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

Pentest-Report ChubaoFS Pentest 08.-09.2020

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. D. Weißer, MSc. F. Fäßler, MSc. R. Peraglie

Index

Introduction
Scope
Identified Vulnerabilities
CFS-01-003 WP1: Insecure SHA1 password-hashing (Low)
CFS-01-004 WP1: Linux file permissions ineffective for ACL (High)
CFS-01-005 WP1: Missing HMAC leads to CBC padding oracle (Medium)
CFS-01-007 WP1: No brute-force protection on time-unsafe comparisons (Low)
CFS-01-008 WP1: Unencrypted raw TCP traffic to Meta- and DataNode (High)
CFS-01-009 WP1: Unauthenticated raw TCP traffic to Meta- and DataNode (High)
CFS-01-010 WP1: Lack of TCP traffic message replay protection (Medium)
CFS-01-011 WP1: Rogue Meta- and DataNodes possible due to lack of ACL (High)
CFS-01-015 WP1: API freely discloses all user-secrets (Critical)
CFS-01-016 WP2: Default Docker deployment insecure on public hosts (High)
CFS-01-022 WP1: HTTP clear-text ObjectNode REST API exposed (High)
CFS-01-024 WP2: Bypassing Skip-Owner-Validation header authentication (Medium)
Miscellaneous Issues
CFS-01-001 WP1: Usage of math/rand within crypto-utils and utils (Info)
CFS-01-002 WP1: TLS version not enforced for AuthNode HTTP server (Low)
CFS-01-006 WP1: Password hashes can be used to authenticate (Low)
CFS-01-012 WP1: HTTP parameter pollution in HTTP clients (Medium)
CFS-01-013 WP1: Unsalted MD5 authKey-Computation in ObjectNode (Low)
CFS-01-014 WP2: Lack of password complexity in MasterNode (Low)
CFS-01-017 WP2: Docker deployment stores client.json as world-readable (Medium)
CFS-01-018 WP1: Docker deployment logs credentials as world-readable (Medium)
CFS-01-019 WP1: Folders can be moved into their own child folders (Low)
CFS-01-020 WP1: Missing filename-validation allows folder corruption (Low)
CFS-01-021 WP1: Debugging endpoint /debug/pprof exposed (Info)
CFS-01-023 WP1: Build system lacks stack canaries, PIE and FORTIFY (Medium)

CFS-01-025 WP1: Outdated vulnerable bzip2 dependency for ARM64 build (Info)
CFS-01-026 WP2: cfs-server processes running with root privileges (Medium)
CFS-01-027 WP1: Potential path traversal in MetaNodes (Low)
CFS-01-028 WP1: Insecure ObjectNode policy-checking behavior (Medium)
Conclusions

Introduction

“ChubaoFS has been commonly used as the underlying storage infrastructure for online applications, database or data processing services and machine learning jobs orchestrated by Kubernetes. An advantage of doing so is to separate storage from compute - one can scale up or down based on the workload and independent of the other, providing total flexibility in matching resources to the actual storage and compute capacity required at any given time.”

From https://github.com/chubaofs/chubaofs

This report describes the results of a thorough and broadly scoped security assessment of the ChubaoFS software, which is a cloud-native storage system with advertised POSIX- and S3-compatibility/ The work was requested by CNCF and executed by Cure53 in late August 2020, precisely in calendar weeks CW34 and CW35. Twenty-eight security-relevant problems were observed by Cure53 on the ChubaoFS-delineated scope.

In terms of resources, six senior testers were involved in this exercise after being selected on the basis of their skills and expertise best-matching the requirements and needs of ChubaoFS. The testing team spent 32 person-days on this project. To ensure that all key aspects are covered to an expected degree, two work packages (WPs) were drafted. In WP1, Cure53 completed a security review and audited the source code of the ChubaoFS in version v2.1.0. Conversely, penetration tests centered on the production-like ChubaoFS v2.1.0. deployment took place during WP2.

Following best practices of CNCF-related Cure53 work, the chosen methodology here was white-box. This was especially dictated by the ChubaoFS source code being openly available on GitHub. In this context, Cure53 was given access to a fully set up testing instance prepared by the ChubaoFS team. The testers were further supplied with additional test-supporting material and documentation. All in all, the preparations were all done very well by the in-house team.

The test started on time and progressed efficiently. A dedicated Slack workspace of ChubaoFS was used for communications, with relevant members of the Cure53 team

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

joining the exchanges. Ongoing feedback has been shared by the involved teams during the tests and audits. Communications were helpful and fluent, Cure53 was able to ask questions and get quick answers, report status updates and keep the ChubaoFS team updated in regards to the progress and findings spotted over the course of this exercise. As a result of the proper setup, Cure53 could focus on reaching very good coverage over the test-targets.

It has already been noted above that Cure53 identified twenty-eight security-relevant issues. Twelve items were classified to be security vulnerabilities of varying severity ratings and the remaining sixteen discoveries represent general weaknesses with lower exploitation potential or impact. One issue was given a *Critical* score, while further six problems should be considered *High* risks. Quite clearly, this is a rather extensive number of findings, especially compared to the results of many other CNCF-related projects that Cure53 completed over the years. Moreover, this outcome is exacerbated by the severity and significance of many flaws to which ChubaoFS has been proven vulnerable. Foreshadowing the conclusions, this leaves the impression of the ChubaoFS complex not being up-to-par when it comes to modern security standards.

In the following sections, the report will first shed light on the scope and key test parameters. Next, all findings will be discussed in a chronological order alongside technical descriptions, as well as PoC and mitigation advice when applicable. Finally, the report will close with broader conclusions about this August 2020 project. Cure53 elaborates on the general impressions pertaining to the ChubaoFS complex and reiterates the verdict based on the testing team’s observations and collected evidence. Tailored hardening recommendations and advice on moving forward are also incorporated into the final section.

Scope

- **Penetration Tests & Security Reviews against ChubaoFS v2.1.0**
    - **WP1**: Security review & source code audit against *"ChubaoFS" v2.1.0*
        - https://github.com/chubaofs/chubaofs
            - *commit 5330cf5b250562c29541c20a675f33d50affaea0*
    - **WP2**: Penetration test against prod-like *"ChubaoFS" v2.1.0* deployment
        - A testing environment was provided.
- **Sources were available as OSS**
- **Test-supporting material was shared with Cure53**
- **A testing environment was made available for Cure53**
    - The test setup used an AWS EC2 instance with 8 CPU, 32GB memory and 80GB EBS storage.
    - The following components were launched in a POD to build a ChubaoFS cluster for functional testing:
        - 3 containers running Master
        - 4 containers running MetaNode
        - 4 containers running DataNode
        - 3 containers running ObjectNode
        - 1 container running Console
        - 1 container running *nginx*
    - The *nginx* container has been set up to map port 80 to the host and has reverse-proxied the S3-compatible object storage interface provided by *ObjectNode*, the web service provided by the *Console*, and the management API provided by the *Master* node.
    - In this environment, a user named *ltptest* has been created in advance, and a volume named *ltptest* with a capacity of 30GB has been created for this user.

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

CFS-01-003 WP1: Insecure SHA1 password-hashing *(Low)*

During an audit of the CFS source code, it was identified that the function *encodingPassword()* inside *master/user.go* is using SHA1 for creating a password hash of the user's credentials. Since the SHA1 hashing algorithm is no longer considered secure and collision-free, it is important to replace SHA1 with another, more robust password hashing algorithm.

Affected File:
master/user.go

Affected Code:
```
package master

import (
	"crypto/sha1"
[...]

func encodingPassword(s string) string {
	t := sha1.New()
	io.WriteString(t, s)
	return hex.EncodeToString(t.Sum(nil))
}
```

Instead of using SHA1 for hashing the user-credentials, Cure53 recommends to use other password hashing algorithms, such as *argon2*[1] or *scrypt*[2].

[1] https://godoc.org/golang.org/x/crypto/argon2
[2] https://godoc.org/golang.org/x/crypto/scrypt

CFS-01-004 WP1: Linux file permissions ineffective for ACL (*High*)

It was discovered that the locally mounted filesystem does not enforce access control, therefore making it possible for unprivileged local users to edit any file independent of owner or permission flags. This poses a great risk as local attackers could access any data on the drive or even escalate privileges and gain persistence in some scenarios. The problem is described by the following *shell* excerpt.

Missing access control:
```
$ ls -al
total 0
-rw------- 1 root root 5 Aug 18 16:18 catz
$ id
uid=1000(bla) gid=1000(bla) groups=1000(bla)
$ cat catz
meow
```

Under normal circumstances, the user *bla* should not be able to read the *root* owned file due to the permissions of the file. However, as the fuse client lacks the appropriate flags that would let the Linux kernel handle the permissions, any user can read/write any file. It is recommended to add the *default_permissions*[3] flag to the *mount* options of the fuse client. This ensures that privileges are handled by the operating system and requires no additional implementation in the client itself.

CFS-01-005 WP1: Missing HMAC leads to *CBC* padding oracle (*Medium*)

The *authnode* is utilized to issue tickets used for authentication. Some parts of the communication with the service is encrypted using *AES* in *CBC* mode. An additional checksum serves as an integrity check. However, this construct is insecure and does not protect the data at all as a *CBC* padding oracle[4] can be employed to decrypt and encrypt arbitrary messages. Shown below is the code responsible for decrypting messages. The data is first decrypted whereas the decryption function already removes the padding. Then the length and the checksum of the plain-data are verified. If something goes wrong, an arrow is returned.

Affected File:
/util/cryptoutil/cryptoutil.go

Affected Code:
```
func unpad(src []byte) []byte {
	length := len(src)
	unpadding := int(src[length-1])
```

[3] https://www.kernel.org/doc/Documentation/filesystems/fuse.txt
[4] https://en.wikipedia.org/wiki/Padding_oracle_attack

```
        return src[:(length - unpadding)]
}

func DecodeMessage(message string, key []byte) (plaintext []byte, err error) {
        [...]

        if decodedText, err = AesDecryptCBC(key, cipher); err != nil {
                return
        }

        if len(decodedText) <= MessageMetaDataSize {
                err = fmt.Errorf("invalid json format with size [%d] less than
message meta data size", len(decodedText))
                return
        }

        msgChecksum := make([]byte, CheckSumSize)
        copy(msgChecksum,
decodedText[CheckSumOffset:CheckSumOffset+CheckSumSize])

        // calculate checksum
        filltext := bytes.Repeat([]byte{byte(0)}, CheckSumSize)
        copy(decodedText[CheckSumOffset:], filltext[:])
        newChecksum := md5.Sum(decodedText)

        // verify checksum
        if bytes.Compare(msgChecksum, newChecksum[:]) != 0 {
                err = fmt.Errorf("checksum not match")
        }

        plaintext = decodedText[MessageOffset:]

        //fmt.Printf("DecodeMessage CBC: %s\n", plaintext)
        return
}
```

It was further observed that the padding function does not verify if a proper padding was used. Instead, it just determines how many bytes to remove based on the last character in the data. This in combination with the key-less checksum makes a *CBC* attack fairly easy.

It is recommended to avoid *CBC* and switch to an authenticated encryption mode such as *AES-GCM*. This mode follows the *encrypt-then-authenticate* principle and eliminates the risk of data exfiltration via padding oracles.

CFS-01-007 WP1: No brute-force protection on time-unsafe comparisons *(Low)*

It was found that the authentication makes use of the time-unsafe comparison when verifying the password of the user. The string comparison in Go lang is an algorithm that is linearly time-variant to the equivalence of the input strings. In this specific case, the more the user-input matches the password, the greater the runtime of the comparison. This could be abused by attackers to send a very large number of requests, rendering the minimal time differences measurable. From there, the password of a targeted user could be brute-forced *character-by-character*.

Affected File:
master/gapi_user.go

Affected Code:
```
func (s *UserService) validatePassword(ctx context.Context, args struct {
    UserID   string
    Password string
}) (*proto.UserInfo, error) {
    ui, err := s.user.getUserInfo(args.UserID)
    if err != nil {
        return nil, err
    }

    ak, err := s.user.getAKUser(ui.AccessKey)
    if err != nil {
        return nil, err
    }

    if ak.Password != args.Password {
        log.LogWarnf("user:[%s] login pass word has err", args.UserID)
        return nil, fmt.Errorf("user or password has err")
    }
    return ui, nil
}
```

For all string comparisons that contain sensitive information, it is recommended to use time-safe comparisons, such as those implemented by the function called *ConstantTimeCompare()* of the *crypto/subtle* package. By doing so, the string comparison runs with a time that is constant for strings of the same length. This will prevent attackers from using time as a side-channel, thus mitigating the risk of sensitive information being extracted.

CFS-01-008 WP1: Unencrypted raw TCP traffic to *Meta*- and *DataNode* (*High*)

During an audit of the CFS source code, it was found that every *Meta*-and *DataNode* is starting up a raw TCP server that is handling incoming messages. The communication to the referred TCP services is performed unencrypted and in clear-text, meaning an attacker potentially capable of intercepting network communication can eavesdrop on or tamper with transmitted messages.

Affected File:
datanode/server.go

Affected Code:
```
func (s *DataNode) startTCPService() (err error) {
       [...]
       l, err := net.Listen(NetworkProtocol, addr)
       [...]
}
```

Affected File:
metanode/server.go

Affected Code:
```
func (m *MetaNode) startServer() (err error) {
       [...]
       ln, err := net.Listen("tcp", ":"+m.listen)
       [...]
}
```

Encrypting communication whenever possible is considered state-of-the-art and is highly recommended for any communication from / to the *Meta*-and *DataNode*. The *tls* package[5] within Go lang can be used for adding TLS encryption to the listening TCP services.

CFS-01-009 WP1: Unauthenticated raw TCP traffic to *Meta*- and *DataNode* (*High*)

During an audit of the CFS source code, it was found that every *MetaNode* and *DataNode* starts up a raw TCP server for handling incoming messages. The communication to the referred TCP services is performed unauthenticated, meaning an attacker with network connectivity to the *Meta*- and *DataNodes* can send arbitrary messages / commands.

[5] https://golang.org/pkg/crypto/tls/

The *startServer()* (*MetaNode)* and *startTCPService()* (*DataNode*) functions are starting the actual TCP listening sockets. Incoming messages / packets for the *MetaNode* will then be handled by the following sequence of function calls:

serveConn() > *m.handlePacket()* > *m.metadataManager.HandleMetadataOperation()*.

Incoming messages / packets for the *DataNode* will then be handled by the following function trace:

serveConn() > *packetProcessor.ServerConn()* > *rp.readPkgAndPrepare()* > *rp.putToBeProcess()*.

The function *putToBeProcess()* puts incoming packets to the *to-be-processed* channel.

The following depicts an example for the *MetaNode* service. Inside *m.metadataManager.HandleMetadataOperation()*, a switch case statement processes received messages. Depending on the submitted opCode, it will continue invoking the respective handler-function. For example the opCode for setting an extended attribute *proto.OpMetaSetXAttr* invokes the function of *opMetaSetXAttr()*, which in turn ends up invoking *mp.SetXAttr()* without any authorization checks. This ensures that the caller is allowed to set an extended attribute.

It is important to emphasize that both TCP servers for the *Meta*- and *DataNodes* are vulnerable against unauthenticated function calls and lack any sort of access control.

Affected File:
datanode/server.go

Affected Code:
```
func (s *DataNode) startTCPService() (err error) {
	[...]
	l, err := net.Listen(NetworkProtocol, addr)
	[...]
	go func(ln net.Listener) {
		for {
			conn, err := ln.Accept()
			[...]
			go s.serveConn(conn)
			[...]
}
```

Affected File:
metanode/server.go

Affected Code:

```
func (m *MetaNode) startServer() (err error) {
	[...]
	ln, err := net.Listen("tcp", ":"+m.listen)
	[...]
	go func(stopC chan uint8) {
		defer ln.Close()
		for {
			conn, err := ln.Accept()
			[...]
			go m.serveConn(conn, stopC)
			[...]
}
```

Cure53 wants to stress the importance of adding authentication and proper access control checks to the TCP servers of the *Meta*- and *DataNode* in order to ensure that unauthenticated communication stops being possible.

CFS-01-010 WP1: Lack of TCP traffic message replay protection *(Medium)*

During an audit of the CFS source code, it was identified that the packet structure definition and message processing of the *Meta*- and *DataNode* TCP server is not protecting against potential message replay attacks. An attacker could capture previously transmitted messages and replay/inject them onto the wire, causing potential inconsistencies or Denial-of-Service situations within the CFS cluster.

Affected File:
proto/packet.go

Affected Code:

```
// Packet defines the packet structure.
type Packet struct {
	Magic              uint8
	ExtentType         uint8
	Opcode             uint8
	ResultCode         uint8
	RemainingFollowers uint8
	CRC                uint32
	Size               uint32
	ArgLen             uint32
	KernelOffset       uint64
	PartitionID        uint64
	ExtentID           uint64
	ExtentOffset       int64
	ReqID              int64
```

```
        Arg                 []byte
        Data                []byte
        StartT              int64
        mesg                string
        HasPrepare          bool
}
```

The protocol should incorporate some sort of replay protection, ensuring that injecting and replaying messages is not possible. Such a replay protection could consist of the sender's source IP address and a timestamp information where the receiver rejects messages that are older than a predefined time window.

When using time information for determination of message replay, it is crucial to properly synchronize the time of all *Meta*- and *DataNodes*. Considering the entire communication to the *Meta*- and *DataNode* is additionally encrypted, an attacker would have no chance to tamper with any of the information, rejecting replayed messages.

CFS-01-011 WP1: Rogue *Meta-* and *DataNodes* possible due to lack of ACL (*High*)

During an audit of the CFS source code, it was identified that the registration of new *Meta*- and *DataNodes* is performed without any form of access control mechanism in place. The sole parameters required to add new nodes are:

- A combination of the source IP address and port number.
- The zone name.

An attacker / malicious user could abuse the lack of access control when adding new *Meta*- and *DataNodes* to potentially inject rogue *Meta*- and *DataNodes*.

Affected File:
datanode/server.go

Affected Code:
```
// registers the data node on the master to report the information such as
IsIPV4 address.
// The startup of a data node will be blocked until the registration succeeds.
func (s *DataNode) register(cfg *config.Config) {
        [...]
        // get the IsIPV4 address, cluster ID and node ID from the master
        for {
                [...]
                if nodeID, err = MasterClient.NodeAPI().AddDataNode(
                        fmt.Sprintf("%s:%v", LocalIP, s.port), s.zoneName);
                        err != nil {
                        log.LogErrorf("action[registerToMaster]
                                        cannot register this node to master[%v] err(%v).",
                                        masterAddr, err)
                                [...]
                        }
                        [...]
}
```

Affected File:
metanode/metanode.go

Affected Code:
```
func (m *MetaNode) register() (err error) {
        step := 0
        var nodeAddress string
        for {
                [...]
                nodeAddress = m.localAddr + ":" + m.listen
                [...]
        }
        [...]
        if nodeID, err = masterClient.NodeAPI().AddMetaNode(nodeAddress,
                m.zoneName); err != nil {
                log.LogErrorf("register: register to master fail:
                        address(%v) err(%s)", nodeAddress, err)
                [...]
        }
        [...]
}
```

It is recommended to ensure that only authorized *Meta-* and *DataNodes* are able to join the network of nodes.

CFS-01-015 WP1: API freely discloses all user-secrets (*Critical*)

Exploring the exposed functionality of CFS revealed that the API to list all users, and even the *cli* tool, discloses the *authKey* as well as *secretKey* of every user. This renders the role-based access control useless as any user can simply get the credentials of the admin.

PoC:
```
root@075f2e931570:/go# cfs-cli user list
ID          TYPE      ACCESS KEY            SECRET KEY
root        Root      cD7iHA2ZVOAUXmSb      mCGWXZFF8KiGtyxzd3baW7WbMTCybBeF
ltptest     Normal    39bEF4RrAQgMj6RV      TRL6o3JL16YOqvZGIohBDFTHZDEcFsyd
root@075f2e931570:/go#
root@075f2e931570:/go# curl "192.168.0.11:17010/user/list"
```

```
{"code":0,"msg":"success","data":
[{"user_id":"root","access_key":"cD7iHA2ZVOAUXmSb","secret_key":"mCGWXZFF8KiGtyx
zd3baW7WbMTCybBeF","policy":{"own_vols":["asd"],"authorized_vols":
{}},"user_type":1,"create_time":"2020-08-17
10:02:45","description":"","EMPTY":false},
{"user_id":"ltptest","access_key":"39bEF4RrAQgMj6RV","secret_key":"TRL6o3JL16YOq
vZGIohBDFTHZDEcFsyd","policy":{"own_vols":["ltptest"],"authorized_vols":
{}},"user_type":3,"create_time":"2020-08-17
10:02:55","description":"","EMPTY":false}]}
```

The credentials should never be exposed by the API. This completely renders all volume permissions useless and allows any user to gain access to any other user's setup. Especially in a multi-tenant setup where different customers use the same deployment, this is a very serious issue.

CFS-01-016 WP2: Default Docker deployment insecure on public hosts *(High)*

The default Docker deployment exposes all services on the main network interface. The test deployment provided by the developers was set up on Amazon Virtual Private Cloud, which means the machine is luckily not exposed to the Internet. However, if ChubaoFS is set up based on the instructions on GitHub on a dedicated server not inside a private network, all services and APIs become publicly accessible.

PoC:
In this PoC, ChubaoFS was set up on a VPS using the provided script of *docker/run_docker.sh*. The server in this case had the IP address of 142.93.100.176. The following output shows that one *MasterNode* is listening on public port 32958.

```
server$ docker ps
[...]
e6ac5197cf5f        chubaofs/cfs-base:1.1   "/bin/sh /cfs/script…"   3 days ago
Up 3 days           0.0.0.0:32992->5901/tcp, 0.0.0.0:32982->5902/tcp,
0.0.0.0:32971->9500/tcp, 0.0.0.0:32958->17010/tcp, 0.0.0.0:32948->17020/tcp
docker_master2_1
[...]
```

The following *curl* command is executed on a different machine using the public IP and the port above to show that a remote attacker can easily extract all the credentials. This can then be used by the attacker to mount any of the volumes and extract all private data.

```
user$ curl -v 142.93.100.176:32958/user/list
{"code":0,"msg":"success","data":
[{"user_id":"root","access_key":"cD7iHA2ZVOAUXmSb","secret_key":"mCGWXZFF8KiGtyx
zd3baW7WbMTCybBeF","policy":{"own_vols":["asd"],"authorized_vols":
```

```
{}},"user_type":1,"create_time":"2020-08-17
10:02:45","description":"","EMPTY":false},
{"user_id":"ltptest","access_key":"39bEF4RrAQgMj6RV","secret_key":"TRL6o3JL16YOq
vZGIohBDFTHZDEcFsyd","policy":{"own_vols":["ltptest"],"authorized_vols":
{}},"user_type":3,"create_time":"2020-08-17
10:02:55","description":"","EMPTY":false}]
```

Even in a private cloud deployment, it leaves ChubaoFS exploitable once a single host inside the network gets compromised. It is recommended to not expose the ports by default, especially because the Docker setup uses an internal network anyway. Without any form of meaningful authentication and access control reported in other issues, ChubaoFS must not be deployed on any publicly reachable host.

CFS-01-022 WP1: HTTP clear-text *ObjectNode REST* API exposed (*High*)

During an audit of the CFS source code, it was found that the *ObjectNode* service is starting up an HTTP *REST* API for handling incoming messages. The listening port of this *REST* API can be configured by the user through the configuration file, however, a user cannot enforce the HTTP service to be protected by TLS. It has to be noted that the example configuration of an *ObjectNode,* stored within *docker/conf/objectnode.json* of the official CFS repo, uses the *enableHTTPS* configuration option within the JSON file. However, when looking at the respective source code responsible for processing / parsing the configuration, *enableHTTPS* is not processed and, therefore, never gets used.

As a result, the communication to this *REST* API is performed unencrypted and in clear-text, meaning an attacker who is potentially capable of intercepting network communication can eavesdrop on or tamper with transmitted messages.

Affected File:
objectnode/server.go

Affected Code:
```
func (o *ObjectNode) startMuxRestAPI() (err error) {
	[...]
	var server = &http.Server{
		Addr:    ":" + o.listen,
		Handler: router,
	}

	go func() {
		if err = server.ListenAndServe(); err != nil {
			[...]
	o.httpServer = server
	return
}
```

Cure53 wants to stress the importance of encrypting communication whenever possible. HTTP services can, for example, be implemented by using security-aware middleware, such as the *Secure*[6] library, which offers additional security features and improves the overall security posture of the HTTP service.

CFS-01-024 WP2: Bypassing *Skip-Owner-Validation* header authent. (*Medium*)

It was found that the *volume manager* takes an authentication key of the owner to return details of the volume. The information includes the access and secret key for the volume. Not only is the authentication check weak due to using the *md5* hash of the owner, but it can also be bypassed with the *Skip-Owner-Validation* HTTP header. The exposed information allows anybody to mount and access the data of this volume.

Example:
The following request attempts to get the information of the volume named "*asd*", but fails due to the missing authentication key.

```
# curl "192.168.0.11:17010/client/vol?name=asd"

[operate_util.go 174] parameter authKey not found
```

By providing the *Skip-Owner-Validation* HTTP header, no key is required and an attacker gains access to the required keys to access the data of this volume.

```
# curl "192.168.0.11:17010/client/vol?name=asd" -H "Skip-Owner-Validation: 1"

{"code":0,"msg":"success","data":
{"Name":"asd","Owner":"root","Status":0,"FollowerRead":true,"MetaPartitions":
[{"PartitionID":4,"Start":0,"End":16777216,"MaxInodeID":1,"InodeCount":1,"Dentry
Count":0,"IsRecover":false,"Members":
["192.168.0.24:17210","192.168.0.22:17210","192.168.0.21:17210"],
[...]
,"IsRecover":false},{"PartitionID":11,"Status":2,"ReplicaNum":3,"Hosts":
["192.168.0.31:17310","192.168.0.34:17310","192.168.0.33:17310"],"LeaderAddr":"1
92.168.0.34:17310","Epoch":0,"IsRecover":false}],"OSSSecure":
{"AccessKey":"GjAZYkDISiR2tLkF","SecretKey":"yPS1zr6KbG8u9TLyvyJYXtzEdokIs6Ar"},
"CreateTime":1597995597}}
```

Affected File:
chubaofs/master/api_service.go

6 https://github.com/unrolled/secure

Affected Code:

The following code excerpt shows the weak authentication check and bypass.

```
if !param.skipOwnerValidation && !matchKey(vol.Owner, param.authKey) {
    sendErrReply(w, r, newErrHTTPReply(proto.ErrVolAuthKeyNotMatch))
    Return
}
```

ChubaoFS has a general pattern of openly exposing critical data. Even though the parameter is called *authKey*, the method implemented is a very weak form of authentication. This is made even worse by allowing an entire bypass of the check. It is recommended to never expose secrets on APIs and add proper authentication for critical data, especially if intended to be used in a multi-tenant deployment.

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.

CFS-01-001 WP1: Usage of *math/rand* within crypto-utils and utils (*Info*)

During an audit of the CFS source code, it was identified that the *encodeMessage()* function inside insecure *util/cryptoutil.go* is using the pseudo-random number[7] generation *math/rand* for generating a random *Uint64* number.

Affected File:
util/cryptoutil.go

Affected Code:
```
rand2 "math/rand"
[...]
func EncodeMessage(plaintext []byte, key []byte) (message string, err error) {
       var cipher []byte
       // 8 for random number; 16 for md5 hash
       buffer := make([]byte, RandomNumberSize+CheckSumSize+len(plaintext))
       // add random
       random := rand2.Uint64()
       binary.LittleEndian.PutUint64(buffer[RandomNumberOffset:], random)
[...]
```

It was also found that the generation of *accessKey* and *secretKey* is using a random string implementation building upon *math/rand* while being seeded by the current time. This results in an insecure generation of secrets.

Affected File:
util/string.go

Affected Code:
```
import (
       "math/rand"
       "strings"
       "time"
)
[...]
func RandomString(length int, seed RandomSeed) string {
       runs := seed.Runes()
```

[7] https://cwe.mitre.org/data/definitions/338.html

```
        result := ""
        for i := 0; i < length; i++ {
                rand.Seed(time.Now().UnixNano())
                randNumber := rand.Intn(len(runs))
                result += string(runs[randNumber])
        }
        return result
}
```

The above code snippets illustrate two locations within the CFS source code that are using the *math/rand* functionality. It is recommended to revisit the entire source code (a complete list can be obtained by searching for the string *math/rand*) for all occurrences of random value generation, especially those used for authentication and encryption. They must be replaced with secure random alternatives. The Go package *crypto/rand* offers an alternative to providing cryptographically secure random generation.

CFS-01-002 WP1: TLS version not enforced for *AuthNode* HTTP server *(Low)*

During an audit of the CFS source code, it was identified that the function *startHTTPService()* inside the file *authnode/http_server.go* has an empty TLS configuration in place. This effectively allows insecure and deprecated TLSv1.0 version.

Affected File:
authnode/http_server.go

Affected Code:
```
func (m *Server) startHTTPService() {
        go func() {
                m.handleFunctions()
                if m.cluster.PKIKey.EnableHTTPS {
                        // not use PKI to verify client certificate
                        // Instead, we use client secret key for authentication
                        cfg := &tls.Config{
                                //ClientAuth: tls.RequireAndVerifyClientCert,
                                //ClientCAs:  caCertPool,
                        }
[...]
```

The official Go lang documentation of the *tls* package[8] provides further information about the configuration setting for *MinVersion*, allowing to specify the minimum TLS version supported. As recommended by *NIST*[9], Cure53 encourages strict usage of TLSv1.2 and TLSv1.3 because older versions of TLS are vulnerable and no longer considered secure.

[8] https://golang.org/pkg/crypto/tls/
[9] https://csrc.nist.gov/News/2019/nist-publishes-sp-800-52-revision-2

CFS-01-006 WP1: Password hashes can be used to authenticate *(Low)*

It was found that the password hash of a user can be abused to authenticate to ChubaoFS. This introduces the risk that a compromise of the password storage instantly leaks all secrets that can be used to authenticate as any user to the system. Usually a hashing logic forces the attackers to reverse the hash to the clear-text password before authenticating to the system. ChubaoFS makes use of a hashing scheme to store the passwords. However, the password is hashed on the client-side before sending it to the server.

HTTP request:
```
POST /login HTTP/1.1
Host: console.chubao.io
[...]
content-type: application/json
authorization: null
Origin: http://console.chubao.io
Content-Length: 267
Connection: close

{"operationName":"Login","variables":
{"userID":"root","password":"082c2c44e2bfae761275e7e2f71d8771b276b32a"},"query":
"query Login($userID: String, $password: String) {\n  login(userID: $userID,
password: $password) {\n   token\n       userID\n     __typename\n  }\n}\n"}
```

It is recommended that all authentication secrets are hashed on the server. As a result, the risk of a password store compromise is mitigated by forcing attackers to defeat the cryptographic hash function used by the password verification logic.

CFS-01-012 WP1: HTTP parameter pollution in HTTP clients *(Medium)*

It was found that the application embeds potential user-input directly into the *value* part of a query parameter within a HTTP URI. This means the risk of user-input being poisoned by special *meta*-characters which allow the attackers to escape from the parameter allowing them to pollute other HTTP parameters. This could be abused by attackers to change the intention of the request sent to the *master* service, potentially resulting in unauthorized actions.

Affected File:
sdk/master/client.go

Affected Code:
```
func (c *MasterClient) mergeRequestUrl(url string, params map[string]string)
string {
    if params != nil && len(params) > 0 {
        buff := bytes.NewBuffer([]byte(url))
        isFirstParam := true
        for k, v := range params {
               if isFirstParam {
                      buff.WriteString("?")
                      isFirstParam = false
               } else {
                      buff.WriteString("&")
               }
               buff.WriteString(k)
               buff.WriteString("=")
               buff.WriteString(v)
        }
        return buff.String()
    }
```

Also Affected:
cli/api/metaapi.go
util/master_helper.go

It is recommended that the vulnerability is mitigated by escaping the user input before embedding it into the query parameters. This could be done by Go's *QueryEscape()* function offered by Golang's *net/url* package. By doing so, the attackers cannot escape from the parameter that they inject into preventing the HTTP Parameter Pollution.

CFS-01-013 WP1: Unsalted MD5 authKey-Computation in ObjectNode *(Low)*

During an audit of the CFS source code, it was identified that the *calculateAuthKey()* function uses the *MD5* hashing function to compute an authentication key based on the userID of the user. The *authKey* value is used by various functions to authenticate the caller against the API, for example when deleting a volume. The computation of the *MD5* hash is performed without a *salt* value, making it potentially vulnerable against rainbow / dictionary attacks.

Affected File:
objectnode/api_handler_bucket.go

Affected Code:
```
func calculateAuthKey(key string) (authKey string, err error) {
       h := md5.New()
       _, err = h.Write([]byte(key))
       [...]
       cipherStr := h.Sum(nil)
       return strings.ToLower(hex.EncodeToString(cipherStr)), nil
}
```

Cure53 recommends to replace *MD5* with *scrypt*[10] for computing hashes that are used for authentication purposes, as it is the case for the *authKey* value.

CFS-01-014 WP2: Lack of password complexity in *MasterNode* (*Low*)

An audit of the CFS source code revealed the user-management API is not enforcing a password complexity when creating a new user or updating the password of an existing user. It should enforce for instance, at least one uppercase and/or lowercase letter, a number or special characters or a minimum password length.

The lack of password policy allows users to set weak passwords, which makes it easy for attackers to guess passwords of existing users, for instance by mounting automated brute force or dictionary attacks.

Affected File:
master/user.go

Affected Code:
```
func (u *User) createKey(param *proto.UserCreateParam) (userInfo
*proto.UserInfo, err error) {
	var (
		AKUser     *proto.AKUser
		userPolicy *proto.UserPolicy
		exist      bool
	)
	[...]
	var userID = param.ID
	var password = param.Password
	if password == "" {
		password = DefaultUserPassword
	}
	[...]
```

Longer passwords are generally more resilient to brute-force attacks and the minimal length should be set at eight characters. As also described by *NIST*[11], passwords shorter than eight characters are considered to be weak.

[10] https://godoc.org/golang.org/x/crypto/scrypt
[11] https://pages.nist.gov/800-63-3/sp800-63b.html

CFS-01-017 WP2: Docker deploym. stores *client.json* as world-readable *(Medium)*

The default Docker deployment stores *accessKey* and *secretKey* in the *client.json* file, making it world-readable for all users. Even unprivileged users can gain access to the necessary credentials, achieving direct access on the volume.

PoC:
```
sh-4.4$ id
uid=1000(test1) gid=1000(test1) groups=1000(test1)
sh-4.4$ ls -la /cfs/conf/client.json
-rw-r--r-- 1 root root 501 Aug 17 12:48 /cfs/conf/client.json
sh-4.4$ cat /cfs/conf/client.json
{
  "masterAddr": "192.168.0.11:17010,192.168.0.12:17010,192.168.0.13:17010",
  "mountPoint": "/cfs/mnt",
  "volName": "ltptest",
  "owner": "ltptest",
  "logDir": "/cfs/log",
  "logLevel": "info",
  "consulAddr": "http://192.168.0.101:8500",
  "exporterPort": 9500,
  "profPort": "17410",
  "authenticate": false,
  "ticketHost": "192.168.0.14:8080,192.168.0.15:8081,192.168.0.16:8082",
  "enableHTTPS": "false",
  "accessKey": "39bEF4RrAQgMj6RV",
  "secretKey": "TRL6o3JL16YOqvZGIohBDFTHZDEcFsyd"
}
```

Storing sensitive information, like credentials, in clear-text inside a world-readable file is insecure and poses a severe security risk. In this context, it is recommended to change the permissions of the file to be more specific and not grant *read* permissions to any user on the system.

CFS-01-018 WP1: Docker deploym. logs credentials as world-readable *(Medium)*

It was discovered that the client stores logs which are quite verbose as they contain credentials (*accessKey* and *secretKey*). This log file is stored as world-readable, letting local attackers obtain the information with full access to the filesystem. Shown below is an excerpt from the logged information.

Affected File:
/cfs/log/client/output.log

Affected Logs:
```
2020/08/19 06:54:34 [certFile] string:
```

```
2020/08/19 06:54:34 [token] string:
2020/08/19 06:54:34 [accessKey] string: 39bEF4RrAQgMj6RV
2020/08/19 06:54:34 [secretKey] string: TRL6o3JL16YOqvZGIohBDFTHZDEcFsyd
2020/08/19 06:54:34 [disableDcache] bool: false
2020/08/19 06:54:34 [subdir] string:
```

Sensitive information like credentials should not be logged at all. In this context, it is recommended to censor the information in the log file. Additionally, the affected files should not be accessible for local users.

CFS-01-019 WP1: Folders can be moved into their own *child* folders *(Low)*

Inspecting the binary protocol demonstrated that the nodes lack plausibility checks for what clients submit, allowing to partly corrupt the filesystem. For example, it was found possible to move a folder into its own child item which render the affected directories useless on a standard Linux system. The following *shell* excerpt shows how such a directory loop was created. Reproducing the issue will require adjustments to the *Inode* and partition numbers.

PoC:

```
root@51cc9bb14576:/cfs/mnt/bla# mkdir a b
root@51cc9bb14576:/cfs/mnt/bla# ls -id a b
33554504 a        71 b
root@51cc9bb14576:/cfs/mnt/bla# (perl -e
'$r="{\"vol\":\"ltptest\",\"pid\":3,\"pino\":33554504,\"ino\":71,\"name\":\"b\",
\"mode\":2147484141}";print "\xff\x00\x22\x00" . "\x00"x8 . chr(length($r)) . "\
x00"x34 . "\x07\xbe" . "\x00"x8; print $r;' ) | nc
docker_metanode1_1.docker_extnetwork 17210
�"�O�{"vol":"ltptest","pid":3,"pino":33554504,"ino":71,"name":"b","mode":214748
4141}
root@51cc9bb14576:/cfs/mnt/bla# ls -al b/a/
ls: cannot access 'b/a/b': Too many levels of symbolic links
total 0
d????????? ? ? ? ?            ? b
```

The impact of this issue is fairly low. Given that it requires direct communication with the node, an attacker would have better ways for causing damage than corrupting individual folders. However, there might be scenarios where an attacker can gain more from this issue than simply delete all files. The issue is already partly addressed in the code of the *MetaNode* server but it is only checked if the *ParentID* is equal to the current *Inode*. As this issue shows, this check is not sufficient to prevent directory loops.

Affected File:
metanode/partition_op_dentry.go

Affected Code:
```
if req.ParentID == req.Inode {
	err = fmt.Errorf("parentId is equal inodeId")
	p.PacketErrorWithBody(proto.OpExistErr, []byte(err.Error()))
	return
}
```

Despite the low impact of the issue, it is recommended to check if *Dentries* link to one of their parents when created or modified in order to prevent corruptions caused by directories linking in the wrong way.

CFS-01-020 WP1: Missing filename-validation allows folder corruption *(Low)*

Inspecting the binary protocol reveals that the nodes lack plausibility checks for what clients submit, allowing to create file-names which lead to errors on Linux systems. This renders the affected directory useless and makes the contained data inaccessible unless the problem is resolved by manually crafting requests to the *MetaNodes*. The following *shell* excerpt shows how such a directory loop was created. Reproducing the issue will require adjustments to the *Inode* and partition numbers.

PoC:
```
root@51cc9bb14576:/cfs/mnt/t# ls -id . bla
33554514 .        82 bla
root@51cc9bb14576:/cfs/mnt/t# (perl -e
'$r="{\"vol\":\"ltptest\",\"pid\":3,\"pino\":33554514,\"ino\":82,\"name\":\"a/\"
,\"mode\":2147484141}";print "\xff\x00\x22\x00" . "\x00"x8 . chr(length($r)) .
"\x00"x34 . "\x07\xbe" . "\x00"x8; print $r;' ) | nc
docker_metanode4_1.docker_extnetwork 17210
�"�P�{"vol":"ltptest","pid":3,"pino":33554514,"ino":82,"name":"a/","mode":21474
84141}
root@51cc9bb14576:/cfs/mnt/t# ls -al
ls: reading directory '.': Input/output error
total 0
```

Similar to CFS-01-019, the impact of this issue is also fairly low. There are easier ways to cause damage once an attacker can talk to nodes directly. However, it is recommended to check on the node's side if the submitted data can lead to problems on the client-side. In this particular case, this can be achieved by validating names of fields and directories.

CFS-01-021 WP1: Debugging endpoint */debug/pprof* exposed (*Info*)

During an audit of the CFS source code, it was noted that the *pprof debug* endpoint[12] is exposed by various services of the CFS ecosystem. The *pprof debugging* endpoint can potentially leak sensitive information, such as internal memory addresses and configuration.

PoC:
The *pprof debugging* endpoint of the fuse client can, for example, be queried as follows:

```
# wget -O trace.out http://127.0.0.1:17410/debug/pprof/trace?seconds=5
```

Running the above command will store a five-seconds trace inside *trace.out*.

Affected File:
client/fuse.go

Affected Code:
```
import (
	"flag"
	"fmt"
	syslog "log"
	"net"
	"net/http"
	_ "net/http/pprof"

[...]

func mount(opt *proto.MountOptions) (fsConn *fuse.Conn,
	super *cfs.Super, err error) {
	[...]
	go func() {
		if opt.Profport != "" {
			syslog.Println("Start pprof with port:", opt.Profport)
			http.ListenAndServe(":"+opt.Profport, nil)
		} else {
			pprofListener, err := net.Listen("tcp", ":0")
			if err != nil {
				daemonize.SignalOutcome(err)
				os.Exit(1)
			}

			[...]
			http.Serve(pprofListener, nil)
		}
```

[12] https://golang.org/pkg/net/http/pprof/

```
	}()
```

Affected File:
cmd/cmd.go

Affected Code:
```
import (
	"flag"
	"fmt"
	syslog "log"
	"net/http"
	_ "net/http/pprof"

[...]

func main() {
	[...]

	if profPort != "" {
		go func() {
			http.HandleFunc(log.SetLogLevelPath, log.SetLogLevel)
			e := http.ListenAndServe(fmt.Sprintf(":%v", profPort), nil)
			if e != nil {
				log.LogFlush()
				daemonize.SignalOutcome(fmt.Errorf("cannot listen” \
					“ pprof %v err %v", profPort, err))
				os.Exit(1)
			}
		}()
	}
	[...]
```

Cure53 wants to stress the importance of not exposing *debug* interfaces in an unauthenticated form to all users that are in the position to interact with the CFS cluster. *pprof* should only be exposed within *debug* builds and when explicitly configured by users.

CFS-01-023 WP1: Build system lacks stack canaries, *PIE* and *FORTIFY* (*Medium*)

While checking the properties of the compiled *cfs-authtool*, *cfs-cli*, *cfs-client* and *cfs-server* binaries, it has been identified that none of the binaries have compile time security hardening flags enabled. The following security hardening options are missing across all binaries:

- *STACK CANARY*
- *PIE*
- *FORTIFY*

The following security hardening options are missing for the *cfs-authtool* and *cfs-client* binary as well:

- *RELRO*

A detailed description of the referred security hardening compiler flags can be found online[13].

PoC:

cfs-server:

```
# /root/tools/checksec.sh/checksec --file=cfs-server
[...]  STACK CANARY    [...]  PIE    [...]  FORTIFY
       No canary found        No PIE         No
```

cfs-cli:

```
# /root/tools/checksec.sh/checksec --file=cfs-cli
[...]  STACK CANARY    [...]  PIE    [...]  FORTIFY
       No canary found           No PIE         No
```

cfs-authtool:

```
# /root/tools/checksec.sh/checksec --file=cfs-authtool
RELRO         STACK CANARY      PIE      FORTIFY
Partial RELRO No canary found   No PIE    No
```

cfs-client:

```
# /root/tools/checksec.sh/checksec --file=cfs-client
RELRO         STACK CANARY      PIE      FORTIFY
Partial RELRO No canary found    No PIE    No
```

Setting the following environment variables within the build process in the file *build/build.sh* will result in having the referred security hardening options enabled:

[13] https://wiki.archlinux.org/index.php/Arch_package_guidelines/Security#Golang

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

```
export GOFLAGS='-buildmode=pie'
export CGO_CPPFLAGS="-D_FORTIFY_SOURCE=2"
export CGO_LDFLAGS="-Wl,-z,relro,-z,now"
```

Cure53 encourages the use of existing compiler security features in order to raise the bar for attackers who aim to exploit vulnerabilities within CFS.

CFS-01-025 WP1: Outdated vulnerable *bzip2* dependency for *ARM64* build (*Info*)

While reviewing the CFS build script, it has been noticed that the build for *ARM64* architecture uses an outdated and vulnerable version of the *bzip2* library[14].

Affected File:
build.sh

Affected Code:
```
# wget compress dep
get_rocksdb_compress_dep() {

if [ ! -d "${RootPath}/vendor/dep" ]; then
   mkdir -p ${RootPath}/vendor/dep
   cd ${RootPath}/vendor/dep
   wget https://astuteinternet.dl.sourceforge.net/project/bzip2/bzip2-1.0.6.tar.gz
   [...]
```

Affected File:
build/build.sh

Affected Code:
```
pre_build_server() {
   rocksdb_libs=( z bz2 lz4 zstd )
   if [[ "$CPUTYPE" == arm64* ]];
   then
       build_zlib
       build_bzip2
       build_lz4
    #   build_zstd
   else
   [...]

   build_bzip2() {
   Bzip2SrcPath=${VendorPath}/dep/bzip2-1.0.6
   [...]
```

[14] https://www.cvedetails.com/vulnerability-list/vendor_id-1198/produ...Bzip-Bzip2-1.0.6.html

The current stable version is *bzip2* 1.0.8[15] and it is recommended to use the latest available version to prevent potential exploitation of existing vulnerabilities.

CFS-01-026 WP2: *cfs-server* processes running with *root* privileges (*Medium*)

It was found that the nodes of the CFS cluster, including *MetaNode, DataNode* and *MasterNode*, are running with *root* privileges. It is considered bad practice to let services run under *root* privileges, also inside Docker containers. A single bug within one of the running processes could potentially be leveraged by an attacker to gain *root* privileges.

PoC:

MetaNode

```
root@c61d3d4d9d16:/cfs/log/metaNode# top
[...]
7 root      20   0  839816  63144  14400 S   0.0  0.8  63:33.52 cfs-server
root@c61d3d4d9d16:/cfs/log/metaNode# cat /proc/7/cmdline
/cfs/bin/cfs-server -f -c /cfs/conf/metanode.json
```

DataNode:

```
root@b9065346f8ee:~# top
[...]
8 root      20   0 2006564 114992  14280 S   0.0  1.4 265:56.13 cfs-server
root@b9065346f8ee:~# cat /proc/8/cmdline
/cfs/bin/cfs-server -f -c /cfs/conf/datanode.json
```

Master:

```
root@1b326b438ba2:~# top
[...]
7 root      20   0 1447252  65388  15100 S   0.0  0.8  74:53.19 cfs-server
root@1b326b438ba2:~# cat /proc/7/cmdline
/cfs/bin/cfs-server -f -c /cfs/conf/master.json
```

Cure53 recommends to assign the least privileges necessary and not to run all nodes within the CFS cluster with *root* privileges. This can be accomplished by creating a user with a known UID in the Dockerfile and running the applications as the newly created user.

[15] https://www.sourceware.org/bzip2/downloads.html

CFS-01-027 WP1: Potential path traversal in *MetaNodes* (*Low*)

While auditing the code of the *MetaNodes*, it was noticed that some operations are vulnerable to path traversals. When a protocol participant can send arbitrary operations to *MetaNodes*, commands like *opFSMInternalDelExtentCursor* can be used to influence arbitrary files.

Affected File:
chubaofs/metanode/partition_fsmop.go

Affected Code:
```
func (mp *metaPartition) setExtentDeleteFileCursor(buf []byte) (err error) {
       str := string(buf)
       var (
              fileName string
              cursor   int64
       )
       _, err = fmt.Sscanf(str, "%s %d", &fileName, &cursor)
       fp, err := os.OpenFile(path.Join(mp.config.RootDir, fileName),
              os.O_CREATE|os.O_RDWR,
              0644)
       //[...]
       if err = binary.Write(fp, binary.BigEndian, cursor); err != nil
       //[...]
```

The full impact of this issue is unclear, as the limited time frame did not allow for a complete research into the *raft* protocol. Cure53 is unclear on whether this code path is only reachable for other *MetaNodes*, or if anybody on the network could join and issue such operations. That is why the issue is labeled as an *informational misc* flaw. However, it is a security beneficial approach if an input is seen as attacker-controlled. Thus, file-paths should not be blindly trusted and additional checks should be used to make sure a path-traversal outside of the configured *root* directory is not possible.

CFS-01-028 WP1: Insecure *ObjectNode* policy-checking behavior (*Medium*)

While auditing the code of the *ObjectNode* component, it was noticed that the policy-checking routing is verifying the bucket policy and ACL. The verification of the policy and ACL is performed in two steps:

1. The bucket policies are checked; if the request is not allowed, the function returns *false*.
2. The bucket ACLs are checked; if the request is not allowed, the function should return *false*.

However, it was noticed that the function verifying if the request is allowed for an ACL is returning *true* when the list of ACL grants is empty. This insecure default behavior is risky and should be negated, meaning that if an empty list of ACL grants is provided, the bucket access should be denied. Moreover, it was noticed that the function *policyCheck()* returns *true*, similarly to allowing bucket access when the calls to *policy.IsEmpty()* and *acl.IsAclEmpty()* return *true*. This is demonstrated below.

Affected File:
objectnode/policy.go

Affected Code:
```
func (o *ObjectNode) policyCheck(f http.HandlerFunc) http.HandlerFunc {
       [...]
       if vol != nil && policy != nil && !policy.IsEmpty() {
              allowed = policy.IsAllowed(param, isOwner)
              if !allowed {
                     log.LogWarnf("policyCheck: bucket policy not allowed:
                     requestID(%v) userID(%v) accessKey(%v) volume(%v) action(%v)",
                     [...]
                     return
              }
       }

       if vol != nil && acl != nil && !acl.IsAclEmpty() {
              allowed = acl.IsAllowed(param, isOwner)
              if !allowed {
                     log.LogWarnf("policyCheck: bucket ACL not allowed:
                     requestID(%v) userID(%v) accessKey(%v) volume(%v) action(%v)",
                     [...]
                     return
              }
       }

       allowed = true
       [...]
```

Affected File:
objectnode/acl.go

Affected Code:
```
func (acp *AccessControlPolicy) IsAllowed(param *RequestParam, isOwner bool) bool {
       log.LogDebugf("acl is allowed: %v param: %v", acp, param)
       if len(acp.Acl.Grants) == 0 {
              return true
       }
       if isOwner {
              return true
```

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

```
        }
        for _, grant := range acp.Acl.Grants {
                if grant.IsAllowed(param) {
                        return true
                }
        }
        return false
}
```

The full impact of this default *ALLOW* behavior, and in case the list of ACL grants is empty, it has not been verified by Cure53 as this issue has been identified at the end of the assessment. The behavior of the *IsAllow()* method for ACL objects should be similar to the *IsAllow()* method for policy objects and return *false* (denying access) in case an empty list of ACL grants is provided.

Conclusions

As already discussed in the opening paragraphs of this report, Cure53 believes that the ChubaoFS project still has a long way to go before reaching decent security maturity. As can be derived from the large total number of findings standing at twenty-eight, as well as numerous high-graded problems, this August 2020 project makes it very clear that ChubaoFS needs further security-centered investments and efforts. After spending 32 days investigating the scope, six members of the Cure53 team can conclude that approaching the problems with a large-scale re-audit is advised. While the project benefited from generous support from the CNCF scheme, Cure53 identifies a pressing need for a follow-up that needs to take place once the majority of issues has been fixed by the ChubaoFS development team.

Even though the outcome might not be in line with what has been expected, Cure53 must emphasize that the ChubaoFS team has been extremely helpful in terms of both test-preparations and during the actual assessment. It is clear to Cure53 that a lot of time and energy has been invested to make optimal coverage possible. Further, the work has undoubtedly gone into preparing a properly set-up and complex test-environment, which is just another positive sign of professionalism and dedication in-house. It is also in part this steady support that enabled the audit team to get very good coverage reflected by the numbers of findings in this report.

To offer some general notes on the security posture of the examined compound, it should be repeated that the claimed core functionality of ChubaoFS is to provide a distributed filesystem. In the current state, however, this goal is not realized due to lacking authentication features. In a private network deployment, ChubaoFS can likely be used while maintaining low risk. At the same time, the lack of authentication means that any multi-tenant-style deployment is inherently insecure, making it possible for any client to access any other client’s data.

This means that even inside an internal network deployment, vulnerable or malicious hosts inside that same network place the deployment at risk. As such, ChubaoFS might be a fruitful target for an attacker who wishes to further escalate or steal data. In that sense, the software is not ready for production use when it comes to storage of *personalised, private* or *confidential* information. It is only deemed fit for storing publicly available information at the moment. The developers themselves already reference some of the security concerns in *docs/source/design/authnode.rst*. Some of these points were raised by the auditors during this assessment as well. Once the mentioned *AuthNode* feature has been completed, it will potentially solve some of the issues, yet it is simply not ready for going through the security ‘graduation’.

Moving to some details, authentication is a primary concern and shall be seen as 'work in progress' at the moment. The provided test-setup lacked this feature, basically letting everyone within the test network access all data. This is especially problematic for the API that can be accessed by pretty much anyone (CFS-01-015). Authentication problems are also present in the client itself as there are no restrictions for local system users, signifying that anyone can read and modify *root*-owned files (CFS-01-004).

On top of that, the client stores world-readable logs and configuration files containing the credentials used to access the filesystem (CFS-01-017, CFS-01-018). Specifically, the lack of authentication for HTTP endpoints also opens up the possibility of SSRF attacks as this report shows in CFS-01-015. Further, the fuse mount options disable the default Linux file permissions behavior. Thus, unprivileged users can access everything in the same way a *root* user could. In most deployments this might not be an issue, however this is not clearly communicated and might result in unexpected problems elaborated on in CFS-01-004.

Cure53 wishes to next up comment on the encryption issues. On a few occasions where ChubaoFS actually utilizes cryptography related functions, they are mostly used in insecure ways. The encryption mechanism used to protect authentication tickets is flawed and could allow attacks via a padding oracle (CFS-01-005), especially since random data is generated using functions with predictable output rather than with reliance on proper cryptographic generators (CFS-01-001). Further, comparisons are not safe against timing attacks (CFS-01-007) and obsolete hashing functions are in use as well (CFS-01-003, CFS-01-013). Beyond the authentication related parts, the insufficient encryption security could also be an issue inside bigger deployments across network boundaries, for example when a ChubaoFS is deployed across multiple datacenters, as shown in CFS-01-008.

To sum up this realm, it was rather surprising to see ChubaoFS test setup offering a multi-tenant deployment, as no security boundaries between tenants can be established in the first place. For potential future audits, it is strongly advised to foster some deep-dive research into the *raft* protocol dependency[16] ChubaoFS makes use of. Close inspection of the security impact it might have on ChubaoFS in a multi-tenant deployment should be seen as "a must" step for securing the premise.

When it comes to best practices and documentation, the ChubaoFS console features exposes various problematic issues related to storage (CFS-01-003) and verification (CFS-01-007, CFS-01-006) of user-passwords. Therefore, authentication on the console should be improved in general to prevent unauthorized access either after the compromise of the password storage or via targeted timing-attacks that are cluster-

[16] https://github.com/tiglabs/raft

optimized. After fixing the issues in this report and introducing several new security settings, particularly to improve authentication, the documentation for ChubaoFS should make use of a dedicated security section where the intended modus operandi is described and relevant *security goals* and *security recommendations* are explained in-depth.

It is advisable that the majority of best-practice security settings within the proposed guide should be applied by default to both the Docker system and the Kubernetes Helm Chart. In Cure53's view, this should also include explanations for potential *NetworkPolicies* and ingress controllers. Anything that requires custom configuration by user-interaction should be followed-up on with alerting the user both in the administrator panel and in *CLI* that deploys ChubaoFS. Cure53 envisions best practice settings being deployed in a test-environment with multiple clients, so that such complex can be targeted during a re-audit, verifying whether the security promises made by ChubaoFS are in fact kept.

Another point to make is that the maintainers claim that it is not necessary to perform data and / or metadata encryption at rest. They justify this by relying on the fact that, for example, data is stored as distributed within the filesystem and an attacker needs access to metadata information in order to get all relevant chunks of a file before being able to combine them. However, this statement is not accurate and it must be emphasized that, in the current state, the ChubaoFS lets anyone query metadata information of their choosing, thus potentially facilitating file-recovery. This weakness might be mitigated once the mentioned *AuthNode* setup has been completed.

In addition, the *ObjectStorage* interface for S3 is also incomplete and certain features and operations offered by S3 are not implemented in ChubaoFS (e.g. encryption). Features, such as S3 encryption, are very valuable for protecting data at rest within S3 buckets and would offer additional security for the CFS customers. On the plus side, the support of HTTPS within the various exposed HTTP services is inconsistent. This is apparent when one searches for the configuration directive and string *"enableHTTPS"* within the GitHub repository. For example, various *.json* configuration files are having *"enableHTTPS"* set to *false.*

When looking at the corresponding Go lang code fragments responsible for parsing the *.json* files, the parser is not even looking for the *"enableHTTPS"* option. This lets Cure53 conclude that the relevant code is still being largely developed. Correspondingly, encrypted communication for the various services has been considered but not implemented yet. For future assessments and engagements auditing this scope, it is highly encouraged to engage externals once the *AuthNode* development has been finished.

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

It is quite difficult to briefly summarize the state of security found on ChubaoFS during this summer 2020 project requested by CNCF. On the one hand, Cure53 can conclude that the core features and basic idea behind the ChubaoFS complex as a distributed storage platform is great from a functional and feature-centric perspective. This is especially valuable when providing POSIX-compliant and S3-compatible interfaces is at stake. However, taking all identified security problems and further considerations into account, it is evident that security and privacy have not been the highest priority during the development of ChubaoFS until now.

Conclusively, one can safely say that ChubaoFS was audited in a rather early state of security implementations. Cure53 can recommend using this August 2020 audit and its results for clear pointers regarding the existing gaps and shortcomings. It is hoped that the ChubaoFS team can use it for direction when reworking the software in terms of security and general resilience against attacks and data leaks. In its current state, the software cannot be recommended for production as long as the intended use involves non-public data. It cannot be repeated enough that another thorough look must be taken at the ChubaoFS project upon the implementation of *AuthNode* and successful finalization of fixes for issues identified by Cure53 and beyond.

Cure53 would like to thank Liying Zhang, Mofei Zhang and Wei Ding from the ChubaoFS team as well as Chris Aniszczyk of The Linux Foundation for their excellent project coordination, support and assistance, both before and during this assignment.