



# CubeFS Security Audit

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

## **Prepared by**

Adam Korczynski, Ada Logics  
David Korczynski, Ada Logics

**Report version:** 1.0

**Published:** 2nd January 2024

This report is licensed under Creative Commons 4.0 (CC BY 4.0)

This page has intentionally been left blank

# Table of contents

---

Executive summary

# Executive summary

---

In the fall of 2023, Ada Logics conducted a security audit of CubeFS in a coordinated collaboration between Ada Logics, CubeFS, OSTIF and the CNCF. The CNCF funded the work. The security audit was a holistic security audit with the following goals:

1. Assess and formalize a threat model for CubeFS highlighting entrypoints, risks and at-risk components.
2. Review the CubeFS codebase for security vulnerabilities of any severity.
3. Review CubeFS's supply-chain maturity against SLSA.

To formalize the threat model, Ada Logics relied on three sources of information: 1) CubeFS's official documentation, 2) the CubeFS source tree and 3) feedback from the CubeFS maintainers. The manual review was performed against the threat model to allow the auditors to consider trust levels and threat actors as they were reviewing the code.

The report contains all issues found from both the threat modelling and manual code audit exercises. Five of these issues were exploitable by threat actors identified during the threat modelling, and these issues were assigned the following CVE's:

Issue	CVE	CVE severity
Authenticated users can crash the CubeFS servers with maliciously crafted requests	CVE-2023-46738	Moderate
Timing attack can leak user passwords	CVE-2023-46739	Moderate
Insecure random string generator used for sensitive data	CVE-2023-46740	Moderate
CubeFS leaks magic secret key when starting Blobstore access service	CVE-2023-46741	Moderate
CubeFS leaks users key in logs	CVE-2023-46742	Moderate

Ada Logics disclosed these findings responsibly to CubeFS through CubeFS's public Github Security Advisory disclosure channels. The CubeFS security response team responded to the disclosures with fixes in a timely manner and before the audit had been completed.

The SLSA review found that CubeFS scores low because it does not include provenance for releases. Ada Logics included practical steps for achieving SLSA Level 3 compliance.

## Strategic recommendations

---

In this section, we include our strategic recommendations for CubeFS to maintain a secure project moving forward. Several points in this section are reflected in "Found Issues" or other parts of the report, whereas some are only included here.

### Supply-Chain Security

CubeFS has undoubtedly included supply-chain security in its ongoing work. For example, CubeFS has adopted Scorecard, which considers several different aspects of supply-chain security risks in an automated manner. Nonetheless, Supply-chain Security is an area where CubeFS can improve its ongoing work. The audit found that releases are not signed and do not include provenance, which makes consumers vulnerable to known supply-chain risks. We have included practical steps to take to add this to releases. While CubeFS has integrated the Scorecard Github Action, CubeFS currently scores a 6,5 Scorecard score, which leaves room for improvement. Open and closed-sourced software ecosystems are seeing an increase in supply-chain attacks and their sophistication, with major recent attacks having had their first compromise in the software development lifecycle rather than after deployment.

### Static analysis

CubeFS uses automated SAST in its development pipeline however limited to only CodeQL for security tooling. During the audit, Ada Logics tested CubeFS with other SAST tools, which found true positives in the CubeFS code base. We recommend adding the GoSec and Semgrep tools as wellm and add `ignore` directives for false positives.

### Security-relevant documentation

CubeFS has good documentation but lacks a dedicated security-best-practices section to help users deploy a security-hardened CubeFS instance. We recommend adding and maintaining this to ensure users can consume CubeFS in a secure manner and avoid security issues arising from misconfiguration.

During the security audit, the CubeFS team added a security-best-practices section to the official CubeFS documentation which is available here:

[https://cubefs.io/docs/master/maintenance/security\\_practice.html](https://cubefs.io/docs/master/maintenance/security_practice.html)

# Project Scope

---

The following Ada Logics auditors carried out the audit and prepared the report.

Name	Title	Email
Adam Korczynski	Security Engineer, Ada Logics	Adam@adalogics.com
David Korczynski	Security Researcher, Ada Logics	David@adalogics.com

The following CubeFS team members were part of the audit.

Name	Title	Email
Leon Chang	maintainer	changliang@oppo.com
Xiaochun He	maintainer	hexiaochun@oppo.com
Baijiaruo	maintainer	huyao2@oppo.com
Lei Zhang	maintainer	zhanglei12@oppo.com

The following OSTIF members were part of the audit.

Name	Title	Email
Derek Zimmer	Executive Director, OSTIF	Derek@ostif.org
Amir Montazery	Managing Director, OSTIF	Amir@ostif.org
Helen Woeste	Project coordinator, OSTIF	Helen@ostif.org

# Threat model

---

In this part, we look at CubeFS's threat model. We have used open-source materials to formalize the threat model including mainly from documentation produced by the CubeFS ecosystem, recorded talks, presentations and third-party documentation.

CubeFS is a cloud-native data storage infrastructure often used on top of databases, machine-learning platforms and applications deployed on top of Kubernetes. It supports multiple access protocols like S3, POSIX and HDFS with flexibility for consumers using multiple protocols in the same deployment.

CubeFS has four main components: 1) A metadata subsystem, 2) a data subsystem, 3) a resource management node also called "Master" and 4) an Object Subsystem. Below, we enumerate the components.

## Metadata subsystem

The Metadata subsystem runs the `MetaNode` which stores all file metadata in the cluster. In Kubernetes, this is deployed as a `DaemonSet` K8s resource.

## Data subsystem

The data subsystem is known internally in CubeFS as `DataNode` and handles the actual storing of file data. It mounts a large amount of disk space to store file data. When using CubeFS with Kubernetes, `DataNode` is deployed as a `DaemonSet`.

## Resource management

The resource management component is called `Master` and is responsible for managing resources and maintaining the metadata of the whole cluster. When deploying CubeFS on Kubernetes, the Master Node is deployed as a `StatefulSet` K8s resource.

## Object Subsystem

This component runs `ObjectNodes` and acts as an interface between different protocols - HDFS, POSIX and S3 - such that CubeFS works as the underlying data store, and the user can operate CubeFS by way of either or several of these protocols. The Object Subsystem is also called the Object Gateway internally in the CubeFS ecosystem.

In addition to the four core components, CubeFS implements an `AuthNode` which handles authentication and authorization in a CubeFS deployment.

CubeFS is meant to be deployed in such a manner that it is available to users of varying permission levels. This means that at a high level, CubeFS must be resistant to malicious cluster users who have been granted access. For example, if an organization grants access to an employee who gets convinced by a competitor to steal or corrupt data, the CubeFS devops team must know the impact this employee has for risk mitigation and impact remediation purposes. User permissions in CubeFS should start at the lowest and increase with the permissions that CubeFS admins intend to add to the user.

There are at least two security-relevant implications for CubeFS's architectural and permission design:

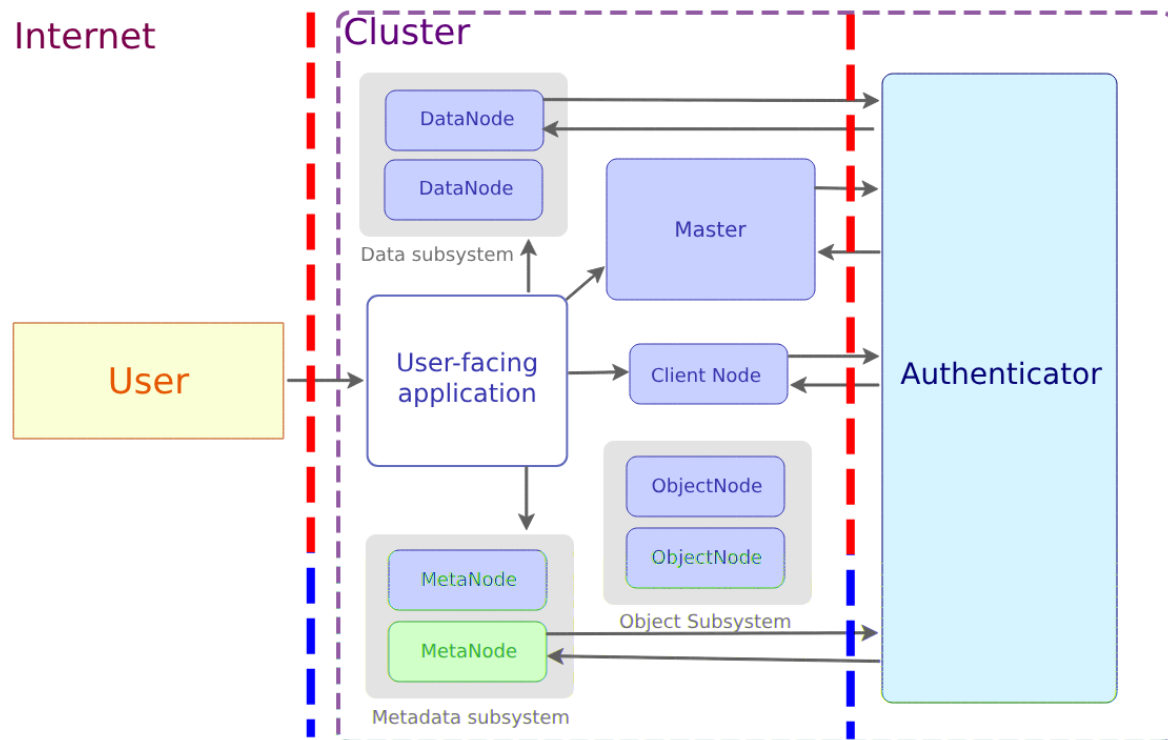
1. Users should not be able to achieve permissions they have not been granted. A permission should not imply another permission, whether intended or not. At this level, we are considering defined permissions that are not assigned to a user. This part of CubeFS's security model distinguishes between privileges at a granular level.
2. The second implication is the distinction between root and non-root permissions. CubeFS should accept a full cluster deletion by the cluster admin; it is not a security breach if the cluster admin or CubeFS admin can take down the entire cluster or cause any other harm to any part of CubeFS. There is an implied list of non-permitted actions that users should not be allowed to perform. These are general security risks that pertain to other software applications, such as Denial-of-Service attacks, stealing data, remote code execution, corruption of data and other general threats.

Most commonly, CubeFS is not exposed directly to the internet but will be available to services inside the cluster to which it is deployed. A CubeFS deployment will have multiple client nodes that include a client container, which is intended to communicate with the remaining CubeFS components. Communication between components happens via HTTP(S); Each component exposes a web server to the cluster. As such, threats are likely to come from users who already have a position in the cluster. This position can be through a legitimate use case - a user that should have access and has been granted so by the CubeFS admin, or it could be through a threat actor who has already escalated privileges and who seeks to further advance their position inside the cluster. In the former scenario, we have covered the expectations above, which we can sum up as such: If a legitimate user turns malicious, the CubeFS admin should know what their impact is and should be in control of reducing any permissions that the user has. In other words, what the CubeFS admin expects the user can do represents the user's privileges precisely. For the latter, CubeFS should reduce the ease with which an attacker can further escalate privileges inside the cluster.

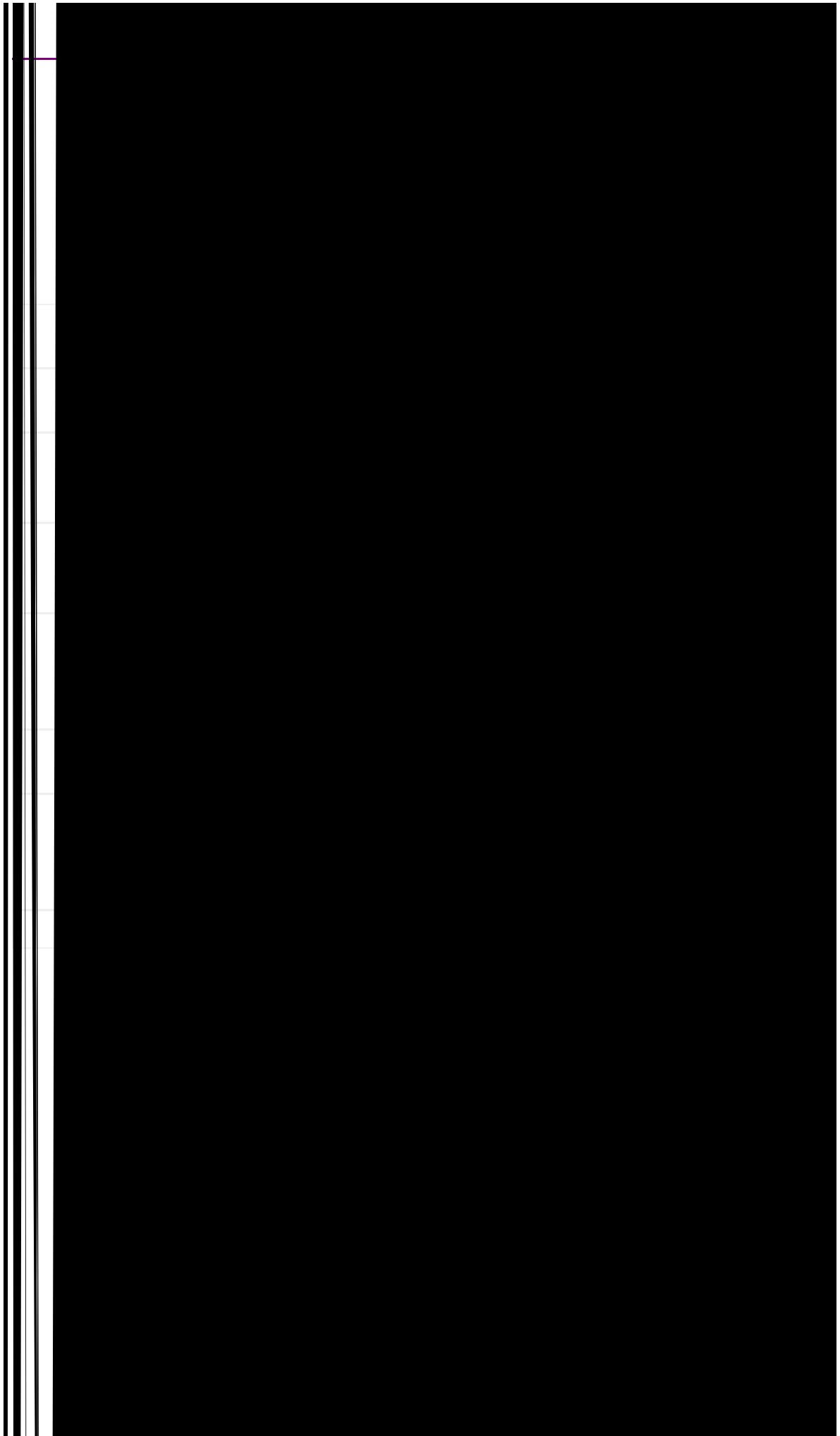


## Trust boundaries

In this section, we identify the trust boundaries of a CubeFS deployment. Below, we include a trust-flow diagram of an out-of-the-box CubeFS deployment:



Typically, a CubeFS deployment will be deployed alongside an internet-facing application in the cluster with which users communicate. When traffic enters the cluster, it crosses a trust boundary and flows low to high in the direction from the internet to the cluster. This trust boundary could also exist between the user-facing application and the CubeFS client nodes, depending on the specific use case. The reason for this is that the user-facing application could do its own validation and sanitization. From the user-facing application, traffic flows to the CubeFS client nodes. These authenticate the request before processing it, and the traffic crosses another trust boundary when being authenticated. At this point, trust flows low to high in the direction from the CubeFS client nodes to the authenticator. Trust remains high until CubeFS responds to the user external to the cluster.



---

# Issues found

---

Ada Logics found 12 issues during the audit. The list includes all issues found by way of manual auditing and fuzzing. Ada Logics uses a scoring system that considers impact and ease of exploitation. This is different from the CVSS scoring system, and there may be discrepancies between the severity assigned by Ada Logics and the severity resulting from a CVSS calculation.

#	Title	Status	Severity
1	Authenticated users can crash the CubeFS servers with maliciously crafted requests	Fixed	Moderate
2	CubeFS leaks magic secret key when starting Blobstore access service	Fixed	Moderate
3	CubeFS leaks users key in logs	Fixed	Moderate
4	Insecure cryptographic primitive used for sensitive data	Fixed	Moderate
5	Insecure random string generator used for sensitive data	Fixed	Moderate
6	Lack of security-best-practices documentation	Fixed	Moderate
7	Possible deadlocks	Fixed	Moderate
8	Possible nil-dereference from unmarshalling double pointer	Fixed	Low
9	Potential Slowloris attacks	Fixed	Low
10	Releases are not signed	Fixed	Moderate
11	Security Disclosure Email Does Not Work	Fixed	Low
12	Timing attack can leak user passwords	Fixed	Moderate

# Authenticated users can crash the CubeFS servers with maliciously crafted requests

---

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-NKbh4NJK
Component:	ObjectNode

The root cause is that when CubeFS reads the body of incoming requests, it reads it entirely into memory and without an upper boundary. As such, an attacker can craft an HTTP that contains a large body and exhausts memory of the machine, which results in crashing the server.

## Details

The issue exists across multiple CubeFS components. We have not made an exhaustive list and will follow up with that. For now, we exemplify the issue with the `deleteObjectsHandler` of the objectnode component. This handler reads the body of the incoming request entirely into memory on line 561 below:

```
https://github.com/cubefs/cubefs/blob/45442918591d25e7ab555469df384df468df5dbc/objectnode/api_handler_object.go#L532C22-L567

532 func (o *ObjectNode) deleteObjectsHandler(w http.ResponseWriter, r *http.Request) {
533     var (
534         err      error
535         errorCode *ErrorCode
536     )
537     defer func() {
538         o.errorResponse(w, r, err, errorCode)
539     }()
540
541     param = ParseRequestParam(r)
542     if param.Bucket() == "" {
543         errorCode = InvalidBucketName
544         return
545     }
546
547     var vol *Volume
548     if vol, err = o.getVol(param.Bucket()); err != nil {
549         log.Errorf("deleteObjectsHandler: load volume fail: requestID(%v)
volume(%v) err(%v)",
550             GetRequestID(r), param.Bucket(), err)
551         return
552     }
553
554     requestMD5 := r.Header.Get(ContentMD5)
555     if requestMD5 == "" {
556         errorCode = MissingContentMD5
557         return
558     }
559
560     var bytes []byte
561     bytes, err = ioutil.ReadAll(r.Body)
562     if err != nil {
563         log.Errorf("deleteObjectsHandler: read request body fail:
requestID(%v) volume(%v) err(%v)",
564             GetRequestID(r), param.Bucket(), err)
565         errorCode = UnexpectedContent
566         return
567     }
```

In this case, a user does not require permission to delete objects since the ACL check is done after reading the request body.

## PoC

We include two programs to reproduce this issue. Warning: save all work before running this PoC, including work in browser tabs.

The first program is a server that represents the `deleteObjectsHandler`. We have stripped unrelated parts of the function body that the HTTP request can easily pass legitimately. Start up this server by creating the following go module and run it with `go run main.go`:

```
1 package main
2
3 import (
4     "fmt"
5     "io/ioutil"
6     "net/http"
7 )
8
9 func main() {
10     http.HandleFunc("/deleteObjects", func(w http.ResponseWriter, r
11         *http.Request) {
12
13
14
15
16
17         fmt.Println("Got request")
18         _, err := ioutil.ReadAll(r.Body)
19         if err != nil {
20             return
21         }
22         fmt.Println("Finished reading body")
23     })
24
25     fmt.Printf("Starting server at port 8080\n")
26     if err := http.ListenAndServe(":8080", nil); err != nil {
27         panic(err)
28     }
29 }
30 }
```

You should see `Starting server at port 8080` in the terminal when starting this program.

The next program is the client. This program represents the malicious user who crafts a request with a large body and sends it to the server. Depending on the system used when running this program, it may be necessary to reduce or increase the size of the body. Create the following `main.go` in another module and run it with `go run main.go`

```
1 package main
2
3 import (
4     "io"
5     "strings"
6     "net/http"
7 )
8
9 func main() {
10     req := maliciousRequest()
11
12     _, err := http.DefaultClient.Do(req)
13     if err != nil {
14         panic(err)
15     }
16 }
17
18 func maliciousRequest() *http.Request {
19     s := strings.Repeat("malicious string", 100000000)
20     r1 := strings.NewReader(s)
21     r2 := strings.NewReader(s)
22     r3 := strings.NewReader(s)
23     r4 := strings.NewReader(s)
24     r5 := strings.NewReader(s)
25     r6 := strings.NewReader(s)
26     r7 := strings.NewReader(s)
27     r8 := strings.NewReader(s)
28     r := io.MultiReader(r1, r2, r3, r4, r5, r6, r7, r8)
```

```
29     req, err := http.NewRequest("POST", "http://localhost:8080/deleteObjects", r)
30     if err != nil {
31         panic(err)
32     }
33     return req
34 }
```

This request should exhaust memory temporarily and then crash the server.

### **Impact**

All CubeFS users are impacted by this issue.

# CubeFS leaks magic secret key when starting Blobstore access service

---

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-MNJHBrv3
Component:	BlobStore

CubeFS leaks secret configuration keys during initialization of the blobstore access service controller, more specifically here:

<https://github.com/cubefs/cubefs/blob/26da9925a3db98ff9a1e9a12cca2c457f736b831/blobstore/access/server.go#L76-L86>

```
76 func initWithRegionMagic(regionMagic string) {
77     if regionMagic == "" {
78         log.Warn("no region magic setting, using default secret keys for
checksum")
79         return
80     }
81
82     log.Info("using magic secret keys for checksum with:", regionMagic)
83     b := sha1.Sum([]byte(regionMagic))
84     initTokenSecret(b[:8])
85     initLocationSecret(b[:8])
86 }
```

Users with access to the logs can retrieve the secret key and escalate privileges to carry out operations on blobs that they otherwise don't have the necessary permissions for. For example, a threat actor who has successfully retrieved a magic secret key from the logs can delete blobs from the blob store by validating their requests in this step:

<https://github.com/cubefs/cubefs/blob/26da9925a3db98ff9a1e9a12cca2c457f736b831/blobstore/access/server.go#L546-L569>

```
546 func (s *Service) DeleteBlob(c *rpc.Context) {
547     args := new(access.DeleteBlobArgs)
548     if err := c.ParseArgs(args); err != nil {
549         c.RespondError(err)
550         return
551     }
552
553     ctx := c.Request.Context()
554     span := trace.SpanFromContextSafe(ctx)
555
556     span.Debug("accept /deleteblob request args:%+v", args)
557     if !args.IsValid() {
558         c.RespondError(errcode.ErrIllegalArguments)
559         return
560     }
561
562     valid := false
563     for _, secretKey := range tokenSecretKeys {
564         token := uptoken.DecodeToken(args.Token)
565         if token.IsValid(args.ClusterID, args.Vid, args.BlobID,
uint32(args.Size), secretKey[:]) {
566             valid = true
567             break
568         }
569     }
```

To exploit this security issue, the attacker needs to have privileges to read the logs. They could have obtained these privileges legitimately, or they could have obtained them by already having escalated privileges.



# CubeFS leaks users key in logs

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-vc34CGVVJB
Component:	Master

CubeFS leaks secret user keys and access keys in the logs in multiple components. When CubeFS creates new users, it leaks the user's secret key. This could allow a lower-privileged user with access to the logs to retrieve sensitive information and impersonate other users with higher privileges than themselves.

## Details

The vulnerable API that leaks secret keys is `createKey`:

<https://github.com/cubefs/cubefs/blob/26da9925a3db98ff9a1e9a12cca2c457f736b831/master/user.go#L43-L111>

```
43 func (u *User) createKey(param *proto.UserCreateParam) (userInfo *proto.UserInfo, err
error) {
44     var (
45         AKUser      *proto.AKUser
46         userPolicy  *proto.UserPolicy
47         exist        bool
48     )
49     if param.ID == "" {
50         err = proto.ErrInvalidUserID
51         return
52     }
53     if !param.Type.Valid() {
54         err = proto.ErrInvalidUserType
55         return
56     }
57
58     var userID = param.ID
59     var password = param.Password
60     if password == "" {
61         password = DefaultUserPassword
62     }
63     var accessKey = param.AccessKey
64     if accessKey == "" {
65         accessKey = util.RandomString(accessKeyLength,
util.Numeric|util.LowerLetter|util.UpperLetter)
66     } else {
67         if !proto.IsValidAK(accessKey) {
68             err = proto.ErrInvalidAccessKey
69             return
70         }
71     }
72     var secretKey = param.SecretKey
73     if secretKey == "" {
74         secretKey = util.RandomString(secretKeyLength,
util.Numeric|util.LowerLetter|util.UpperLetter)
75     } else {
76         if !proto.IsValidSK(secretKey) {
77             err = proto.ErrInvalidSecretKey
78             return
79         }
80     }
81     var userType = param.Type
82     var description = param.Description
83     u.userStoreMutex.Lock()
84     defer u.userStoreMutex.Unlock()
85     u.AKStoreMutex.Lock()
86     defer u.AKStoreMutex.Unlock()
87
88     if _, exist = u.userStore.Load(userID); exist {
89         err = proto.ErrDuplicateUserID
90         return
91     }
92     _, exist = u.AKStore.Load(accessKey)
93     for exist {
```

```

94         accessKey = util.RandomString(accessKeyLength,
util.Numeric|util.LowerLetter|util.UpperLetter)
95         _, exist = u.AKStore.Load(accessKey)
96     }
97     userPolicy = proto.NewUserPolicy()
98     userInfo = &proto.UserInfo{UserID: userID, AccessKey: accessKey, SecretKey:
secretKey, Policy: userPolicy,
99         UserType: userType, CreateTime: time.Unix(time.Now().Unix(),
0).Format(proto.TimeFormat), Description: description}
100     AKUser = &proto.AKUser{AccessKey: accessKey, UserID: userID, Password:
encodingPassword(password)}
101     if err = u.syncAddUserInfo(userInfo); err != nil {
102         return
103     }
104     if err = u.syncAddAKUser(AKUser); err != nil {
105         return
106     }
107     u.userStore.Store(userID, userInfo)
108     u.AKStore.Store(accessKey, AKUser)
109     log.Infof("action[createUser], userID: %v, accesskey[%v], secretkey[%v]",
userID, accessKey, secretKey)
110     return
111 }

```

`createKey` creates a `UserInfo`, an access key and a secret key and stores it in the respective stores. If `createKey` successfully creates all three pieces of information and successfully stores them, it will log the created pieces of information on this line:

<https://github.com/cubefs/cubefs/blob/26da9925a3db98ff9a1e9a12cca2c457f736b831/master/user.go#L109>

```

109     log.Infof("action[createUser], userID: %v, accesskey[%v], secretkey[%v]",
userID, accessKey, secretKey)

```

## Impact

An attacker who has access to the logs can see the secret key in plain text and impersonate the user. The attacker can either be an internal user with limited privileges to read the log, or it can be an external user who has escalated privileges sufficiently to access the logs.

To find the places where CubeFS logs the users `accessKey`, we refer to the following grep call: `grep -r "log\." . --exclude=*test.go | grep accessKey`. Not all occurrences of this constitute a vulnerability: Only cases of logging after authorization represent a security issue.

# Insecure cryptographic primitive used for sensitive data

---

Moderate

```

81     var userType = param.Type
82     var description = param.Description
83     u.userStoreMutex.Lock()
84     defer u.userStoreMutex.Unlock()
85     u.AKStoreMutex.Lock()
86     defer u.AKStoreMutex.Unlock()
87
88     if _, exist = u.userStore.Load(userID); exist {
89         err = proto.ErrDuplicateUserID
90         return
91     }
92     _, exist = u.AKStore.Load(accessKey)
93     for exist {
94         accessKey = util.RandomString(accessKeyLength,
95             util.Numeric|util.LowerLetter|util.UpperLetter)
96         _, exist = u.AKStore.Load(accessKey)
97     }
98     userPolicy = proto.NewUserPolicy()
99     userInfo = &proto.UserInfo{UserID: userID, AccessKey: accessKey, SecretKey:
100     secretKey, Policy: userPolicy,
101     UserType: userType, CreateTime: time.Unix(time.Now().Unix(),
102     0).Format(proto.TimeFormat), Description: description}
103     AKUser = &proto.AKUser{AccessKey: accessKey, UserID: userID, Password:
104     encodingPassword(password)}
105     if err = u.syncAddUserInfo(userInfo); err != nil {
106         return
107     }
108     if err = u.syncAddAKUser(AKUser); err != nil {
109         return
110     }
111     u.userStore.Store(userID, userInfo)
112     u.AKStore.Store(accessKey, AKUser)
113     log.LogInfof("action[createUser], userID: %v, accesskey[%v], secretkey[%v]",
114     userID, accessKey, secretKey)
115     return
116 }

```

An attacker who can retrieve the database records of users has a lower barrier for getting the actual passwords of users than if Cubefs used a secure primitive such as SHA2 or SHA3. To exploit this weakness, an attacker would already need to escalate privileges or gain access to database records from misconfiguration of a Cubefs deployment. Even so, an attacker has the potential for further escalating privileges by exploiting this weakness depending on the user credentials they can steal.

## Mitigation

We recommend using a secure primitive for user passwords. This would mitigate risk even if an attacker has access to the encrypted user passwords.

## Insecure ra

Severity:


s t

s K

ation, as stated in the document

Pv

**Impact**

An attacker could exploit the predictable random string generator and guess a users access key to impersonate the user and obtain higher privileges.

# Lack of security-best-practices documentation

---

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-vc34CGVVJB
Component:	CubeFS

CubeFS maintain documentation on how to easily get started with CubeFS, which is positive; however, CubeFS lacks a section or dedicated page on deploying and using CubeFS in a secure, production-ready manner.

We recommend setting up a dedicated page to accommodate this. See the Istio security-best-practices page for reference: <https://istio.io/latest/docs/ops/best-practices/security/>.

Without an officially maintained security-best-practices page, users may deploy CubeFS in ways that are known by the community to be insecure and obviously necessary for secure but also easy to overlook. Users should not be expected to read through the entire documentation to dissect the critical parts for deployment. Instead, we recommend a dedicated page for this purpose.

The work to maintain secure-best-practices documentation should be considered an ongoing process. Adding this to the documentation, maintaining it and developing it over time is good practice.

# Possible deadlocks

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-LK432hu
Component:	Multiple

Cubefs is susceptible to a number of deadlocks across multiple components. This is an umbrella issue for all identified possible deadlocks. Deadlocks happen when two threads or programs are waiting for each other to finish, where one of them does not finish. This has security implications if an attacker is able to cause the deadlock. The attacker will steer the execution of the program into a path where the program invokes a lock but does not unlock it.

Below we enumerate the places across the Cubefs source tree where this can happen.

## Rate limiter

Below, Cubefs locks the mutex on line 60 and unlocks it on line 72. Between the mutex lock and unlock, the method can exit in two places: line 63 and line 67.

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/util/ratelimit/keyratelimit.go#L58-L73>

```
58 func (k *KeyRateLimit) Release(key string) {
59
60     k.mutex.Lock()
61     limit, ok := k.current[key]
62     if !ok {
63         panic("key not in map. Possible reason: Release without Acquire.")
64     }
65     limit.refCount--
66     if limit.refCount < 0 {
67         panic("internal error: refs < 0")
68     }
69     if limit.refCount == 0 {
70         delete(k.current, key)
71     }
72     k.mutex.Unlock()
73 }
```

## flowctrl

A similar case to the Rate limiter exists in the `flowctrl` package:

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/util/flowctrl/keycontroller.go#L55-L71>

```
55 func (k *KeyFlowCtrl) Release(key string) {
56
57     k.mutex.Lock()
58     ctrl, ok := k.current[key]
59     if !ok {
60         panic("key not in map. Possible reason: Release without Acquire.")
61     }
62     ctrl.refCount--
63     if ctrl.refCount < 0 {
64         panic("internal error: refs < 0")
65     }
66     if ctrl.refCount == 0 {
67         ctrl.c.Close()
68         delete(k.current, key)
69     }
70     k.mutex.Unlock()
71 }
```

Cubefs locks the mutex on line 57 and unlocks it on line 70. The method can exit on lines 60 and 64 without unlocking.



## Metanode

Metanodes method for marshalling a value to bytes has a potential deadlock if the call to `binary.Write` fails with an error, which will cause the method to panic without releasing the lock.

Below, `MarshalValue()` locks on line 703 and unlocks on line 719. On line 709, the method panics without releasing the lock:

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/metanode/inode.go#L698-L721>

```
698 func (i *Inode) MarshalValue() (val []byte) {
699     var err error
700     buff := bytes.NewBuffer(make([]byte, 0, 128))
701     buff.Grow(64)
702
703     i.RLock()
704     i.MarshalInodeValue(buff)
705     if i.getLayerLen() > 0 && i.getVer() == 0 {
706         log.Log.Fatalf("action[MarshalValue] inode %v current verseq %v, hist
len (%v) stack(%v)", i.Inode, i.getVer(), i.getLayerLen(), string(debug.Stack()))
707     }
708     if err = binary.Write(buff, binary.BigEndian, int32(i.getLayerLen())); err
!= nil {
709         panic(err)
710     }
711
712     if i.multiSnap != nil {
713         for _, ino := range i.multiSnap.multiVersions {
714             ino.MarshalInodeValue(buff)
715         }
716     }
717
718     val = buff.Bytes()
719     i.RUnlock()
720     return
721 }
```

An attacker who can trigger the panic in a controlled manner has the potential to exploit this by locking a lot or all resources on the machine and thereby cause denial of service.

## QosCtrlManager

The Cubefs QoS manager's method for assigning QoS to clients, `assignClientsNewQos` is susceptible to a deadlock in case the manager has not enabled QoS. Below, the manager locks on line 692 and unlocks on line 722. On line 694, the manager will return if the QoS is not enabled:

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/master/limiter.go#L691-L735>

```
691 func (qosManager *QosCtrlManager) assignClientsNewQos(factorType uint32) {
692     qosManager.RLock()
693     if !qosManager.qosEnable {
694         return
695     }
696     serverLimit := qosManager.serverFactorLimitMap[factorType]
697     var bufferAllocated uint64
698
699
700     for _, cliInfoMgr := range qosManager.cliInfoMgrMap {
701         cliInfo := cliInfoMgr.Cli.FactorMap[factorType]
702         assignInfo := cliInfoMgr.Assign.FactorMap[factorType]
703
704         if cliInfo.Used+cliInfoMgr.Cli.FactorMap[factorType].Need == 0 {
705             assignInfo.UsedLimit = 0
706             assignInfo.UsedBuffer = 0
707         } else {
708             assignInfo.UsedLimit =
uint64(float64(cliInfo.Used+cliInfo.Need) * float64(1-serverLimit.LimitRate))
709             if serverLimit.Allocated != 0 {
710                 assignInfo.UsedBuffer =
uint64(float64(serverLimit.Buffer) * (float64(assignInfo.UsedLimit) /
float64(serverLimit.Allocated)) * 0.5)
711             }
712         }
713     }
714 }
```

```

713
714             if assignInfo.UsedBuffer > assignInfo.UsedLimit {
715                 assignInfo.UsedBuffer = assignInfo.UsedLimit
716             }
717         }
718
719         bufferAllocated += assignInfo.UsedBuffer
720     }
721
722     qosManager.RUnlock()
723
724     if serverLimit.Buffer > bufferAllocated {
725         serverLimit.Buffer -= bufferAllocated
726     } else {
727         serverLimit.Buffer = 0
728         log.LogWarnf("action[assignClientsNewQos] vol [%v] type [%v] clients
buffer [%v] and server buffer used up trigger flow limit overall",
729 qosManager.vol.Name, proto.QosTypeString(factorType),
bufferAllocated)
730     }
731
732     log.QosWriteDebugf("action[assignClientsNewQos] vol [%v] type [%v]
serverLimit buffer:[%v] used:[%v] need:[%v] total:[%v]",
733 qosManager.vol.Name, proto.QosTypeString(factorType),
serverLimit.Buffer, serverLimit.Allocated,
serverLimit.NeedAfterAlloc, serverLimit.Total)
734
735 }

```

An attacker cannot control whether Cubefs should proceed into this branch and return:

```

1         if !qosManager.qosEnable {
2             return
3         }

```

For an attacker to return on line 694 and thereby prevent Cubefs from unlocking the manager, they would need to know that the victims Cubefs deployment has disabled QoS and thereby cause Cubefs to invoke `assignClientsNewQos`.

## Block cache

The Block cache manager has a method for removing item keys from the cache to free up space, `freeSpace`. This method invokes a loop that ends when a counter, `cnt` reaches 500000. Each loop iteration performs the following steps: 1) The Block cache manager locks, 2) an item is deleted from the store, 3) the Block cache manager unlocks. This process is susceptible to a deadlock because the `freeSpace` method can exist between step 1 and 3, i.e. it is possible for `freeSpace` to lock the Block cache manager and return without unlocking it.

On line 390 the manager enters the for loop. Inside the loop, the manager locks on line 399 and unlocks on line 415. On line 403, `freeSpace` can return without unlocking the manager.

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/blockcache/bcache/manage.go#L379-L419>

```

379 func (bm *bcacheManager) freeSpace(store *DiskStore, free float32, files int64) {
380     var decreaseSpace int64
381     var decreaseCnt int
382
383     if free < store.freeLimit {
384         decreaseSpace = int64((store.freeLimit - free) *
(float32(store.capacity)))
385     }
386     if files > int64(store.limit) {
387         decreaseCnt = int(files - int64(store.limit))
388     }
389
390     cnt := 0
391     for {
392         if decreaseCnt <= 0 && decreaseSpace <= 0 {
393             break
394         }
395
396         if cnt > 500000 {
397             break
398         }

```

```

399         bm.Lock()
400
401         element := bm.lruList.Front()
402         if element == nil {
403             return
404         }
405         item := element.Value.(*cacheItem)
406
407         if err := store.remove(item.key); err == nil {
408             bm.lruList.Remove(element)
409             delete(bm.bcacheKeys, item.key)
410             decreaseSpace -= int64(item.size)
411             decreaseCnt--
412             cnt++
413         }
414
415         bm.Unlock()
416         log.Debugf("remove %v from cache", item.key)
417     }
418 }
419 }

```

## Volume manager

When Cubefs's Volume Manager applies an update to a volume unit, it does so with `applyAdminUpdateVolumeUnit`. `applyAdminUpdateVolumeUnit` gets the disk info with a call to the disk managers `GetDiskInfo`. If this call fails, `applyAdminUpdateVolumeUnit` returns the error. Before getting the disk info, `applyAdminUpdateVolumeUnit` puts a lock on the volume that is being modified, and `applyAdminUpdateVolumeUnit` will not release that lock if the call to `GetDiskInfo` fails. In other words, if the call to `GetDiskInfo` fails, the lock will not be released. The parameter to `GetDiskInfo` is passed directly from a parameter to `applyAdminUpdateVolumeUnit`.

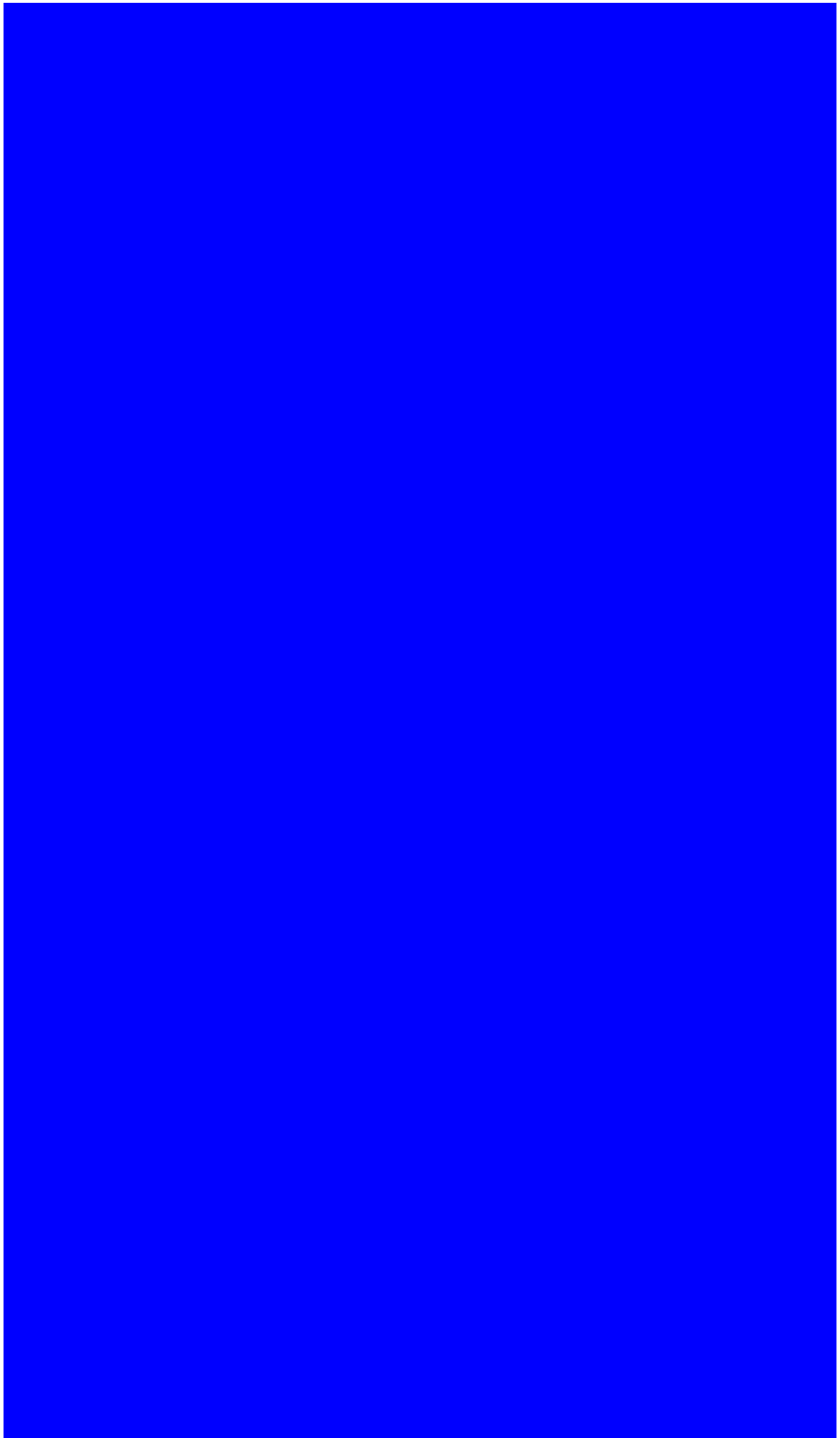
`applyAdminUpdateVolumeUnit` locks the volume on line 691 and unlocks it again on line 710. On line 701, `applyAdminUpdateVolumeUnit` returns without unlocking the volume.

<https://github.com/cubefs/cubefs/blob/46cb4d149c45f1ad7b40381b5a2a20bd6d599e25/blobstore/clustermgr/volumemgr/volumemgr.go#L675-L711>

```

675 func (v *VolumeMgr) applyAdminUpdateVolumeUnit(ctx context.Context, unitInfo
    *cm.AdminUpdateUnitArgs) error {
676     span := trace.SpanFromContextSafe(ctx)
677     vol := v.all.getVol(unitInfo.Vuid.Vid())
678     if vol == nil {
679         span.Errorf("apply admin update volume unit,vid %d not exist",
unitInfo.Vuid.Vid())
680         return ErrVolumeNotExist
681     }
682     index := unitInfo.Vuid.Index()
683     vol.lock.RLock()
684     if int(index) >= len(vol.vUnits) {
685         span.Errorf("apply admin update volume unit,index:%d over vuids
length ", index)
686         vol.lock.RUnlock()
687         return ErrVolumeUnitNotExist
688     }
689     vol.lock.RUnlock()
690
691     vol.lock.Lock()
692     if proto.IsValidEpoch(unitInfo.Epoch) {
693         vol.vUnits[index].epoch = unitInfo.Epoch
694         vol.vUnits[index].vuInfo.Vuid =
proto.EncodeVuid(vol.vUnits[index].vuidPrefix, unitInfo.Epoch)
695     }
696     if proto.IsValidEpoch(unitInfo.NextEpoch) {
697         vol.vUnits[index].nextEpoch = unitInfo.NextEpoch
698     }
699     diskInfo, err := v.diskMgr.GetDiskInfo(ctx, unitInfo.DiskID)
700     if err != nil {
701         return err
702     }
703     vol.vUnits[index].vuInfo.DiskID = diskInfo.DiskID
704     vol.vUnits[index].vuInfo.Host = diskInfo.Host
705     vol.vUnits[index].vuInfo.Compacting = unitInfo.Compacting
706
707     unitRecord := vol.vUnits[index].ToVolumeUnitRecord()
708     err = v.volumeTbl.PutVolumeUnit(unitInfo.Vuid.VuidPrefix(), unitRecord)
709     vol.lock.Unlock()
710     return err

```



```
319
320     shards.mu.Lock()
321     shards.shards[bid].ok = true
322     shards.mu.Unlock()
323     return nil
324 }
```

# Possible nil-dereference from unmarshalling double pointer

Severity:	Low
Status:	Fixed
Id:	ADA-CUBEFS-ASBDVGA
Component:	ObjectNode

Unmarshalling into a double-pointer can result in nil-pointer dereference if the raw bytes are `NULL`.

CubeFS has a case that would trigger a nil-pointer dereference and crash the CubeFS ObjectNode:

```
https://github.com/cubefs/cubefs/blob/45442918591d25e7ab555469df384df468df5dbc/objectnode/acl_api.go#L186-L201

186 func getObjectACL(vol *Volume, path string, needDefault bool) (*AccessControlPolicy,
187     error) {
188     xAttr, err := vol.GetXAttr(path, XAttrKeyOSSACL)
189     if err != nil || xAttr == nil {
190         return nil, err
191     }
192     var acp *AccessControlPolicy
193     data := xAttr.Get(XAttrKeyOSSACL)
194     if len(data) > 0 {
195         if err = json.Unmarshal(data, &acp); err != nil {
196             err = xml.Unmarshal(data, &acp)
197         }
198     } else if needDefault {
199         acp = CreateDefaultACL(vol.owner)
200     }
201     return acp, err
202 }
```

On line 194, `getObjectACL` unmarshals into a double pointer. `acp` is declared on line 191 as a pointer and is referenced with a pointer on line 194. If `data` on line 194 is the byte sequence equal to `NULL`, `acp` will be `nil` on line 194 and return `nil, nil`.

This behaviour will trigger a nil-pointer dereference on 145 in the below code snippet:  
[https://github.com/cubefs/cubefs/blob/6a0d5fa45a77ff20c752fa9e44738bf5d86c84bd/objectnode/acl\\_handler.go#L110-L153](https://github.com/cubefs/cubefs/blob/6a0d5fa45a77ff20c752fa9e44738bf5d86c84bd/objectnode/acl_handler.go#L110-L153)

```
1 func (o *ObjectNode) getObjectACLHandler(w http.ResponseWriter, r *http.Request) {
2     var (
3         err error
4         erc *ErrorCode
5     )
6     defer func() {
7         o.errorResponse(w, r, err, erc)
8     }()
9
10    param := ParseRequestParam(r)
11    if param.Bucket() == "" {
12        erc = InvalidBucketName
13        return
14    }
15    if param.Object() == "" {
16        erc = InvalidKey
17        return
18    }
19
20    var vol *Volume
21    if vol, err = o.getVol(param.bucket); err != nil {
22        log.LogErrorf("getObjectACLHandler: load volume fail: requestID(%v)
23        volume(%v) err(%v)",
24            GetRequestID(r), param.bucket, err)
25    }
```

```

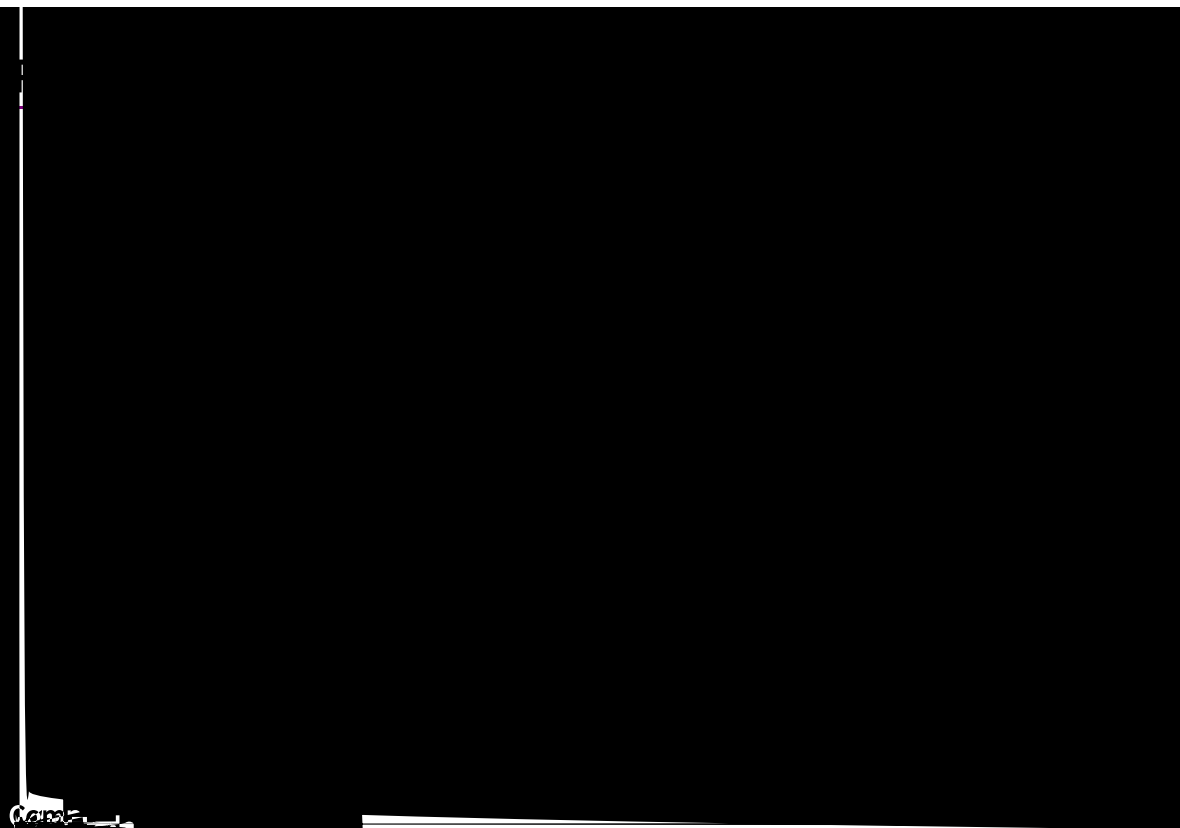
24         return
25     }
26     var acl *AccessControlPolicy
27     if acl, err = getObjectACL(vol, param.object, true); err != nil {
28         log.Errorf("getObjectACLHandler: get acl fail: requestID(%v)
volume(%v) path(%v) err(%v)",
29                     GetRequestID(r), param.bucket, param.object, err)
30         if err == syscall.ENOENT {
31             erc = NoSuchKey
32         }
33         return
34     }
35     var data []byte
36     if data, err = acl.XmlMarshal(); err != nil {
37         log.Errorf("getObjectACLHandler: xml marshal fail: requestID(%v)
volume(%v) path(%v) acl(%+v) err(%v)",
38                     GetRequestID(r), param.bucket, param.object, acl, err)
39         return
40     }
41
42     writeSuccessResponseXML(w, data)
43     return
44 }

```

On line 136 `getObjectACLHandler` invokes `getObjectACL`. If this returns `nil, nil`, then a nil-pointer dereference will be triggered on line 145.

### Mitigation

Unmarshal into a single pointer instead of a double pointer.



https://github.com/cubefs/cubefs/blob/9c9f0bad65fc4a904160ff22cdaba2d9d6becd7c/authr  
https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/master/http\_servego#L37-L64

```
37 func (m *Server) startHTTPService(modulename string, cfg *config.Config) {
38     router := NewRouter().SkipClean(true)
39     m.registerAPIRoutes(router)
40     m.registerAPIMiddleware(router)
41     if m.cluster.authenticate {
42         m.registerAuthenticationMiddleware(router)
43     }
44     exporter.InitWithRouter(modulename, cfg, router, m.port)
45     addr := fmt.Sprintf(":%s", m.port)
46     if m.listen {
47         addr = fmt.Sprintf("%s:%s", m.ip, m.port)
48     }
49     startHTTPService := func() {
50         server := &http.Server{
51             Addr: addr,
52             Handler: router,
53         }
54         var serveAPI = func() {
55             if err := server.ListenAndServe(); err != nil {
56                 log.Errorf("serveAPI: serve http server failed: err(%v)", err)
57                 return
58             }
59         }
60         go serveAPI()
61     }
```

Because of the Master server Slowloris issue is that it does not declare a timeout. On line 47, the `startHTTPService` declares the HTTP Server with address and handler but does not declare a timeout.

p x B N G D  
BB BB  
k bB N  
DQ



```

62     m.apiServer = server
63     return
64 }

```

The server does not have a timeout at all because the server has specified neither `ReadTimeout` nor `ReadHeaderTimeout`. This grants an attacker ample flexibility and possibilities for getting the server to hang. Note that the server also does not have write timeouts, which adds to an attacker's possibilities of triggering this.

Below, we enumerate all other HTTP servers that do not specify timeouts. We do not include tests and examples.

<https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/blobstore/cmd/cmd.go#L135-L144>

```

135     if mod.graceful {
136         programEntry := func(state *graceful.State) {
137             router, handlers := mod.Setup()
138
139             httpServer := &http.Server{
140                 Addr:    cfg.BindAddr,
141                 Handler: reorderMiddlewareHandlers(router, lh, cfg.BindAddr,
142                                     cfg.Auth, handlers),
143             }
144             log.Infof("server is running at:", cfg.BindAddr)

```

<https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/blobstore/cmd/cmd.go#L171-L174>

```

171     httpServer := &http.Server{
172         Addr:    cfg.BindAddr,
173         Handler: reorderMiddlewareHandlers(router, lh, cfg.BindAddr, cfg.Auth,
174                                     handlers),

```

<https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/blobstore/common/raftserver/transport.go#L70-L73>

```

70     tr.httpSvr = &http.Server{
71         Addr:    fmt.Sprintf(":%d", port),
72         Handler: router,
73     }

```

<https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/blobstore/common/consul/consul.go#L216-L227>

```

216     srv = &http.Server{}
217     srv.Addr = ln.Addr().String()
218     port = ln.Addr().(*net.TCPAddr).Port
219     log.Infof("start health check server on: ", srv.Addr)
220     http.HandleFunc(patten, healthCheck)
221     go func() {
222         httpError := srv.Serve(ln.(*net.TCPListener))
223         if httpError != nil && httpError != http.ErrServerClosed {
224             log.Fatalf("health server HTTP error: ", httpError)
225         }
226         log.Infof("health check server exit")
227     }()

```

<https://github.com/cubefs/cubefs/blob/5ab518b3598ee99a74b333d0d2abc80739bbae4d/objectnode/server.go#L463-L473>

```

463     var server = &http.Server{
464         Addr:    ":" + o.listen,
465         Handler: router,
466     }
467
468     go func() {
469         if err = server.ListenAndServe(); err != nil {
470             log.LogErrorf("startMuxRestAPI: start http server fail, err(%v)", err)
471             return
472         }
473     }()

```

---

### **Mitigation**

Add timeouts when declaring the servers.

## Releases are not signed

---

Severity:	Moderate
Status:	Fixed
Id:	ADA-CUBEFS-NJb32hjJBN
Component:	CubeFS

CubeFS releases are not signed, with keys available alongside releases. Signing releases and allowing consumers to verify them mitigates supply-chain risks.

A tool like Cosign makes the signing process easy and low-effort and keeps the overhead for consumers low to verify signatures. These signatures should be available with releases.

### Mitigation

Release signing by way of Cosign can be adopted by way of the official Cosign Github Action: <https://github.com/marketplace/actions/cosign-installer>.

# Security Disclosure Email Does Not Work

---

Severity:

Low

# Timing attack can leak user passwords

---

