and a source code audit. It should be emphasized that this assessment was requested and generally sponsored by CNCF.

As for the methods, the typical approach of using a white-box methodology across various CNCF-funded projects was agreed upon between Cure53 and the Harbor team. Therefore, Cure53 had access to sources, as well as could take advantage of the carefully set up environments and documentation provided by the Harbor team. Moreover, consultations took place during thorough briefings, with the Harbor team voicing their opinion about the key areas that penetration test and audits should focus on.

Given the careful planning and good preparatory phase, the Cure53 team managed to execute the assessment smoothly and efficiently. The project was specifically completed in October 2019 and involved seven testers from the Cure53 team. A total budget spent on the project amounted to eighteen person-days, including resources for core testing, documentation and other project-related tasks. Among them, communication between Cure53 and the Harbor team was managed through a dedicated Slack channel. CNCF Slack workspace was joined by the relevant personnel from Cure53 and Harbor. All exchanges were pleasant and productive, with Cure53 furnishing regular status and progress updates. Note, however, that live-reporting was not requested by the Harbor team.

Cure53 managed to spot six security-relevant findings. Four were classified as security vulnerabilities and two should be seen as general weaknesses. It is worth noting that two vulnerabilities were given “*High*” severity ratings and one even reached a “*Critical*” score. These most significant problems indicate exploitable XSS on an interesting origin, SQL injection and a CSRF issue that would lead to privilege escalation if exploited successfully. While the Harbor software made a well-rounded impression, the results in

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

terms of security posture are not yet optimal, calling for more hardening work across various areas.

In the following sections, the report will first briefly reiterate the scope and then moves on to dedicated, highly-detailed notes on methodology and coverage. Next, tickets are discussed in chronological order and shed light on the discoveries one-by-one. Alongside technical aspects like PoCs, Cure53 furnishes mitigation advice and fix notes when applicable. The report closes with a summary of this October 2019 project and includes a verdict about the tested scope. Conclusions about the security and privacy posture of the Harbor software complex are supplied in the final section of this document.

Scope

- **Harbor & Harbor Helm**
  - https://github.com/goharbor/harbor/tree/v1.9.1-rc1
  - https://github.com/goharbor/harbor-helm/tree/v1.2.0
- **Environments & Info**
  - Two environments were provided for Cure53
  - Credentials were given to Cure53 for accessing both via SSH

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

Test Methodology

The following paragraphs describe the testing methodology used during the assessment of the Harbor complex. The test was divided into two phases, each fulfilling different goals. In the first phase, the focus was on code assisted penetration testing with the general aim of getting a general sense of how the platform functions and which routes on the API require more privileges than others. In addition, the general functionality of the platform was explored to spot areas that may be of interest and require more focus during the later parts of the audit. In the second phase, Cure53 performed a dedicated investigation of the Harbor codebase through a source code analysis.

The adopted bifocal approach means that passive background scanning of the platform can help spot low-hanging fruit, such as missing HTTP security headers or obvious input sanitization issues that trigger server-side errors. After gaining a high-level overview of the Harbor platform, its different functionalities and user-roles with their multilayered privileges, Cure53 utilized a more focused approach of auditing key areas within the Harbor's source code.

Part 1: Code-Assisted Penetration Testing

The following list documents the distinguishable steps taken during the first part of the test against the Harbor's software compound. In this phase of the project, Cure53 focused on a manual approach and obtaining a broad understanding of the platform. The aim was to recognize key areas that deserve dedicated and more granular security evaluations.

- To gain an overview of the platform and its different functionalities, Cure53 started with a manual approach in a form of standard web pentesting. As soon as an actual vulnerability was found, the source code was brought in to the analysis as a reference to understand the vulnerability in more detail.
- Starting with the passive background scanning by using proxy servers that intercept all HTTP traffic, Cure53 enumerated all routes that the application configures within its web-controllers and API.
- It quickly became apparent that Harbor uses the modern approach of a web application that is split into different microservices. Each of those has a separate role and they are mostly glued together by HTTP again (with the exception of services such as SQL). In Cure53's experience, this usually opens the door for split API injections where an attacker can smuggle additional paths and parameters into internally generated HTTP requests that branch into separate APIs. Time was thus spent on figuring out if attackers can reach internal APIs they do not usually have access to from the outside. Since no severe case was

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

identified, the discovery of this insecure code pattern was moved into Part 2 of the audit. More details on this vulnerability can be found in HAR-01-003.

- During further general web pentesting, it was noticed that Harbor makes sure that generated API responses are free from malicious HTML and that each user-influenced return value is output-validated with its content-type explicitly set. This makes it hard for attackers to find reflected XSS vulnerabilities and is likely possible exclusively with a bug in the generated templates. During manual testing it was not possible to find a valid XSS issue. This concerns the absence of simple reflected ones, stored XSS via the *Markdown* renderer or via DOMXSS. One exception was found in the Swagger UI (HAR-01-006), but this is not directly Harbor's fault and may be easily fixed by making sure this software package is always up-to-date.
- Cure53 quickly noticed that Harbor omits CSRF tokens for unknown reasons. While this is often not a problem *per se* when the web application makes sure that authentication tokens must be provided for each request, Harbor's authentication mechanism is mostly based on sessions. Still, an actual exploit with "*Critical"* severity was only found in later during manual audits when discovering that Harbor only loosely checks the request's content-type, making certain routes highly vulnerable to CSRF compromises.
- Cure53 dedicated attention to discovering vulnerabilities in the ACL models of Harbor. With Harbor's different user-groups and capabilities to only run certain actions when a user has enough privileges, it is often a tedious task to verify that all controllers implement the correct ACL checks in the source code, unless they are globalized. Instead, a semi-automated approach privilege checks was applied by, for example, recording all routes and actions a privileged user can perform. Afterwards, the requests' sessions were switched to a low-privileged to see which items are still passed.
- Cure53 hunted for mass-assignment vulnerabilities, such as those described in *CVE-2019-16097,* doing so by looking at other places where mass-assignments are potentially possible. Since this approach yielded no bugs, it was later verified if similar issues can be spotted in the code.
- During manual testing and automated scanning, Cure53 noticed multiple exceptions that raised HTTP 500 error codes. Since access to the Harbor platform was given via SSH, Cure53 could simply latch onto the log files and observe what errors were triggered. This explains how the SQL Injection in HAR-01-002 was originally found. Since the unsanitized user-supplied parameter directly triggered an SQL error printed to the log files, it was easily possible to verify the bug and create a quick PoC. The PoC that allows information extraction via an error-based side-channel. This bug was also analyzed in detail to formalize a pattern that could be searched for in the later phases of the pentest.

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

- All in all, the manual approach has already yielded a few bugs where detailed source code analysis was necessary to find variations. Similar security issues manifested themselves and this step also yielded an interesting, little-more hidden and under-tested functionality.

Part 2: Manual Code Auditing

This section lists the steps that were undertaken during the second phase of the audit against the Harbor software compound. It describes the key aspects of the manual code audit and highlights Cure53's approach into finding vulnerabilities in the core scope items. Further, it indicates strategies used for analyzing bugs from the previous approach and linked to variation analysis.

- Since the previous short round of manual testing already yielded security issues, Cure53 decided that that audit the source code for similar problems is worthwhile.
- Considering the SQL Injection from the previous phase, Cure53 checked Harbor's source code for similar code patterns, especially *sprintf* being used in an insecure manner, i.e. no parameterization with prepared statements in place. This approach allowed to find another issue filed under HAR-01-004. Although it was not possible to directly confirm this issue during the runtime of this test, Cure53 feels confident that it should be fixed.
- Regarding the missing CSRF tokens, Cure53 noticed that *Beego* (Harbor's app framework) has a mediatype filter which whitelists *multipart/form-data* across all API routes. Cure53 then checked all routes defined in *router.go* to see what impact could be generated from this. In essence, it was discovered that a simple *multipart/form-data* request was enough to CSRF administrative users and insert new admins into the user-database. This can be considered a severe issue.
- Further manual source code verification of the authentication mechanisms was done to see if LDAP injection, among others, was an issue. Luckily, this is not the case since *gopkg.in/ldap*'s escaping routines are used. *DB* as authentication mechanism can also be considered secure. Remaining analysis of other authentication mechanisms yielded no issues.
- However, since Harbor exports endpoints that can be used by Docker to tag and push images, the implementation of the *authentication* token was analyzed. A weakness with the token-generation was found and essentially allows project takeover in certain cases. This is described in more detail in HAR-01-005.
- Cure53 covered important functionalities that generally resulted in vulnerabilities in the past. For example, webhooks features that are usually a target for SSRF vulnerabilities need to implement strict protocol whitelists and make sure that attackers cannot smuggle additional input inside the generated requests. These

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

features were found to be defensively implemented, exposing no room for attacks.

- Cure53 also spent time on analyzing core scope items such as the handling and extraction of uploaded files, e.g. the compressed Helm charts. It was verified, both manually and in the source code, that the jobservers responsible for parsing chart-data do not accidentally follow symlinks or extract their files into unexpected directories. The download features were also analyzed to make sure that only the basename of the file path is passed to the download handler.
- Other aspects - such as vulnerability scanning, logging functionalities and handling of docker images - received similar treatment in both manual test and and in source code verification. Cure53 examined whether the features were implemented defensively and utilized the escaping functionalities of their libraries. Cure53 arrived at a positive impression about the remaining scope areas.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in 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. *HAR-01-001*) for the purpose of facilitating any future follow-up correspondence.

HAR-01-001 Web: Missing CSRF protection leads to privilege escalation *(**Critical**)*

It was discovered that the Harbor web interface does not implement protection mechanisms against Cross-Site Request Forgery (CSRF). By luring an authenticated user onto a prepared third-party website, an attacker can execute any action on the platform in the context of the currently authenticated victim. The following HTML snippet would force an administrator to add a user with *admin* privileges.

Proof-of-Concept:

```
<html>
you have just been CSRF'd
<script>
  var url = "https://18.191.179.250/api/users";
  fetch(url, {
    method: 'POST',
    credentials: 'include',
    headers: {
    'Content-Type': 'multipart/form-data'
    },
```

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

```
    body: '{"username":"csrf_user","email":"csrf@foo.bar","realname":"csrf
created","password":"Harbor12345","comment":null,"has_admin_role":true}'
  });
</script>
</html>
```

The HTML snippet works since JavaScript's *fetch* function adds the current user cookies when issuing a request. Furthermore, the Harbor interface accepts *json* encoded data despite the Content-Type header being set to *multipart/form-data*. As shown below, the latter is checked using a *Beego* filter.

Affected File:
src/core/main.go

Affected Code:

```
func main() {
[...]
    filter.Init()
    beego.InsertFilter("/*", beego.BeforeRouter, filter.SecurityFilter)
    beego.InsertFilter("/*", beego.BeforeRouter, filter.ReadonlyFilter)
    beego.InsertFilter("/api/*", beego.BeforeRouter,
filter.MediaTypeFilter("application/json", "multipart/form-data",
"application/octet-stream"))
```

The crucial part is that the filter allows a content-type of *multipart/form-data*, which in return lets an attacker use the *fetch()* API without triggering a preflight request that checks the CORS headers (as opposed to setting the content-type to *application/json*).

Missing CSRF protections are a fundamental flaw and the fact that the issue can be exploited without limitations renders it *"Critical"*. It is recommended to make use of the CSRF tokens in order to prevent forgery of requests. *Beego* has a fairly good documentation[1] on how to make use of their built-in CSRF protection.

HAR-01-002 API: SQL Injection via *project quotas* (*High*)

An SQL Injection was found in the *quotas* section of the Harbor API. An authenticated administrator can send a specially crafted SQL payload through the *GET* parameter *sort*, allowing the extraction of sensitive information from the database. The vulnerable lines of code can be seen in the following excerpts.

Affected Files:
harbor-1.9.1-rc1/src/common/dao/quota.go

[1] https://beego.me/docs/mvc/controller/xsrf.md

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

harbor-1.9.1-rc1/src/common/dao/quota_usage.go

Affected Code:

```
func castQuantity(field string) string {
  // cast -1 to max int64 when order by field
  return fmt.Sprintf("CAST( (CASE WHEN (%[1]s) IS NULL THEN '0' WHEN (%[1]s) =
'-1' THEN '9223372036854775807' ELSE (%[1]s) END) AS BIGINT )", field)
}

func quotaOrderBy(query ...*models.QuotaQuery) string {
  orderBy := "b.creation_time DESC"

  if len(query) > 0 && query[0] != nil && query[0].Sort != "" {
     if val, ok := quotaOrderMap[query[0].Sort]; ok {
        orderBy = val
     } else {
        sort := query[0].Sort

        order := "ASC"
        if sort[0] == '-' {
           order = "DESC"
           sort = sort[1:]
        }

        prefix := []string{"hard.", "used."}
        for _, p := range prefix {
           if strings.HasPrefix(sort, p) {
              field := fmt.Sprintf("%s->>'%s'", strings.TrimSuffix(p, "."),
                    strings.TrimPrefix(sort, p))
              orderBy = fmt.Sprintf("(%s) %s", castQuantity(field), order)
              break
           }
        }
     }
  }

  return orderBy
}
```

The user-controlled *GET* parameter flows into the *sort* variable of the *quotaOrderBy()* function highlighted above. The input is then concatenated into the *field* placeholder and further processed by the *castQuantity()* function, which embeds the potentially malicious user-input unsanitized into a raw SQL statement. This sequence leads to vulnerability.

PoC URL:
https://18.191.179.250/api/quotas?sort=used.count%27||(case+when+version()like+%27Postgre%25%27+then+1+else+(select+1+union+select+2)+end)||%27

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

In a similar way, the *quotaUsageOrderBy()* function of the *quota_usage.go* file is affected by this issue. It is not possible to exploit this particular function in the current release since it is only called without an argument. However, it is recommended to fix this vulnerability to protect future releases of Harbor. Although an authenticated administrator is required to exploit this vulnerability, an unauthenticated attacker can take advantage of this through CSRF, namely by luring an authenticated, benign administrator onto a malicious website. Therefore, this issue was rated as "*High*".

HAR-01-005 ACL: Unauthorized project-access through *project name* (Medium)

The implementation of access control allows unauthorized access to a project by leveraging a *JWT* token which claims a specific *project name*. As soon as the project is deleted and another one is created under the same name, the *JWT* token is valid for the new project, thus allowing unauthorized access.

The *JWT* token will grant access to specific actions on a *project repository* with a specific name of the Docker registry. However, the *JWT* token is not invalidated after a project's deletion. Any other project which will be created with the same name will now be vulnerable to unauthorized access by the *JWT* token.

The vulnerability can be confirmed with the following steps. These must be completed in order.

Steps to Reproduce:

1. Authenticate as an ordinary user
2. Create a project called *project123*
3. Tag and push image by CLI
4. *docker tag <imagename> 18.191.179.250/project123/imagename*
5. *docker push 18.191.179.250/project123/imagename*
6. Get a valid *JWT* token to access Docker registry at https://18.191.179.250/service/token?scope=repository%3aproject123%2Fimagename%3Apush%2Cpull&service=harbor-registry
7. Delete the project
8. Switch to another user and create a project with the same name, i.e. *project123*
9. Repeat *Step 3.*
10. Check token validity with *curl* providing unauthorized access to registry:
    - *curl 'https://18.191.179.250/v2/project123/imagename/tags/list' -k -H 'Authorization: Bearer <JWT Token>'*

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

JWT Claims:


```
{"iss":"harbor-token-issuer","sub":"user","aud":"harbor-
registry","exp":1570800092,"nbf":1570798292,"iat":1570798292,"jti":"bnkrmgGVDyee
mJhl","access":[{"type":"repository","name":"project123/imagename","actions":
["push","*","pull"]}]}
```


By leveraging this token, an unauthorized attacker can now push and pull images to and from the registry. It is recommended to bind the *JWT* claims to non-ambiguous project identifiers which cannot be reused by other projects.

HAR-01-006 Web: DOMXSS in outdated Swagger UI *(High)*

It was found that an XSS vulnerability exists in the used Swagger UI[2]. This UI loads the URL specified in the "*url*" parameter and uses the contents to render the HTML. The HTML written there is sanitized using the DOMPurify sanitize library[3] and then rendered. However, since the DOMPurify library is not used in the latest version, XSS can occur through a known bypass[4]. The problem can be reproduced via the following URL on Google Chrome browser.

PoC:

https://18.191.179.250/devcenter?url=data:,openapi:%20%223.0.0%22%0A%0Ainfo:%0A%20%20title:%20XSS%20via%20DOMPurify%20Bypass%0A%20%20description:%20%0A%20%20%20%20%3Ch4%3ETEST%3C%2Fh4%3E%0A%20%20%20%20%3Csvg%3E%3C%2Fp%3E%3Ctitle%3E%3Ctemplate%3E%3Cstyle%3E%3C%2Ftitle%3E%3Cimg%20src%20onerror%3Dalert(document.domain)%3E

It is recommended to update Swagger UI to 3.23.11, which is the latest version[5]. Despite versioning of DOMPurify library, the issue should be solved either way due to the *style* element disabling XSS.

[2] https://swagger.io/tools/swagger-ui/
[3] https://github.com/cure53/DOMPurify
[4] https://github.com/cure53/DOMPurify/releases/tag/2.0.3
[5] https://github.com/swagger-api/swagger-ui/releases/tag/v3.23.11

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

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.

HAR-01-003 API: Split-API Injections due to unsanitized input *(Low)*

It was discovered that unsanitized user-input was being used to construct endpoint URLs for server-side requests to the chart server. During this test, this could not be exploited in an impactful way, thus ending up in the *miscellaneous* section. However, the problem entails a bad practice and can lead to a potentially more serious issue in the future.

Below is a Proof-of-Concept (PoC) request demonstrating the issue. Note the highlighted part, which contains the path traversal (../).

PoC Request:

```
GET /api/chartrepo/library/charts/nginx%3Cs%3Exxx/0.1.0%2ffoo%2f..%2f HTTP/1.1
Host: 18.191.179.250
[...]
Cookie: sid=32f875103335589b3a9c768fccf9b884; harbor-lang=en-us
```

The affected code is listed below, with the relevant parts highlighted.

Affected File:
src/chartserver/handler_manipulation.go

Affected Code:

```
func (c *Controller) GetChartVersion(namespace, name, version string)
(*helm_repo.ChartVersion, error) {
    if len(namespace) == 0 {
        return nil, errors.New("empty namespace when getting summary of chart
version")
    }

    if len(name) == 0 || len(version) == 0 {
        return nil, errors.New("invalid chart when getting summary")
    }

    url := fmt.Sprintf("%s/%s/%s", c.APIPrefix(namespace), name, version)

    content, err := c.apiClient.GetContent(url)
```

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

Note that the above is just an example from the chart server part. In other words, there are multiple instances where *sprintf* is used to construct the URL path with untrusted user-input. It is recommended to revisit all areas where a URL is constructed based on user-input and ensure prior sanitization.

HAR-01-004 Auth: Potential SQL Injection via user-groups *(High)*

While analyzing Harbor's source code to find more vulnerabilities similar to HAR-01-002, it was possible to notice another case of dangerous *sprintf* usage. In the vulnerable code shown below, the SQL queries that are later sent to the database are dynamically generated. This is done by using *sprintf* with *%s* as a format parameter instead of using prepared statements.

Affected File:
harbor-1.9.1-rc1/src/core/auth/authproxy/auth.go

Affected Code:
```
func (a *Auth) Authenticate(m models.AuthModel) (*models.User, error) {
[...]
            ugList := reviewResponse.Status.User.Groups
            log.Debugf("user groups %+v", ugList)
            if len(ugList) > 0 {
                  groupIDList, err := group.GetGroupIDByGroupName(ugList,
common.HTTPGroupType)
```

Affected File:
harbor-1.9.1-rc1/src/common/dao/group/usergroup.go

Affected Code:
```
func GetGroupIDByGroupName(groupName []string, groupType int) ([]int, error) {
      var retGroupID []int
      var conditions []string
      if len(groupName) == 0 {
            return retGroupID, nil
      }
      for _, gName := range groupName {
            con := "'" + gName + "'"
            conditions = append(conditions, con)
      }
      sql := fmt.Sprintf("select id from user_group where group_name in ( %s )
and group_type = %v", strings.Join(conditions, ","), groupType)
```

As one can see in the code above, the current user-groups are stored in the *ugList* variable that gets passed to *GetGroupIDByGroupName().* It is later embedded there in a *SELECT* statement without any form of sanitization.

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

Since the code in question hides behind *authproxy* / OIDC as authentication provider and the test-platforms were configured to use either *LDAP* or *DB,* it was not possible to trigger the problem during the runtime of this test. Still, with user-groups that can be set arbitrarily, Cure53 feels confident that a user with *Project-Admin* capabilities can exploit this SQL Injection to read secrets from the underlying database or conduct privilege escalation.

It is recommended to fix this issue by removing the dangerous *sprintf* call and instead parameterize the *gname* variable via prepared statements.

Conclusions

As noted already in the *Introduction,* this Cure53 assessment of the Harbor software compound concludes with mixed results. On the one hand, seven members of the Cure53 who spent eighteen days on the scope in October 2019, can certainly call Harbor a modern web application that follows various up-to-date and modern security practices. On the other hand, this CNCF-sponsored project exposed the shortcoming of the scope as well. Particularly, the Cure53 demonstrated that Harbor suffers from a number of security issues that are usually found in completely untested applications.

The conclusion proposed by Cure53 is mainly drawn from findings such as HAR-01-001 and HAR-01-002. The first one describes a severe CSRF flaw that is exploitable simply because Harbor does not implement anti-CSRF tokens. As it stands, this is a fairly important security principle for each web application and Cure53 cannot find any justification as to why Harbor simply skips it. The other describes a SQL Injection, another bug class that is less often found these days. The latter is because modern frameworks utilize prepared statements, thus eradicating SQL Injections altogether. Nevertheless, Harbor often takes the wrong route and generates queries in an insecure manner. This is not the best sign for a web application that uses a language and framework that actually make it hard to generate vulnerabilities.

While the words above may sound harsh and the found vulnerabilities are in fact severe, they do not signify that the Harbor complex is in a particularly bad shape. For example, the number of findings is very low and the overall pentest results and general impressions about the codebase are rather positive. Lack of further “*Critical”* input sanitization issues is definitely a good indicator as well.

Furthermore, Cure53 feels obliged to state that the Harbor code is clean, easy to follow and yields itself well to auditing. Similarly, the platform is built upon containerized microservices that raise the security level in a non-negligible way. It is worth pointing out

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

that even if one part of the application is compromised, it does not necessarily mean that the server is also directly prone to successful attacks. The separation of duty principle is implemented in a praiseworthy manner.

It is also important to note that Cure53 did not find flaws in the areas that the Harbor was most worried about as a result of previous vulnerabilities residing there. In fact, these were almost completely issue-free. Access control via RBAC with its different user-models and groups is tightly implemented, uploaded files are handled safely and dangerous features such as webhooks are secure and robust.

To conclude, Cure53 feels that Harbor is fully legitimized in calling its software compound secure and trustworthy. The examined scope items exhibit strong security posture, even though this CNFC-funded project revealed that some problems exist. Once the issues reported by Cure53 on the basis of this October 2019 test are fixed, the project will likely offer even more security. It is hoped that the Harbor team can reflect on the causes and implications of the findings in order to prevent similar flaws from happening in the future.

Cure53 would like to thank Michael Michael, Steven Zou, Steven Ren, Alex Xu and Daniel Jiang of VMWare, 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.