Fine penetration tests for fine websites

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

Pentest-Report Axelar Core & Components 12.2021

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, Dipl.-Ing. D. Gstir, M. Kinugawa, R. Weinberger

Index

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

Introduction

"Empower all ecosystems and dApp developers to build on any blockchain and enable arbitrary cross-chain composability via the Axelar infrastructure stack"

From https://axelar.network/roadmap

This report - entitled AXE-01 - details the scope, results, and conclusory summaries of a penetration test and source code audit against the Axelar core and components, a compound of decentralized networks, protocols, tools, and APIs that facilitates fluid cross-chain communication.

The work was requested by the Axelar Foundation in mid-August 2021 and initiated by Cure53 in late December, namely in CW49, CW50, and CW51. A total of twenty-eight days were invested to reach the coverage expected for this project. Notably, this project marks Cure53's inaugural engagement against the Axelar scope with additional audits scheduled throughout 2022. The testing conducted for AXE-01 was divided into three separate work packages (WPs) for execution efficiency, as follows:

- **WP1**: Sec. Assessments & Code Reviews against Axelar Core & Components
- **WP2**: Cryptography Reviews & Audits against Axelar Core, KMS & Components
- **WP3**: Security Assessments & Code Reviews against Axelar Web Frontend & UI

Cure53 was granted access to extensive documentation regarding the scope, threat model, and expectations; several environments to test against; all relevant sources; and direct collaboration with the development team for Q&A. Given that all of these were necessarily required to procure the coverage levels expected by Axelar, the methodology chosen here was white-box. A team of five senior testers was assigned to this project's preparation, execution, and finalization.

All preparatory actions were completed in early December 2021, namely in CW49, to ensure that the testing phase could proceed without hindrance and to grant a comprehensive understanding of the scope, associated risks, and audit focus areas.

Communications were facilitated via a dedicated shared Slack channel that was deployed to combine the workspaces of Axelar and Cure53, thereby allowing an optimal collaborative working environment to flourish. All participatory personnel from both parties were invited to partake throughout the test preparations and discussions.

One can denote that communications proceeded smoothly on the whole. The scope was well prepared and clear, no noteworthy roadblocks were encountered throughout testing, and the plethora of cross-team queries were conducted concisely and effectively.

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

Axelar delivered excellent test preparation and assisted the Cure53 team in every respect to procure maximum coverage and depth levels for this exercise.

Cure53 gave frequent status updates concerning the test and any related findings, whilst simultaneously offering prompt queries and receiving efficient, effective answers from the maintainers. Live reporting was not requested at first but implemented towards the end of testing. Initially, concise issue headlines were shared to keep the Axelar team in the loop; subsequently, live reporting was requested by the Axelar team and executed by Cure53 via a dedicated ticket thread on Slack.

With regards to the findings in particular, the Cure53 team achieved excellent coverage over the WP1 to WP3 scope items, identifying a total of eleven findings. Positively, none of these findings were categorized as security vulnerabilities; all eleven were deemed general weaknesses with lower exploitation potential. Evidently, the Axelar software compound is a complex entity, thereby one should not consider the lack of significant vulnerabilities as surprising for this opening audit. However, the Cure53 team was able to identify several minor issues that provide leeway for risk in the future. These have been documented throughout this report to enable the Axelar team to address them promptly and prevent their proliferation.

All in all, the volume and assigned severity levels of the findings clearly demonstrate that the Axelar team implements good practices concerning security. The argument can be made that the lack of issues of noteworthy severity is considerably impressive, despite the extensively thorough audits and ample resources with which to conduct them.

The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. Subsequently, the report will list all findings identified in chronological order. Each finding will be accompanied by a technical description and Proof of Concepts (PoCs) where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the Axelar core and components, giving high-level hardening advice where applicable.

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

Scope

- **Security assessments and source-code audits against Axelar core and components**
  - **WP1**: Security Assessments & Code Reviews against Axelar Core & Components
    - Axelar Core
      - https://github.com/axelarnetwork/axelar-core
    - Tofnd
      - https://github.com/axelarnetwork/tofnd
    - Solidity GCP Gateway
      - https://github.com/axelarnetwork/solidity-cgp-gateway
    - Tofn
      - https://github.com/axelarnetwork/tofn
  - **WP2**: Cryptography Reviews & Audits against Axelar Core, KMS & Components
    - See above
  - **WP3**: Security Assessments & Code Reviews against Axelar Web Frontend & UI
    - Web App URL:
      - https://bridge.devnet.axelar.dev/
    - Main UI webapp:
      - https://github.com/axelarnetwork/axelar-web-app
    - JavaScript SDK:
      - https://github.com/axelarnetwork/axelarjs-sdk
    - Rest server:
      - https://github.com/axelarnetwork/axelar-bridge-rest-server
  - **Documentation**
    - Testnet Docs:
      - https://docs.axelar.dev
    - Axelar Whitepaper
      - https://github.com/axelarnetwork/whitepaper
  - **Detailed test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

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

Miscellaneous Issues

This section covers any and all noteworthy findings that did not lead to an exploit but might assist an attacker in successfully achieving malicious objectives 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.

AXE-01-001 WP3: Metrics endpoint discloses access logs *(Info)*

The discovery was made that the REST server exposes the metrics information generated by the *hapi-k8s-health* plugin[1]. The metrics information can be accessed via the following URL.

URL:
https://axelar-bridge-devnet.herokuapp.com/metrics

This page enumerates the accessed URL paths which facilitate sensitive information leakage.

Steps to reproduce:

1. Open the following URL: https://axelar-bridge-devnet.herokuapp.com/**secret-string**
2. Open the following URL: https://axelar-bridge-devnet.herokuapp.com/metrics
3. The latter URL's response will contain the access log to the */secret-string* path, as displayed by the following response.

Response:

```
# HELP http_request_count Total number of http requests
# TYPE http_request_count counter
[...]
http_request_count{method="get",status_code="404",path="/secret-string"} 1
```

The affected code was detected in the following file. As can be deduced, the option to disable the metrics endpoint remains unutilized.

Affected file:
axelar-bridge-rest-server-main/src/Plugins/health/healthPlugin.ts

[1] https://github.com/radenui/hapi-k8s-health

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

Affected code:

```
import {HealthPlugin} from "hapi-k8s-health";

export const healthPlugin = {
	plugin: HealthPlugin,
	options: {
		livenessProbes: {
			status: () => Promise.resolve('Yeah !')
		},
		readinessProbes: {
			health: () => Promise.resolve('ready TODO')
		}
	}
};
```

Even though the confirmation could not be made that information leakage occurred during this test, nevertheless it is recommended to disable the metrics endpoint by optimally configuring the *hapi-k8s-health* plugin option.

AXE-01-002 WP3: Potential XSS via unescaped error message *(Info)*

The discovery was made that the error message returned by the *getTokenAddress* endpoint is insufficiently escaped, which may cause an XSS vulnerability. When the error is thrown, the endpoint displays the parameter value and the exception value of the *try...catch* statement in text/HTML format. Here, the parameter's value is escaped properly; however, the exception value of the *try...catch* statement remains unescaped. Since the exception value can contain user input, the potential for XSS risk persists.

The affected code was detected in the following file and highlighted next.

Affected file:
axelar-bridge-rest-server-main/src/Plugins/client-rest-routes/get/getTokenAddress.ts

Affected code:

```
const moduleParam = escape(request.query.module);
const chainParam = escape(request.query.chain);
const assetParam = escape(request.query.asset);
[...]
try {
    [...]
} catch (e: any) {
	const msg: string = `ERROR GET_TOKEN_ADDRESS for module=${moduleParam},
chain=${chainParam}, asset=${assetParam}, DETAILS: ` + e;
	request.logger.error({
		tag: "GET_TOKEN_ADDRESS",
		message: msg,
```

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

```
            traceId: "TBD"
        });
        ConsoleLogger.logError(msg, request.headers["x-traceId"]);
        return msg;
}
```

Even though no errors were found containing user input in this test, to mitigate any potential risk, it is recommended to additionally utilize the *escape()* function for the error message before displaying it.

AXE-01-003 WP3: Path traversal in *getTokenAddress* endpoint *(Info)*

In the REST server's *getTokenAddress* endpoint, the discovery was made that unsanitized user input was utilized to construct endpoint URLs for server-side requests. However, this scenario could not be exploited in an impactful way during this test, hence the assigned *Info* severity rating. Nevertheless, this should be considered a negative practice and could lead to significant damage in the future.

A PoC request demonstrating the persistent issue is offered below. Note the highlighted section containing the path traversal ("*..%252F*", namely encoded "*../*").

PoC:

- https://axelar-bridge-devnet.herokuapp.com/getTokenAddress?module=**..%252F**query%252Fevm&chain=ethereum&asset=uusd
- https://axelar-bridge-devnet.herokuapp.com/getTokenAddress?module=evm&chain=**..%252F**batched-commands%252Fethereum&asset=uusd

The affected code was detected and highlighted below.

Affected file:
axelar-bridge-rest-server-main/src/Plugins/client-rest-routes/get/getTokenAddress.ts

Affected code:

```
const moduleParam = escape(request.query.module);
const chainParam = escape(request.query.chain);
const assetParam = escape(request.query.asset);
[...]
try {

        tokenAddr = await require("axios").create({
                headers: {
                        'Access-Control-Allow-Origin': '*',
                        'Access-Control-Allow-Methods': 'GET'
                }
```

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

```
        }).get(
                process.env.RPC_TENDERMINT_URL + `/query/${moduleParam}/token-
address/${chainParam}?asset=${assetParam}`
        );
        return {
                data: (tokenAddr?.data as any)?.result?.address as string
        }

} catch (e: any) {
        [...]
}
```

To avoid access to unexpected paths, it is recommended to check if user input comprises expected alpha-numeric characters before concatenating them with the URL.

AXE-01-004 WP3: Absence of general HTTP security headers *(Info)*

The observation was made that the Axelar web application lacks certain HTTP security headers in HTTP responses. This does not directly lead to a security issue, yet might aid attackers in their efforts to exploit other areas of weakness. The following list enumerates the headers that require review to prevent associated flaws.

- **X-Frame-Options**: This header specifies whether the web page is frameable. Although this header is known to prevent Clickjacking attacks, many other attacks can be achieved when a web page is frameable[2]. It is recommended to set the value to either **SAMEORIGIN** or **DENY**.
- Note that the CSP framework offers similar protection to X-Frame-Options via methods that overcome some of the shortcomings of the aforementioned header. To optimally protect users of older browsers and modern browsers simultaneously, it is recommended to consider deploying the *Content-Security-Policy: frame-ancestors 'none';* header in addition.
- **X-Content-Type-Options**: This header determines whether the browser should perform MIME Sniffing on the resource. The most common attack abusing the absence of this header pertains to tricking the browser into rendering a resource as an HTML document, effectively leading to Cross-Site-Scripting (XSS).

All in all, the absence of proper security headers is a negative practice that should be avoided. It is recommended to insert the following headers into every server response, including error responses such as *4xx* items. More broadly, it is recommended to reiterate the importance of deploying all HTTP headers at a specific, shared, and central location rather than randomly assigning them.

[2] https://cure53.de/xfo-clickjacking.pdf

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

This should either be handled by a load balancing server or a similar infrastructure. If the latter is not possible, mitigation can be achieved by deploying the web-server configuration and a matching module.

AXE-01-005 WP1: Absence of file IO result check *(Low)*

Whilst reviewing the *tofnd-main* repository, the discovery was made that writing the entropy file does not ensure that the entire buffer is successfully written to disk. Specifically, *file.write_all()*[3] does not ensure that file contents and metadata are successfully written to disk. Filesystems usually function asynchronously and many errors will only arise when the filehandle is closed. This also includes errors that denote out of space. When the file object is out of scope, rust will automatically close the filehandle. However, Rust's Drop implementation of *std::fs::File* ignores errors of this nature. This can lead to an inconsistent state and facilitate future attacks.

Affected file:
`tofnd-main/src/mnemonic/file_io.rs`

Affected code:
```
pub(super) fn entropy_to_file(&self, entropy: Entropy) -> FileIoResult<()> {
        // delegate zeroization for entropy; no need to worry about mnemonic, it
is cleaned automatically
        let mnemonic = bip39_from_entropy(entropy)?;
        let phrase = mnemonic.phrase();
        // if there is an existing exported file raise an error
        self.check_if_not_exported()?;
        let mut file = std::fs::File::create(&self.export_path())?;
        file.write_all(phrase.as_bytes())?;
        info!("Mnemonic written in file {:?}", &self.export_path());
        Ok(())
}
```

it is recommended to insert *file.sync_all()*?; directly after *file.write_all()*.

AXE-01-006 WP1: Insufficient in-memory protection of secret values *(Info)*

Whilst reviewing the *tofnd-main* repository, the observation was made that the mechanism to safely store the password and other secret values such as keys in memory only ensures that the memory is zeroed when the affected variables persist out of scope. One should consider alternative risks depending on the threat model; most significantly, that modern operating systems will write memory pages to disk, especially in the eventuality that memory pressure becomes excessive.

[3] https://doc.rust-lang.org/std/fs/struct.File.html

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

The persistence of secrets of this nature to disk results in leakage to persistent storage.

This specifically affects the password used to derive the encryption key for the sled K/V store, but also other secrets such as the mnemonic used to derive signing keypairs.

Nevertheless, this issue is marked as *Info* only, since the actual threat model here is not entirely clear concerning the aforementioned issues. Furthermore, the comment "*// TODO use https://docs.rs/secrecy ?"* within the source code indicates that the authors may be privy to this issue already. However, one can also presume that the secrecy crate[4] does not offer protection whilst transferring sensitive data to disk.

Affected file:
`tofnd-main/src/encrypted_sled/password.rs`

AXE-01-007 WP1: Leakage of secrets via crypto libraries *(Low)*

During the *tofn* repository review, the observation was made that particular processes are implemented to delete sensitive data such as passwords and secret keys from memory immediately after use. The approach used in the code is provided via the *Zeroize* trait from the crate *zeroize*[5]. This approach fails, however, when secrets are handed over to third-party libraries, which do not take diligent measures of this nature and copy secrets to alternative memory regions. One instance of this is the HMAC in *tofn-main/src/crypto_tools/rng.rs*, whereby *rng_seed_ecdsa_signing_key()* leaks the value of *secret_recovery_key* (i.e. the secret to generating *all* signing key-pairs) via the HMAC implementation from the HMAC crate.

Affected files:
`tofn-main/src/crypto_tools/rng.rs`
`rust hmac crate (see https://github.com/RustCrypto/MACs/blob/hmac-v0.11.0/hmac/src/lib.rs#L118-L168)`

Affected code:

```
fn new(key: &GenericArray<u8, Self::KeySize>) -> Self {
        Self::new_from_slice(key.as_slice()).unwrap()
    }

    #[inline]
    fn new_from_slice(key: &[u8]) -> Result<Self, InvalidKeyLength> {
        let mut hmac = Self {
            digest: Default::default(),
            i_key_pad: GenericArray::generate(|_| IPAD),
            opad_digest: Default::default(),
```

[4] https://crates.io/crates/secrecy/
[5] https://crates.io/crates/zeroize

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

```
        };

        let mut opad = GenericArray::<u8, D::BlockSize>::generate(|_| OPAD);
        debug_assert!(hmac.i_key_pad.len() == opad.len());

        [...]
        if key.len() <= hmac.i_key_pad.len() {
            for (k_idx, k_itm) in key.iter().enumerate() {
                hmac.i_key_pad[k_idx] ^= *k_itm;
                opad[k_idx] ^= *k_itm;
            }
        } else {
            [...]
        }

        hmac.digest.update(&hmac.i_key_pad);
        hmac.opad_digest.update(&opad);

        Ok(hmac)
}
```

While the code of *rng_seed_ecdsa_signing_key()* takes care to avoid copying the value of *secret_recovery_key*, the implementation of the *Hmac* type (see code listing above) then XORs the key with the *IPAD* constant and stores it in *Hmac.i_key_pad*. Since the *IPAD* constant is well known, XOR is thus simple to revert. As a result, this will leak the value of *secret_recovery_key* after the instance of *Hmac* is dropped upon exiting *rng_seed_ecdsa_signing_key()*.

As can be deduced via the code above, this issue is only triggered when the length of the key is less or equal to the output size of the hash function. The call path from *tofnd*'s *multisig* module will trigger this, since *secret_recovery_key* contains the entropy from the BIP39[6] 24-word mnemonic, which constitutes 256 bits of entropy. The output size of SHA256 is also 256 bits.

The authors of all RustCrypto's crates allude to their awareness of said issue as indicated by GitHub issue 87[7].

In the eventuality that leaking secrets via memory is a valid attack scenario in the Axelar thread model, one should deem it necessary to ensure that all code - including library code - wipes secrets after usage. Additionally, other potential scenarios that could incur leakage from memory, such as that noted in AXE-01-006, may need to be taken into consideration.

[6] https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
[7] https://github.com/RustCrypto/hashes/issues/87

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

AXE-01-008 WP2: ECDSA signature nonce generation permits all-zero value *(Low)*

The *multisig* logic in *axelar-core* and *tofnd* utilizes the ECDSA functions provided by *tofn* to sign messages. *tofn* itself uses the Rust crate *ecdsa*[8] as the underlying base implementation. The audit of the signing logic in *tofn* demonstrated that a custom algorithm is used to generate the factor *k,* which is utilized during the signing of messages (see RFC 6979 section 2.4[9]). Per ECDSA standard, *k* must be nonce - a random number that must be kept secret and never reused.

RFC 6979 specifies an approach for a deterministic method to generate *k* from the message to be signed and the (secret) signing key. *tofn* implements a similar approach as RFC 6979 in functions *rng_seed_ecdsa_ephemeral_scalar* and *rng_seed_ecdsa_ephemeral_scalar_with_party_id*. While this construction used by *tofn* seems secure, it fails to check if the generated value for *k* contains an all-zero value. As per ECDSA standard, a value of all zeros for *k* is not permitted. Granted, the likelihood of this occurring is extremely low; nevertheless, best practices should be implemented to ensure its impossibility.

Additionally, it should be mentioned that non-standard constructs such as those implemented in *tofn* are also prone to unnoticed vulnerabilities since they have never been reviewed by cryptographic experts. As mentioned above, the auditors could not find any issues during the review of *tofn*, but in this case, a vulnerability would be considered critical since this is one of the most sensible and error-prone components of ECDSA. High-profile cases such as Sony's PlayStation 3 vulnerability demonstrated that knowledge of the value of a single bit of *k* is all one requires to derive the private key, as documented by Aranha et al's "LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage"[10] article. The recommendation can therefore be made to implement an airtight approach to generating *k* as specified in RFC 6979, for example.

Affected file:
`tofn-main/src/crypto_tools/rng.rs:104-153`

[8] https://crates.io/crates/ecdsa
[9] https://datatracker.ietf.org/doc/html/rfc6979#section-2.4
[10] https://eprint.iacr.org/2020/615.pdf

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

AXE-01-009 WP2: Mnemonic compromise constitutes single point of failure *(Medium)*

Whilst reviewing the key generation flow in *tofnd*, the observation was made that a BIP39-mnemonic is used to generate entropy of 256 bits. This is then used as a seed to generate every ECDSA signing keypair of this *tofnd* instance (i.e. node). The variable naming in *tofnd* indicates that deriving all keypairs from the same secret is intended as a recovery mechanism. Since this key generation process also includes a unique key identifier (keyUID) of at least 32 bits, the individual keypairs differ as long as the keyUID does so too. This is also enforced in *tofnd*, which detects duplicate keyUID usage as long as the on-disk K/V store is available. A side effect of reusing the same 256 bits of entropy, however, is that its successful compromise is fatal to the security of all generated keys and associated operations. This owes to the fact that any attacker with knowledge of the secret mnemonic would be able to regenerate all previously generated keypairs. This is possible because the keyUID is a public nonce.

This scenario should be considered significantly critical in attack scenarios whereby compromise of the mnemonic remains unnoticed. In such cases, an attacker would be able to generate any key pair used in the future as long as they are privy to the keyUID. Since the keyUID is a public nonce, this is easily possible. This essentially means that the current design fails to achieve forward secrecy.

Nevertheless, the overall design of the Axelar network mitigates this issue's severity since a consensus among multiple nodes is required to cause tangible harm. Thus, an attacker would be required to compromise the mnemonic of enough nodes to achieve voting power.

Affected files:
tofnd-main/src/multisig/keygen.rs

Affected code:

```
impl MultisigService {
	pub(super) async fn handle_keygen(&self, request: &KeygenRequest) ->
TofndResult<Vec<u8>> {
	let secret_recovery_key = self.kv_manager.seed().await?;

	let key_pair = keygen(&secret_recovery_key, request.key_uid.as_bytes())
		.map_err(|_| anyhow!("Cannot generate keypair"))?;

	Ok(key_pair.encoded_verifying_key().to_vec())
	}
}
```

The highlighted line in the code listing above fetches the same mnemonic entropy from the encrypted persistent storage and uses it to generate key pairs.

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

It is also worth mentioning that the mnemonic phrase is written to disk in plain text upon generation. The README of *tofnd*, however, states: “The mnemonic entropy is stored in clear text on disk. Our current security model assumes secure device access.” Thereby, it is assumed that the thread model excludes disk access by an attacker.

It is recommended to ensure that the mnemonic entropy is rotated frequently. Currently, this would constitute a manual process of restarting *tofnd* and generating a new mnemonic. Feedback from the developers revealed that processes are being formulated to allow users to automate this more easily in the near future.

AXE-01-010 WP1: Unhandled error in *axelarnet confirm-deposit* command *(Info)*

During a source code review of the *axelar-core* repository, the observation was made that the handler responsible for processing the *axelard tx axelarnet confirm-deposit* CLI and REST API command fails to return an error early when decoding the transaction ID using *hex.DecodeString()*[11].

One can pertinently note that the impact of this can be neglected as the *ValidateBasic()* routine for the *ConfirmDepositRequest* message validates that the transaction ID constitutes a 32-byte length before returning from the command handler. Therefore, this issue is solely listed for completeness to stress the importance of consistently validating all input and failing on error.

Affected file:
axelar-core-main/x/axelarnet/client/cli/tx.go

Affected code:

```
func GetCmdConfirmDeposit() *cobra.Command {
	cmd := &cobra.Command{
		[...]
		RunE: func(cmd *cobra.Command, args []string) error {
			[...]

			txID, err := hex.DecodeString(args[0])
			coin, err := sdk.ParseCoinNormalized(args[1])
			if err != nil {
				return err
			}
			burnerAddr, err := sdk.AccAddressFromBech32(args[2])
			if err != nil {
				return err
			} [...]
```

[11] https://pkg.go.dev/encoding/hex#DecodeString

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

Affected file:
axelar-core-main/x/axelarnet/rest/tx.go

Affected code:
```
func TxHandlerConfirmDeposit(cliCtx client.Context) http.HandlerFunc {
	[...]
	txID, err := hex.DecodeString(req.TxID)

	coin, err := sdk.ParseCoinNormalized(req.Amount)
	if err != nil {
		rest.WriteErrorResponse(w, http.StatusBadRequest, err.Error())
		return
	}
	[...]
}
```

It is recommended to consistently validate all arguments and return an error when *args[0]*, the actual transaction ID, cannot be decoded and *hex.DecodeString()* returns an error.

AXE-01-011 WP1: Absence of validation in EVM *confirm-deposit* command *(Info)*

During a source code review of the axelar-core repository, the observation was made that the handler responsible for processing the *axelard tx evm confirm-erc20-deposit* CLI and REST API command fails to validate the transaction ID read from the CLI arguments or HTTP request with regards to length.

One can pertinently note that the tangible impact of this issue could not be evaluated to the extent envisioned, as this issue was identified at the culmination of testing. One could make an educated guess that the provision of an invalid transaction ID would be detected at a later stage and deeper within the source code. As one of the primary objectives of this security audit pertained to the review of all input-parameter validations, this issue was listed for completeness reasons.

Affected file:
axelar-core-main/x/evm/client/cli/tx.go

Affected code:
```
func GetCmdConfirmERC20Deposit() *cobra.Command {
	cmd := &cobra.Command{
		Use:   "[...]",
		Short: "[...]",
		Args:  cobra.ExactArgs(4),
```

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

```
                RunE: func(cmd *cobra.Command, args []string) error {
                        [...]

                        chain := args[0]
                        txID := common.HexToHash(args[1])
                        amount, err := sdk.ParseUint(args[2])
                        if err != nil {
                                return fmt.Errorf("given amount must be an integer
                                value, make sure to convert it into the appropriate
                                denomination")
                        }

                        burnerAddr := common.HexToAddress(args[3])

                        msg := types.NewConfirmDepositRequest(
                                cliCtx.GetFromAddress(), chain, txID,
                                amount, burnerAddr)
                        if err := msg.ValidateBasic(); err != nil {
                                return err
                        }

                        [...]
}
```

The *msg.ValidateBasic()* routine of *ConfirmDepositRequest* fails to validate the length of the provided transaction ID.

Affected file:
axelar-core-main/x/evm/types/msg_deposit.go

Affected code:

```
func (m ConfirmDepositRequest) ValidateBasic() error {
        if err := sdk.VerifyAddressFormat(m.Sender); err != nil {
                return sdkerrors.Wrap(
                        sdkerrors.ErrInvalidAddress,
                        sdkerrors.Wrap(err, "sender").Error())
        }

        if m.Chain == "" {
                return fmt.Errorf("missing chain")
        }

        return nil
}
```

It is recommended to consistently validate all arguments and return an error in the eventuality that the provided arguments are incorrect.

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

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW49, CW50, and CW51 testing against the Axelar core and components by the Cure53 team - will now be discussed at length. To summarize, the confirmation can be made that the components under scrutiny have left a positive impression.

The assessments featured the *axelar-core*, *tofn*, *tofnd-main,* and *solidity-cgp-gateway* repositories, with a focus on identifying particular areas of interest such as the unique and custom aspects of the solution. This included, but was not limited to: correctness of cross-chain asset transfers; input validation and correctness of supported commands; double-spending attacks; leakage of keys; malicious injection of requests that would lead to funds loss; the liveness (availability) of the protocol; and the smart contracts.

The code of all work packages in scope is well commented and it is evident that the developers are aware of and implement secure programming best practices. However, due to the vast complexity of the software compound, extensive efforts were required to understand the method by which all components interact and collaborate. The software complex utilizes *cosmos-sdk* as a framework for the solution within the *axelar-core* repository, offering several means by which to interact with Axelar, including a command-line, REST, and gRPC interface.

Axelar has evidently already made effective strategic decisions by utilizing Rust and Golang as technologies and programming languages. Both provide a multitude of benefits towards secure programming, for instance relating to compiled languages such as C/C++. Concerning the findings on the whole, no vulnerabilities were uncovered and all identified issues were miscellaneous in nature. These mostly pertained to implementation issues, which should not be considered unusual for a codebase of this magnitude.

WP1 included a security assessment of the repositories in scope. A heightened focus was bestowed upon input data validation in the *axelar-core* repository, particularly any input data received through the REST API and gRPC endpoints. The modules were located in the subfolders of the /x folder. In addition, the handler implementations of the subfolder *cmd/axelard/cmd/vald/* were assessed for absent validation. The authentication of all interfaces relies on the *cosmos-sdk* Ante handler, which made a solid and secure impression; the additional Ante handlers for authorization did not highlight any security issues. Furthermore, the primary workflows in *axelar-core* were analyzed for security issues, though these likewise all made a robust impression.

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

The *tofn* and *tofnd* repositories comprise soundly-written Rust code which avoids error-prone programming patterns such as unsafe code and dynamic borrow checking. The codebase utilizes a defensive coding style and duplicated error checks to reduce the attack surface to an absolute minimum.

The *tofnd* repository was also examined for security issues. This repository corresponds to a gRPC wrapper encompassing the *tofn* library, which initiates the cryptographic operations of a validator node. The communication with *tofnd* is plain and unauthenticated; however, it appears to be limited to local host-only connections. One must emphasize that no sensitive information is communicated over the gRPC channel. Both *tofnd* and *tofn* are implemented within Rust in a visibly transparent and secure way. However, the highest severity-rated issue was identified within this repository and related to the compromise of the *tofnd* mnemonic. A compromise of this nature would have severe consequences since all the keys derive from the mnemonic in a deterministic manner.

Regarding the *tofn* repository, only the ECDSA implementation and code related to the *multisig* logic were in scope for this audit. These were assessed for common vulnerabilities and implementation issues of cryptographic primitives, such as sensitive information leakage or side-channel attacks. The code utilizes third-party libraries for the base ECDSA and secp256k1 algorithms. Notably, the secp256k1 repository specifically contains a warning that this code had not yet received an independent audit. While attempts were made to ensure constant-time operations, no guarantees were given by the developers of this library.

This current Axelar audit engagement did not extend to the secp256k1 Rust crate in full depth, as the allotted timeframe was not sufficient for a deep dive into all third-party dependencies. Worthy of mention here is the fact that one finding within *tofn* (as documented in ticket AXE-01-008) pertained to the ECDSA signature process - specifically, the nonce generation code. This is considered one of the most sensitive components of ECDSA and was the cause for multiple significant vulnerability issues in the past, such as the PlayStation 3 fail0verflow hack. Thus, best practice purports to integrate the algorithm specified in RFC 6979 for these purposes. Similarly, the code in *tofn* uses a custom logic that is based on RFC 6979 but missed an (albeit rather unlikely) all-zero result check.

Whilst *tofnd* and *tofn* attempt to ensure that sensitive data is wiped from memory, they occasionally fail in relation to third-party libraries and the transfer of program memory to disk by OS, for example.

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

The focus points for the smart contracts included typical vulnerability classes such as arithmetic issues (integer overflows etc.), reentrancy bugs, usage of weak cryptographic primitives, misuse of cryptographic primitives, unchecked external calls, and frontrunning.

Naive fuzzing over the network, targeting exposed ports of the validators and other node components of the Axelar ecosystem such as the gRPC interface, was attempted within a local setup whereby all components operated within dedicated Docker containers. Positively, no issues were detected during these initial fuzz tests. In order to increase test coverage and uncover vulnerabilities located within deeper areas of the source code, it is highly recommended to incorporate fuzz testing into the Axelar codebase itself by utilizing *fuzz*[12] *or gofuzz*[13] for *axelar-core*, for instance.

Elsewhere, client-side issues such as DOM-based XSS; the REST server; and all *hapi* framework plugin features implemented were assessed. The lack of issues across all of these components corroborates the argument that the application is correctly implemented with the React framework and that user input has been processed with due diligence.

Regarding the findings, only several minor issues were discovered. The *hapi-k8s-health* plugin exposed the metrics information, which could potentially lead to information leakage (see AXE-01-001 for further information). On one endpoint, values that may contain user input were not properly escaped, which could lead to potential XSS (see AXE-01-002). Although this situation could not be abused in practice, a problem was identified that allowed path traversal, as documented in ticket AXE-01-003.

Generally speaking, the outcome of this source code review demonstrates that the Axelar software complex has already laid solid foundations from a security perspective, which is also reflected by the lack of exploitable vulnerabilities identified. One can only assume that a plethora of code-base audits have already been conducted prior to this engagement.

To summarize, excellent coverage was achieved over all work packages by the testing team. Nevertheless, Cure53 would like to emphasize that the overall codebase appears in a state of flux; certain components are currently undergoing heavy revisions and require further auditing.

[12] https://github.com/dvyukov/go-fuzz
[13] https://github.com/google/gofuzz

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

Therefore, the Axelar software complex could profit from recurrent security assessments moving forward, as the immense complexity of all working packages and components is challenging to handle with each and every new framework iteration. Likewise, alterations installed within one area of the system may have an exponential security impact across the entire compound, thereby requiring more comprehensive assessments to identify hitherto unforeseen areas of weakness.

Cure53 would like to thank Sergey Gorbunov, Milap Sheth, Gus Gutoski, Canh Trinh, Sammy Liu, Talal Ashraf, and Christian Gorenflo from the Axelar Foundation team for their excellent project coordination, support, and assistance, both before and during this assignment.