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

Pentest- & Audit-Report Swarm 03.-04.2021

Cure53, Dr.-Ing. M. Heiderich, M. Wege, Dipl.-Inf. G. Kopf & other team members

Index

Introduction

"Swarm is a decentralised data storage and distribution technology, ready to power the next generation of censorship resistant unstoppable serverless apps. It is the complement to blockchain based smart contracts originally envisaged by the Ethereum cryptocurrency and provides the mass storage piece in the Dapp building puzzle. Swarm is the hard drive of the world computer."

From https://docs.ethswarm.org/docs/

This report describes the results of a penetration test and source code audit against the Ethereum Swarm software compound, encompassing various pieces of software and smart contracts. Carried out by Cure53 in spring 2021, the project identified twelve security-relevant risks on the scope delineated by the Swarm team.

To give some context, the work was requested by the Swarm Association in February 2021 and then promptly scheduled. Cure53, with a team consisting of six senior testers, conducted the assignment in March and April, namely in a timeframe between CW13 and CW16 of 2021. A total of thirty-three days were invested to reach the coverage expected for this project by six members responsible for the project's preparation, execution and finalization.

For optimal progress, coverage and tracking of issues, the work was split into four separate work packages (WPs). These read as follows:

- **WP1**: Threat-Modeling Exercise to determine the exact scope for WP2-WP4
- **WP2**: Penetration Tests & Source Code Audits against Ethswarm *bee*
- **WP3**: Penetration Tests & Delta Code Audits against *bee-clef/clef*
- **WP4**: Smart Contract Audits against Swarm-related Solidity files

White-box methods were applied in the project. Cure53 was given access to all relevant material, including source code behind various components and detailed documentation. Most importantly, several meetings and mini-workshops with the Swarm team were held to facilitate comprehensive understanding of the objectives and technical traits of the objects subject to this review.

All preparations were done in mid-to-late March 2021, namely in CW11 and CW12, so Cure53 could have a smooth start and a firm grasp of the scope before the actual threat-modeling and following source-code auditing and pentesting work started. Communications during the test were done using a Mattermost channel that all participating Cure53 team members joined.

Together with the Swarm team, they could discuss the test and audit-related issues. The communications were very smooth and productive; no noteworthy roadblocks were encountered during the test.

Given the complex scope, a lot of questions were asked and promptly answered by the Swarm maintainer team. The in-house team generally did a great job helping Cure53 to navigate through this exercise. Cure53 further furnished frequent status updates about the test and the related findings. Although live-reporting was not requested, the Swarm team was regularly kept up to date about the spotted findings and security concerns when such appeared.

The Cure53 team managed to get good coverage over the WP1-4 scope items, despite several parts of the complex still being 'work-in-progress' and quite in flux. As noted, the test and audit team managed to spot a total of twelve findings, nine of which were classified to be security vulnerabilities and three to be general weaknesses with lower exploitation potential. It needs to be noted that no findings of *Critical* or even *High* severity were spotted, which means that the majority of issues were concluded to represent flaws ith *Medium, Low* or *Informational* scores.

The absence of serious problems is generally a good sign. However, it should not be seen as a conclusive verdict given that the software compound is still quite young and, as mentioned, continues to have many moving parts, alongside some unsolved design and architectural choices. At the same time, because the work took place so early in the development process, it can be stated that the Swarm team exposes a high level of security and privacy awareness.

In the following sections, the report will first shed light on the scope and key test parameters, as well as the structure and contents of the subsequent WPs. After that, a dedicated chapter will present the results of the threat-modeling sessions preceding this test and audit. This is envisioned as a means to showcase clearly under what security privacy assumptions the examinations have taken place.

Next, all findings will be discussed in grouped vulnerability and miscellaneous categories, then following a chronological order in each group. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions about this April 2021 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team's observations and collected evidence. Tailored hardening recommendations for the Ethereum Swarm complex are also incorporated into the final section.

Scope

- **Penetration Tests, Reviews & Audits against Swarm**
  - **WP1**: Threat-Modeling Exercise to determine exact scope of work for WP2
  - **WP2**: Penetration Tests & Source Code Audits against Swarm *bee*
  - **WP3**: Penetration Tests & Delta Code Audits against Swarm *bee-clef/clef*
  - **WP4**: Smart Contract Audits against Swarm-related Swarm Solidity files
- **Sources were made available for auditing**
  - https://github.com/ethersphere/bee/tree/v0.5.3
  - Out of scope
    - *cmd/bee-file*
    - *cmd/bee-join*
    - *cmd/bee-split*
    - *pkg/pss*
    - *pkg/collection*
    - *pkg/recovery*
  - https://github.com/ethersphere/bee-clef/tree/v0.4.9
  - https://github.com/ethereum/go-ethereum/tree/master/cmd/clef
  - https://github.com/ethersphere/swap-swear-and-swindle/tree/v0.4.0
  - https://github.com/ethersphere/storage-incentives
    - Commit: *2a19961ff8dccd017e64d7299d4b5416eb176f49*
- **Servers made available for testing**
  - **Nodes**
    - bee-0.gateway.do.ethswarm.org
    - bee-1.gateway.do.ethswarm.org
    - bee-2.gateway.do.ethswarm.org
    - bee-3.gateway.do.ethswarm.org
    - bee-4.gateway.do.ethswarm.org
  - **Gateway**
    - https://gateway.do.ethswarm.org

Threat Focus

The following paragraphs enumerate the evidence collected in relation to the threat modeling exercise. The main points behind that stem from several online discussions and collation of ideas with the development team, with an emphasis on the protectable assets and the consequentially known and unknown threats to the current implementation of the Swarm system.

Concerns and Assets

- Node blackholing
- Invalidating plausible deniability
- Node profiteering
  - Node thrashing
- Address mining
- Content censoring
- Protocol detection
  - Information deduction
- Key management
- Leaked key material
- Automated signing
- Message misappropriation
  - ANSI terminal escaping
  - Rich formatting abuse
- Topology formation strategy
- Eclipse attacks
  - Address spamming
- Published underlay addresses
  - Leaking internal routing
  - Leaking network configuration
- Logging handling
  - Leaking personal information
  - General anonymization
  - Leaking root hashes
  - Output escaping
- Liquidity provisioning
- API verification
  - Debug API abuse
- File joining
- Harming users
- Balance draining

- Content handling
  - Junking content
  - Content spamming
  - Corrupting content
  - Damaging content
- Game-theoretical aspects
  - Incentivization
  - Token sale
  - Fairness
  - Pricing
  - Privacy
  - Anonymity
- Content-addressed hashes
  - Phishing websites
  - Compromised Dapp
  - Upload forcing
- Feeds
  - Encryption coercion
  - User interaction
  - Write flooding
  - Hash exposure
  - Content extraction

To properly accommodate all the collected details from the initial knowledge gathering phase, the auditors accounted for the following adversarial avenues.

Attack vectors

- Contract coding practices
  - Arithmetic issues
  - Reentrancy bugs
  - Weak cryptography
    - Misused primitives
  - Unchecked external calls
  - Frontrunning
  - Cheque cashing denial
  - Transactions replay
  - Hard deposit bypassing
- Corrupt HIVE protocol
  - Spam node address book
- Confused Kademlia routing
- Reward misappropriation

- CPU-bound DoS issues
  - Cryptographic operations
  - General computations
- Gas-based DoS issues
  - Trick others into spending
- URL and path normalization
  - Directory traversal
- Archive file uploading
- Cause billing events
  - Drain client funds
- Disrupt file chunking
- Corrupt other protocols
  - BZZ handshake
  - Ping-pong
  - Push/pull-sync
- Hash tree construction
  - Create structural loops
- Node blacklisting
  - Unsolicited chunk DoS

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

SWA-01-001 WP4: Signature replay in *setCustomHardDepositTimeout* (*Medium*)

While analyzing the smart contracts, it was found that the scheme for setting custom hard deposit timeouts is susceptible to a signature replay attack.

Affected File:
contracts/ERC20SimpleSwap.sol

Affected Code:
```
function setCustomHardDepositTimeout(
     address beneficiary,
     uint hardDepositTimeout,
     bytes memory beneficiarySig
) public {
     require(msg.sender == issuer, "not issuer");
     require(
     beneficiary == recoverEIP712(customDecreaseTimeoutHash(
            address(this), beneficiary, hardDepositTimeout), beneficiarySig),
     "invalid beneficiary signature"
     );
     hardDeposits[beneficiary].timeout = hardDepositTimeout;
     emit HardDepositTimeoutChanged(beneficiary, hardDepositTimeout);
}
```

It can be observed that the function cannot be invoked without providing a signature created by the beneficiary. This is a sensible decision, as the beneficiary should have to agree to reducing the hard deposit timeout. Otherwise, the issuer could arbitrarily reduce the hard deposit timeout, then immediately reduce the hard deposit for the beneficiary and, ultimately, break their implicit promise to reserve a hard deposit for the beneficiary.

It should, however, be noted that the beneficiary signature only covers the address of the cheque book, the beneficiary address and the hard deposit timeout. In case the hard deposit timeout is changed multiple times, the issuer could record the beneficiary signatures and later reuse them.

As an example: assume that the issuer and beneficiary agree to set a low timeout of one hour. The beneficiary signs this value and the issuer invokes *setCustomHardDepositTimeout*. Later on, both parties agree that this timeout should be increased to one day. Again, the beneficiary provides their signature and the issuer invokes *setCustomHardDepositTimeout*. However, the issuer now has two signatures: one for a timeout of an hour, one for a timeout of one day. The issuer can now invoke *setCustomHardDepositTimeout* as they like, alternating the timeout between one hour and one day without the consent of the beneficiary.

There are several ways to address this problem. One option would be to include additional state into the smart contract for holding a per-beneficiary NONCE value. This NONCE could be required to be included in the beneficiary signature. Each time a beneficiary signature is verified, it could be incremented.

SWA-01-002 WP4: Gas waste amplification *(Low)*

While reviewing the overall SWAP incentivization scheme, the following observation was made: Entering the system and obtaining a cheque-book is an operation that is not for free. However, it requires at most a one-time investment of X. Assume an attacker wants to disturb the system; in order to mount an efficient attack, a rational attacker could argue that if the damage caused by their attack exceeds their original investment X, then the attack is worthwhile.

Assume that the attacker now interacts with various peers, with each interaction governed by the SWAP contract. The attacker issues cheques for large amounts to each of the peers and then starts using their services. This incentivizes the peers to eventually cash their cheques. However, the attacker meanwhile drains all funds from the cheque-book contract, so that all cheques will bounce. One can observe that cashing a cheque is also not an operation that is free. It will cost the respective beneficiary at least the transaction costs Y. If the attacker can now arrange a situation where in the same block multiple beneficiaries (say n) attempt to cash a bouncing cheque, the attacker causes an overall n*Y. However, the attacker initially only invests the amount X for preparing the attack. This means that an attacker could be able to amplify the damage caused by a bouncing cheque by distributing it over multiple parties.

Affected File:
contracts/ERC20SimpleSwap.sol

Affected Code:
```
function _cashChequeInternal(
      address beneficiary,
      address recipient,
        uint cumulativePayout,
        uint callerPayout,
        bytes memory issuerSig
  ) internal {
[...]

/* let the world know that the issuer has over-promised on
        outstanding cheques */
if (requestPayout != totalPayout) {
        bounced = true;
        emit ChequeBounced();
}
```

The above code excerpt shows that a bouncing cheque does not lead to a reversion of the contract. Furthermore, cheques that do not lead to a payout (or, rather, lead to a payout of zero) are not necessarily reversed.

In order to address this problem, it is recommended to adjust the SWAP smart contract so that it will revert when the payout amount of a cash operation is zero.

SWA-01-003 WP4: Unsafe integer arithmetic *(Low)*

While reviewing the smart contract implementations, it was found that there are a number of places where unsafe integer arithmetic are used.

Affected File:
src/PriceOracle.sol

Affected Code:
```
        function setPrice(uint256 _price) external {
        require(hasRole(PRICE_UPDATER_ROLE, msg.sender), "caller is not a price 
updater");

        // if there was a last price, charge for the time since the last update 
with the last price
        if(lastPrice != 0) {
                uint256 blocks = block.number - lastUpdatedBlock;
                postageStamp.increaseTotalOutPayment(lastPrice * blocks);
        }
```

It can be observed that the multiplication of *lastPrice * blocks* is not using safe integer arithmetic and might overflow. Although this does not appear to be particularly likely as high values for *lastPrice* and/or *blocks* would be required, it might be advisable to use *SafeMath*[1] here in order to reduce the attack surface.

[1] https://docs.openzeppelin.com/contracts/2.x/api/math

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

Affected File:
contracts/ERC20SimpleSwap.sol

Affected Code:

```
  function prepareDecreaseHardDeposit(address beneficiary, uint decreaseAmount)
public {
        require(msg.sender == issuer, "SimpleSwap: not issuer");
        HardDeposit storage hardDeposit = hardDeposits[beneficiary];
        /* cannot decrease it by more than the deposit */
        require(decreaseAmount <= hardDeposit.amount,
                "hard deposit not sufficient");
        // if hardDeposit.timeout was never set, apply defaultHardDepositTimeout
        uint timeout = hardDeposit.timeout == 0 ? defaultHardDepositTimeout :
                hardDeposit.timeout;
        hardDeposit.canBeDecreasedAt = block.timestamp + timeout;
        hardDeposit.decreaseAmount = decreaseAmount;
        emit HardDepositDecreasePrepared(beneficiary, decreaseAmount);
  }
```

It can be observed that the addition *block.timestamp* + *timeout* is not using safe integer arithmetic and might also overflow. Although this does not appear to be particularly likely as a high value for *timeout* would be required, it might be advisable to use *SafeMath* here as well, so as to reduce the attack surface.

SWA-01-004 WP2: Missing maximum file size check inside client API *(Low)*

During a source code review of the *bee* client API reachable over HTTP, the discovery was made that no checks are in place to prevent a user from uploading very large files. This could potentially cause a Denial-of-Service (DoS) situation when exhausting the allocated storage limit of a *bee* node, for instance.

One can pertinently note that this issue has also been discussed with the customer during the course of this security assessment; Swarm confirmed that no maximum file size check is currently in place.

Affected File:
bee/pkg/api/file.go

Affected Code:
```
func (s *server) fileUploadHandler(w http.ResponseWriter, r *http.Request) {
        [...]
}
```

It is recommended to perform a maximum file size check within the function *fileUploadHandler()* in order to avoid a potential DoS situation. This would prevent a malicious user from disabling a *bee* node server, especially since the system may run out of disk space when (numerous) excessively-sized files are uploaded.

SWA-01-005 WP2: Usage of *ioutil.ReadAll* in client API can result in DoS *(Low)*

During a source code review of the *bee* repository, it was spotted that various client API handler routines are using *ioutil.ReadAll* for reading user-input transmitted as part of HTTP requests within the body. The usage of *ioutil.ReadAll* is dangerous[2] and can result in a DoS situation as this function continues to read data into a buffer allocated on the system heap until it has received EOF. A malicious user could leverage this behavior by sending HTTP requests to a *bee* node in order to cause an out-of-memory situation.

Affected Files:

- *bee/pkg/api/chunk.go*
- *bee/pkg/api/pin_chunks.go*
- *bee/pkg/api/pss.go*
- *bee/pkg/api/soc.go*
- *bee/pkg/api/tag.go*

The following code snippet shows one example where *ioutil.ReadAll* is used to read attacker-controlled input. The same pattern can be observed within all listed source code files.

Affected Code:

```
func (s *server) chunkUploadHandler(w http.ResponseWriter, r *http.Request) {
	[...]
	data, err := ioutil.ReadAll(r.Body)
	[...]
}
```

Using *iotutil.ReadAll* to process untrusted input is discouraged, as it allows malicious users to cause DoS situations. In order to cap the amount of memory that a *bee* node is using during processing of the HTTP requests, an alternative approach is to read into a buffer using Golang's *bufio* package[3]. This would effectively limit the internal buffer used to store the parsed request[4].

[2] https://haisum.github.io/2017/09/11/golang-ioutil-readall/
[3] https://golang.org/pkg/bufio
[4] https://golang.org/pkg/bufio/#Scanner.Buffer

SWA-01-006 WP2: Spamming address books with HIVE protocol *(Medium)*

The HIVE protocol is used by *bee* nodes during bootstrapping and when joining the Swarm network, so as to discover peers within the network. During a review of the HIVE protocol implementation, it was noticed that a *bee* node has no limitations in terms of incoming addresses being added to the address book through HIVE.

Affected File:
bee/pkg/hive/hive.go

Affected Code:
```
func (s *Service) peersHandler([...]) error {
	[...]
	var peersReq pb.Peers
	if err := r.ReadMsgWithContext(ctx, &peersReq); err != nil {
		_ = stream.Reset()
		return fmt.Errorf("read requestPeers message: %w", err)
	}
	[...]
	var peers []swarm.Address
	for _, newPeer := range peersReq.Peers {
		bzzAddress, err := bzz.ParseAddress(newPeer.Underlay, 
newPeer.Overlay, newPeer.Signature, s.networkID)
		if err != nil {
			[...]
			continue
		}

		err = s.addressBook.Put(bzzAddress.Overlay, *bzzAddress)
	}
	[...]
}
```

Addressing such spamming attacks is generally a challenging task. One obvious approach is to rely on rate-limiting and reactive measures (such as purging invalid addresses later on). Another generic approach is to rely on negative economic incentivization. One could think about making the execution of attacks expensive enough as a way to prevent them.

There are several possible approaches for making an operation expensive: one approach here could be to couple it with a blockchain transaction. For instance, if obtaining a valid *bzzAddress* is sufficiently expensive, such attacks would likely be less frequent. However, it should be noted that there is a tradeoff between increasing the costs for a legitimate user and increasing the costs for an attacker. Therefore, providing a concrete recommendation on how to “correctly” address this issue is not possible.

SWA-01-008 WP2: Injection of corrupted underlay into P2P handshake *(Medium)*

While auditing *libp2p* and the handshake protocol, it was observed that the underlay address is retrieved from the unsigned part of the *SynAck* structure (second packet within the handshake), thereby being unprotected. An attacker could leverage this and modify that part of the message without the client noticing during the initial handshake, which would result in corrupting the underlay network information provided to the client.

Affected File:
bee/pkg/p2p/libp2p/internal/handshake/handshake.go

Affected Code:
```
func (s *Service) Handshake(ctx context.Context, stream p2p.Stream, 
peerMultiaddr ma.Multiaddr, peerID libp2ppeer.ID) (i *Info, err error) {
	[...]
	observedUnderlay, err := ma.NewMultiaddrBytes(resp.Syn.ObservedUnderlay)
	if err != nil {
		return nil, ErrInvalidSyn
	}
	[...]
}
```

It is recommended not to retrieve the observed underlay information from the unprotected part of the *SynAck* response message. A better approach would be to take the observed underlay information from the already present *fullRemoteMABytes* variable.

SWA-01-009 WP2: Blocklist bypass via bogus *ACK* in P2P handshake *(Medium)*

While auditing *libp2p* and the handshake protocol, it was noticed that the last message within the handshake obtains the *remoteBzzAddress* from the initiator’s final *ACK* packet (packet number 3 of the handshake protocol). Later on, the retrieved *bzzAddress* is used to check whether the received address is within the node’s blocklist. A malicious user could leverage this behavior and replay an *ACK*, for example extracted from the *SynAck* packet (packet number 2 of the handshake protocol). This would have been transmitted by the “server” side, allowing to bypass the check that verifies if the initiator is part of the node’s blocklist.

Affected File:
bee/pkg/p2p/libp2p/libp2p.go

Affected Code:
```
func New([...]) (*Service, error) {
	[...]
```

```
        // handshake
        s.host.SetStreamHandlerMatch(id, matcher, func(stream network.Stream) {
                i, err := s.handshakeService.Handle([...])
                if err != nil {
                        s.logger.Debugf("handshake: handle %s: %v", peerID, err)
                        [...]
                        return
                }

                blocked, err := s.blocklist.Exists(i.BzzAddress.Overlay)
                if err != nil {
                        s.logger.Debugf("blocklisting: exists %s: %v", peerID, err)
                        [...]
                        return
                }

                if blocked {
                        s.logger.Errorf("blocked connection from blocklisted peer
                                %s", peerID)
                        _ = handshakeStream.Reset()
                        _ = s.host.Network().ClosePeer(peerID)
                        return
                }
                [...]
        }
        [...]
}
```

Affected File:
bee/pkg/p2p/libp2p/internal/handshake/handshake.go

Affected Code:

```
func (s *Service) Handle([...]) (i *Info, err error) {
        [...]
        var ack pb.Ack
        if err := r.ReadMsgWithContext(ctx, &ack); err != nil {
                return nil, fmt.Errorf("read ack message: %w", err)
        }

        remoteBzzAddress, err := s.parseCheckAck(&ack)
        if err != nil {
                return nil, err
        }

        [...]

        return &Info{
                BzzAddress: remoteBzzAddress,
                Light:      ack.Light,
        }, nil
}
```

In order to address the described issue, it is recommended to verify whether the peer information within the *ACK* message in the protocol handshake corresponds to the remote peer.

SWA-01-010 WP2: Unbounded recursion in file joiner (*Medium*)

While reviewing the file joiner implementation, it was found that the code allows for an unbounded recursive call.

Affected File:
pkg/file/joiner/joiner.go

Affected Code:
```
func (j *joiner) readAtOffset(b, data []byte, cur, subTrieSize, off,
bufferOffset, bytesToRead int64, bytesRead *int64, eg *errgroup.Group) {
[...]
        if subTrieSize <= int64(len(data)) {
         [...]
         return
    }

        for cursor := 0; cursor < len(data); cursor += j.refLength {
         if bytesToRead == 0 {
                break
         }
        [...]

        func(address swarm.Address, b []byte, cur, subTrieSize, off,
                bufferOffset, bytesToRead int64) {
                 eg.Go(func() error {
                        ch, err := j.getter.Get(j.ctx,
                              storage.ModeGetRequest, address)
                        if err != nil {
                                return err
                        }

                        chunkData := ch.Data()[8:]
                        subtrieSpan := int64(chunkToSpan(ch.Data()))
                        j.readAtOffset(b, chunkData, cur, subtrieSpan, off,
                              bufferOffset, currentReadSize, bytesRead, eg)
                        return nil
                 })
         }(address, b, cur, subtrieSpan, off, bufferOffset, currentReadSize)
        bufferOffset += currentReadSize
        bytesToRead -= currentReadSize
```

It can be observed that the function *readAtOffset* is invoked recursively. The recursion can be terminated if the conditions *subTrieSize <= int64(len(data))* or *bytesToRead == 0* hold true. However, the respective variables will not necessarily be decreased by the recursive call: *bytesToRead* is only decremented after the call and *subTrueSize* is controlled by the data from within the chunk. Therefore, an attacker might craft a malicious chunk that (directly or indirectly) refers to itself. This would cause at least a DoS issue, as the recursion would not terminate. As retrieving chunks from the network can incur costs; this might also lead to spending an unforeseen amount of funds.

In order to address this problem, it is recommended to ensure that the traversed data structure is indeed a tree. For preventing cycles, the implementation could keep a list of the already traversed nodes. If a node to be traversed occurs on this list, the implementation should signal an error condition.

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.

SWA-01-007 WP2: Unauthenticated local plain-text storage of node state *(Low)*

During a source code review of the *bee* package it was found that the state store of a *bee* node, which uses the *leveldb* Go package[5], is neither encrypted nor authenticated. When an attacker obtains access to the local filesystem of a *bee* node, s/he is able to read and potentially also change the content of the state storage without further notice from the perspective of the application. Depending on which kind of data the application persists to the state store (like e.g. blocklist information, etc.), this could have a severe security impact on the user.

Affected Files:
bee/pkg/statestore/leveldb/leveldb.go

It is recommended to encrypt and authenticate the contents of databases to protect the data from being tampered with by a malicious third-party. Symmetric encryption with a secret key, hosted within a secure storage of the operating system like keychain for macOS, provides confidentiality to the state store, whereas authentication using message authentication codes (with a secret key from a secure store) provides integrity. The *GoLevelDB Encrypted Storage*[6] project adds data-at-rest encryption for *leveldb.*

SWA-01-011 WP2: Cryptographic material in-memory footprint *(Medium)*

It was found that sensitive cryptographic material, such as private keys, can be obtained by a local attacker from the *bee* process memory. An attacker could potentially use extracted sensitive data from the process memory to perform further attacks.

PoC:
The following demonstrates the steps needed to obtain private key material from a *bee* node by using the *dumpmem*[7] utility:

1. Connect to the *bee* node and open a *bash shell*
2. Run *dumpmem* for dumping the process memory of the "*bee*" process:
```
bee@bee-0:~$ /tmp/busybox-x86_64 ps
```

[5] https://github.com/syndtr/goleveldb
[6] https://github.com/tenta-browser/goleveldb-encrypted
[7] https://github.com/muhzii/procmem-dump

```
PID   USER     TIME   COMMAND
    1 bee       27:02 bee start --config=.bee.yaml

bee@bee-0:~$ /tmp/dumpmem 1
```

3. The *dumpmem* tool will store the output within a file named *procmem.dmp*
4. Search for private key material within the obtained memory dump:

```
$ strings procmem.dmp | vim -
[...]
-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDuLnQAI3mDgey3VBzWnB2L39JUU4txjeVE6myuDqkM/uGlfjb9
SjY1bIw4iA5sBBZzHi3z0h1YV8QPuxEbi4nW91IJm2gsvvZhIrCHS3l6afab4pZB
l2+XsDulrKBxKKtD1rGxlG4LjncdabFn9gvLZad2bSysqz/qTAUStTvqJQIDAQAB
AoGAGRzwwir7XvBOAy5tM/uV6e+Zf6anZzus1s1Y1ClbjbE6HXbnWWF/wbZGOpet
3Zm4vD6MXc7jpTLryzTQIvVdfQbRc6+MUVeLKwZatTXtdZrhu+Jk7hx0nTPy8Jcb
uJqFk541aEw+mMogY/xEcfbWd6IOkp+4xqjlFLBEDytgbIECQQDvH/E6nk+hgN4H
qzzVtxxr397vWrjrIgPbJpQvBsafG7b0dA4AFjwVbFLmQcj2PprIMmPcQrooz8vp
jy4SHEg1AkEA/v13/5M47K9vCxmb8QeD/asydfsgS5TeuNi8DoUBEmiSJwma7FXY
fFUtxuvL7XvjwjN5B30pNEbc6Iuyt7y4MQJBAIt21su4b3sjXNueLKH85Q+phy2U
fQtuUE9txblTu14q3N7gHRZB4ZMhFYyDy8CKrN2cPg/Fvyt0Xlp/DoCzjA0CQQDU
y2ptGsuSmgUtWj3NM9xuwYPm+Z/F84K6+ARYiZ6PYj013sovGKUFfYAqVXVlxtIX
qyUBnu3X9ps8ZfjLZO7BAkEAlT4R5Yl6cGhaJQYZHOde3JEMhNRcVFMO8dJDaFeo
f9Oeos0UUothgiDktdQHxdNEwLjQf7lJJBzV+5OtwswCWA==
-----END RSA PRIVATE KEY-----
[...]
```

It is recommended to protect sensitive cryptographic material in the memory by leveraging frameworks such as *memguard*[8][9]. This will help eliminate the risk associated with an attacker getting a hold of sensitive data structures.

[8] https://github.com/awnumar/memguard
[9] https://pkg.go.dev/github.com/awnumar/memguard

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

SWA-01-012 WP2: Potential directory traversal in download API (*Medium*)

While reviewing the API implementation, it was found that the file download API possibly contains a directory traversal issue.

Affected File:
pkg/api/file.go

Affected Code:
```
additionalHeaders := http.Header{
 "Content-Disposition": {fmt.Sprintf("inline; filename=\"%s\"",
      metaData.Filename)},
 "Content-Type":          {metaData.MimeType},
}

s.downloadHandler(w, r, e.Reference(), additionalHeaders, true)
```

It can be observed that the filename provided in the file's metadata is directly returned in the *filename* part of the *Content-Disposition* header. An attacker might, however, provide file-names such as *../../../../etc/passwd* or similar, which could subsequently be accidentally used by client-code. In order to address this issue, it is recommended to already filter file names in the API (e.g., by applying a function like *basename*).

Conclusions

As noted in the *Introduction,* the Ethereum Swarm software compound, encompassing various pieces of software and smart contracts, has been generally evaluated as well-written and clearly documented. After spending thirty-three days on the scope in spring 2021, six senior testers from the Cure53 team conclude that the quality of the code is good and, in particular, the Solidity parts seem to be quite strong in comparison to that of similar applications seen in the wild.

At the same time, Cure53 must emphasize that the overall codebase seems to be in flux. Certain parts are currently undergoing heavy revisions and need further auditing. Therefore, the fact that twelve discoveries were made should be read in this context as concurrently an early sign that more work is needed, and a good indicator in regard to not exposing serious - *Critical* or *High*-ranking - flaws.

One of the key observations is that the concepts described in the Swarm book do not always reflect the reality found in the codebase. This should be rectified and adapted. Other already implemented concepts are ever-changing and will continue to have this characteristic in the near future as their limitations have already become apparent. The developers are clearly capable of adversarial thinking, though there are areas in the codebase where the approach seems clouded.

The design and implementation overall left a positive impression. The product is rather complex, as it is based on recent research ideas. Generally, it appears that the developers are familiar with common security problems. However, given the research-adjacent character of the solution, fluctuations in the design and implementations appear to be frequent, which can in some cases lead to security issues.

The identified issues are often related to the application logic, in particular to handling edge-cases and unexpected behavior of possibly (malicious) participants. “Generic” security issues - like command injection problems, SQL injection and similar - were clearly less prevalent. This highlights the developers’ good baseline security awareness.

Performing a full, in-depth review of the solution does not seem to be realistic at the current state of development. First, there are areas that are still changing. Second, the solution is large and consists of different individual components that would by themselves already fill the scope for an entire assessment (e.g., *libp2p*). Therefore, the assessment focused on identifying particular areas of interest, such as the unique and custom aspects of the solution, including (but not limited to): game theoretic incentives, smart contracts and binary tree data structures for storing information.

As already emphasized, the analyzed projects and its source code packages left a positive impression and it is evident that the developers are aware of secure programming, best practices and common flaws that are often made by developers when writing Golang applications. The overall codebase and material were a bit overwhelming at the beginning, due to the sheer size and vast complexity. It required a longer run-up time to actually start getting deeper insights into some of the interesting areas.

In terms of review focus, several areas were given the most attention. The libp2p handshake protocol required prior establishment of a connection between two nodes within Swarm. Some weaknesses within the handshake protocol were spotted, allowing to inject bogus underlay information into the handshake (SWA-01-008) or to potentially bypass a node's blocklist (SWA-01-009). The HIVE protocol, used to exchange and broadcast initial information to other *bee* nodes in order to bootstrap connectivity, was also checked. As there is no rate-limiting in place, a malicious node could flood another *bee* node's address book (SWA-01-006).

Other review areas included the HTTP client API, used to upload and download files, as well as some local attack vectors related to *leveldb* storage and the identification of cryptographic material by a local attacker. The HTTP client API lacks a maximum file size check when uploading files into Swarm (SWA-01-004) and uses some insecure Golang routines, potentially allowing malicious users to cause a Denial-of-Service (SWA-01-005). It has to be noted that, according to the client, the HTTP client API is under heavy development (at the time of writing), and is subject to change in the near future.

It was found that local attackers are capable of identifying sensitive information from the *bee* process, as data in memory is not sufficiently protected from such kinds of attacks (SWA-01-011). The state store *leveldb* database is not being protected from local attackers in terms of integrity and confidentiality, therefore making it possible for a local attacker to tamper with data persisted within the *leveldb* file (SWA-01-007).

In terms of review coverage, some areas have only been reviewed in a cursory manner and require additional time and resources. Among them, Kademlia is used to route messages between *bee* nodes using overlay addressing. As already discussed with the client towards the end of this review, it seems that the Kademlia implementation within Swarm lacks some certain best practices, for example, it does not favor long-lived *bee* nodes or short-lived nodes. A thorough review of the Kademlia implementation was not possible as this area has been uncovered almost at the end of this review. As this protocol plays a substantial role in Swarm, it is encouraged to deeply review its implementation. It cannot be excluded that it hosts vulnerabilities that can be abused by attackers to cause severe harm.

In summary, it can be said that a thorough review of all components of Swarm is simply unrealistic within short time-frames of external engagements. Therefore, continuous engagement in security is strongly advised to the Ethereum Swarm team moving forward. Nevertheless, Cure53 can report reaching acceptable coverage on some selected areas during this spring 2021 project, especially in connection to the *libp2p* handshake protocol and the HTTP client API. In this context, not spotting any *Critical/High* flaws is a good sign for the complex, even though the identified twelve weaknesses should be tackled as soon as possible.

Cure53 would like to thank Rinke Hendriksen, Elad Nachmias, Črt Ahlin, Attila Gazsó, Ivan Vandot and Ralph Pichler from the Ethereum Swarm team for their excellent project coordination, support and assistance, both before and during this assignment.