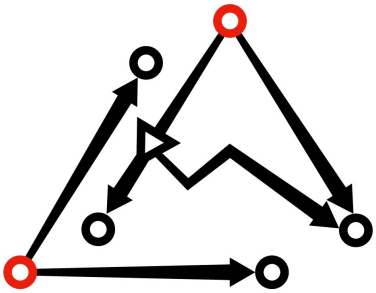

Least Authority

PRIVACY MATTERS

Tezos Protocol

Final Security Audit Report

Tezos

Initial Report Version: 16 March 2019

Table of Contents

Overview

The Tezos Foundation has requested that Least Authority perform a security audit of the Tezos project codebase, including the wallet and website wallet integration, in preparation for the upcoming betanet and mainnet launches. This was the second audit to be started on this code, another review has been undertaken by Inria.

The Least Authority audit was performed from April 16 to May 16, 2018 by Jean-Paul Calderone, Gordon Hall, Ramakrishnan Muthukrishnan, James Prestwich and Dominic Tarr. Slack was used during the audit to communicate with the Tezos team. The initial report was delivered on May 18, 2018 and an updated report following an initial verification phase was delivered on December 14, 2018. A final report following a secondary verification conducted based on the responses provided by Tezos *(Tezos Foundation Comments on the Least Authority Protocol Security Audit Report, 26 February 2019)* was issued on March 16, 2009.

Coverage

Target Code and Revision

The following code repositories and project components are considered in-scope:

- Tezos repository: https://github.com/tezos/tezos
- Tezos wallet and wallet integration with the Tezos website
- PVSS branch: https://gitlab.com/tezos/tezos/tree/pvss_lib

Specifically, we examined the Git revisions:

`11ad8fa8ea068ae9ebee23af4e79bf49ac3f7653`

All file references in this document use Unix-style paths relative to the project's root directory.

Areas of Concern

For this audit, we looked at the following areas and for any similar potential issues:

- The Network layer, including adversaries crafting large or malformed messages to crash, DOS or isolate peers;
- Transaction handling, including incorrectly verifying that transactions are valid, not applying them correctly;
- The Protocol level DOS, including ability to impose arbitrary large bandwidth or computational costs on the network, shutdown the network and consensus-related attacks;
- The use and binding of cryptographic libraries, sources of entropy;
- The CLI client's security, including RPC, can't be accessed from a website, keys are generated correctly and stored encrypted;
- The Consensus algorithm robustness to malicious bakers, selfish mining;
- Private keys and leakage;
- Consensus and anything causing the breakdown and double spending;
- Any bugs leading to loss of funds; and
- `pvss_lib` branch, including a review of the crypto library implementation.

Findings

Language Selection

The Tezos developers have chosen to implement Tezos in the OCaml programming language. The benefit of this selection is that OCaml is a strictly typed language. The drawback of this selection is that it is a less widely used language and both obscures and abstracts certain actions, which may result in more ad hoc code reviews or less reviews overall. We believe this will be offset by the Tezos Foundation's support of more developers learning OCaml. We also recommend that the Tezos project developers continue to reach out and actively pursue regular audits of the code, along with proactive interaction with the broader security community and leaders in governance and voting analysis.

Tezos Response

We agree that OCaml has many technical advantages, particularly for security-critical domains such as cryptocurrency, and that its less widespread adoption is a concern. We have undertaken many efforts to increase both our visibility within the OCaml community and the number of developers and auditors familiar with OCaml. We also intend to continue to engage with auditors and the security community, particularly in the more novel and/or security-critical aspects of the system.

Code Quality

Overall, the code is of good quality. Although an early release of the code, it is clear that effort was directed towards making the code logical and understandable. With more testing, documentation and logistical planning, later versions of the code will reflect substantial improvement facilitated by this earlier effort.

Code Organization

While the code is generally well-organized, there is an opportunity for improvement by removing the duplication of definitions and encoding suggestions to increase readability. This will make it easier for future contributors, and most importantly auditors, to be able to focus efforts on the security risks and vulnerabilities in the code.

For example: Despite the relative documentation defining the Tezos protocol as a bundle of OCaml files, Tezos has its own smart contract language, Michaelson. It may be advantageous to write the protocol (aka, the "constitution") to be a contract, too. This design could remove unnecessary complexity and therefore decrease the attack surface.

Test Coverage

We recommend increasing test coverage of the code base, along with implementing property-based testing to decrease the lines of test code required.

Documentation

In addition to the Position Paper and the OCaml implementation details, the development of technical documentation would help in bridging the gap between what currently exists by connecting the Position Paper to the code base. In addition, regular updates to the Position Paper and Technical Paper to match the most current code implementation is recommended.

Tezos Response

In general, we have been working on these improvement areas since the betanet release (June 2018) as part of the general lifecycle of software, now that the team size has increased. We will periodically

refactor code to make it more readable and maintainable. Increasing test coverage is a particular area of effort. Documentation is recognized as an area which needs to be expanded, and we've engaged multiple teams to improve the available documentation and examples both in the source code and externally.

Issues

We list the issues we found in the code in the order we discovered them.

ISSUE / SUGGESTION	STATUS
Issue A: Peer Authentication Vulnerable to Replay Attacks	*Resolved*
Issue B: Misauthenticated Peers Can Replay Message Packets	*Resolved*
Issue C: Proof of Work is Weak Mechanism to Authenticate a Peer	*Partially Resolved*
Issue D: Tezos Server is Vulnerable to DNS-Rebinding	*Unresolved*
Suggestion A: Explicitly Disallow Cross Origin RPC Access	*Unresolved*
Suggestion B: Use OS System Calls for Random Number Generation	*Resolved*

Issue A: Peer Authentication Vulnerable to Replay Attacks

Synopsis

An unaltered connection packet received from **Peer A** can be sent to other peers who will then believe they are connecting to **Peer A**.

Impact

High, may lead to DOS or eclipse attacks.

Preconditions

The attacker simply needs to connect to a peer and receive a packet, or a passive observer can collect the packets.

Feasibility

Easy.

Technical Details

The handshake is very simple (when not considering some irrelevant fields) where each peer sends a Public Key and a Nonce.

For example: Alice sends `a_pk`, `a_nonce`, and Bob sends `b_pk`, `b_nonce`. Then each derives the encryption key to be used for the channel: `channel_key = combine(b_pk, a_sk)`. The same channel key is used for *every connection* between *any given pair* of peers. Each peer then encrypts packets using the nonce they picked, and decrypts packets using the nonce the other picked.

Since the same key is used each time, anyone who has received a valid connection packet can replay it to other peers, and it will appear to be connected as that peer. The attacker doesn't know the channel key, but they can open a connection and send the connection packet which will be accepted.

Since the attacker only needs to send a single packet, they do not even need to pay the cost of a full TCP connection, they can just send the initial packet, and let the victim allocate a TCP connection.

Remediation

See the remediation for *Issue B: Misauthenticated Peers Can Replay Messgae Packets*.

Tezos Response

Following the potential denial of service issue found by Least Authority about bogus handshake, the negotiation code has been changed, thereby eliminating this as a potential attack. Particularly, a fix to the predictable nonce was committed quickly, and random nonce is generated for each connection attempt. We discussed the idea of ephemeral session key, but it didn't seem useful in our context where the future secrecy of the exchanges is not critical. A very simplified view of the p2p handshake protocol is described by the following pseudocode. Potential errors in this protocol could be caused by the simplification process, please also refer to the authenticate method in the actual code.

```

```

```
authenticate(id, fd) {
      /* random seed for each connection attempt */
      local_nonce_seed = random_nonce ();

      /* prepare a nonce generated from public key and a fresh nonce */
      sent_msg = { id.pk; local_nonce_seed }

      /* sends this nonce */
      socket_send (fd, sent_msg)

      /* receives the message from peer */
      recv_msg = socket_recv(fd);

      /* generates nonce from the received and generated messages */
      local_nonce, remote_nonce = generate_nonces(sent_msg, recv_msg);

      /* stripped: checks identity */
      /* stripped: check proof of work difficulty */

      channel_key = dh(id.sk, );
      box = (channel_key, local_nonce, remote_nonce);
      return (fd, box)

}

send(fd, box, msg) {
      c = enc(box.remote_nonce, box.channel_key, msg);
      socket_send(fd, c);

}

recv(fd, box, peer) {
      c = socket_recv(fd);
```

```
        msg = decrypt(box.local_nonce, box.channel_key, c);
        return msg;

}
```

Furthermore, this handshake protocol was formally verified to be correct against known attacks with the proverif tool, in a joint work with Bruno Blanchet (Prosecco team).

Status

The nonces used now depend on the hashes of both messages sent. If the client resends an old nonce, the server will still use a new one, and send packets encrypted differently.

Pseudocode:

```
generate_nonces (sent_msg, recv_msg) {

  return hash(sent_msg + recv_msg + “Init -> Resp”), hash(recv_msg + sent_msg
+ “Resp -> Init”)

}
```

Verification

Resolved.

Issue B: Misauthenticated Peers Can Replay Message Packets

Synopsis

After exploiting *Issue A: Peer Authentication Vulnerable to Replay Attacks*, a malicious peer can replay encrypted packets received on a previous connection, even without decrypting them.

Impact

DOS attack.

Preconditions

A malicious peer can obtain valid encrypted packets by exploiting *Issue A: Peer authentication vulnerable to replay attacks*.

Feasibility

Easy.

Technical Details

After the handshake, all message packets sent from **Peer A** to **Peer B** are encrypted with the channel key (derived from **A**'s and **B**'s identities) and the nonce *chosen by **A***. If a malicious **Peer M** connects to **B**, they will receive a valid connection packet for **B**. They can then replay that packet to **A**, and will receive a hello packet for **A**, and likely several message packets intended for **B**. They can then replay all of those packets to **B**, and **B** will believe **A** has connected and is making valid requests, **B** will respond to these with valid packets, which could then be replayed to **A**.

Remediation

To prevent replay attacks, the `channel_key` and nonce should be cryptographically guaranteed to be fresh. A peer cannot be sure that a remote peer has genuinely created a fresh random value, but they can be confident that their own random value is fresh because they just generated it. The two nonces should be combined in such a way that if at least one is fresh then the output is fresh.

For example:

```
fresh_local_nonce = hmac(local_nonce, remote_nonce); fresh_remote_nonce =
hmac(remote_nonce, local_nonce)
```

Likewise, for **A** to be certain they are talking to **B**, they need to know not only **B**'s public key, but a proof that the remote peer knows the corresponding private key. A simple proof that would suffice is a value encrypted to an ephemeral key that **A** creates for this connection. For example, if **A** sends an ephemeral key A and then **B** replies with a packet containing something encrypted with `combine(a_pk, B_sk)`, **A** can now be sure that **B** must know `B_sk` since **A** knows that **A** is fresh and **B** does not know it, so therefore, they must know `B_sk`. As with the nonces, we want each connection to have a unique key. If each peer also sends a freshly generated ephemeral key, then the `channel_key` is derived from it.

For example:

```
channel_key = hash(combine(eph_a_pk, eph_b_sk) + combine(eph_a_pk, b_sk) +
combine(a_pk, b_sk))
```

If a malicious peer replayed a hello packet from **B** to **A**, **A** would use a new ephemeral key this time, so the `channel_key` would be different. **A** would have to receive a valid packet from **B** to be sure **B** was able to correctly derive `channel_key` so it is necessary to add at least a second packet to the authentication. By the time that message packets are being sent, authentication is known to be complete.

See also, notes on *Secure Handshakes* section in the *Areas for Further Discussion*.

Tezos Response

We have corrected the handshake protocol, eliminating this potential denial of service attack vector.

Status

This issue has been addressed by a correction of the handshake protocol.

Verification

Resolved.

Issue C: Proof of Work is Weak Mechanism to Authenticate a Peer

Synopsis

Proof of work (PoW) required by peer connection is not an impediment to network attacks.

Impact

Possible DOS attack.

Preconditions

Attacker with some resources that wishes to DOS Tezos network.

Feasibility

Easy.

Technical Details

A peer calculates a Proof of Work for a` public key, `proof = blake2b(pb+nonce,` until it finds a proof value that has at least a target number of zeros. The default number of zeros is 24, which can be accomplished within a few seconds with not much computing power required (such as an older model laptop). Since the proof of work need only be generated once per identity, any attacker controlling a botnet, or similar, could easily just wait for each peer to generate keys. Even if they are paying for the necessary computation, the cost is minimal. For example, with a month's worth of compute with an Amazon EC2 instance (about $5 a month) the attacker could easily generate hundreds of thousands of valid proofs. And if multiple computers are run in parallel, the computations could be completed in a much shorter time.

Mitigation

The default target PoW can be raised, although this inconveniences honest participants in the network.

Remediation

An alternative approach would be to use Proof of Stake instead of Proof of Work. This would be significantly inconvenient for attackers and would allow for traceability of the attackers via their utilized stakes.

If Proof of Work is preferred, then using a more memory hard algorithm such as *scrypt*. To note: libsodium has a function to convert Ed25519 keys, used for signing, to Curve25519, used to derive encryption keys and as peer identity.

Tezos Response

This potential denial of service is addressed by increasing the level of difficulty as warranted by network conditions. We are planning to replace the existing algorithm with a more cpu+ram difficult challenge (such as scrypt) on the next update of the p2p layer. We may investigate other forms of DoS resistance for peer connections at various layers of the system. Requiring proof of stake for connections presents some usability issues, but this is an area of active exploration.

Status

A mitigation for this issue has been applied. However, Least Authority recommends using an asymmetric work function. Although this is a new area of research, equihash is currently a good candidate. Least Authority recently reported a DoS vector in another audit in a similar identity PoW system that used scrypt. The problem was that in order to verify a single identity, a whole round of scrypt was still required, which uses a significant amount of ram and takes 50ms (on a low end laptop) but this effort must be spent even to invalidate a random invalid identity. This means an attacker is able to send just 20 random identities per second and have the ability to completely stall a victim.

Verification

Partially resolved.

Issue D: Tezos Server is Vulnerable to DNS-Rebinding

Synopsis

Using a attacker controlled DNS server, a locally open web page can successfully call RPC methods.

Impact

If you open the attacker website on the same device as the Tezos node, it can call any RPC method, including things that interfere with networking or gather private data.

Preconditions

Attacker with DNS server.

Feasibility

Easy.

Technical Details

The attacker hosts a website on a server with a DNS server that they control and very low Time-To-Live settings. On the first request, the attack site is loaded before a script on that page makes another request to that site. At this point, the DNS server returns 127.0.0.1 resulting in the same origin policy thinking it's talking to the attack site, while it's actually talking to the local host.

Mitigation

Users should not run the Tezos nodes and a browser at the same time on the same machine.

Remediation

Tezos RPC server should check that the host header is as expected, or as is allowed via a cors setting. It would be optimal and recommended to have a local RPC password, as other blockchain systems do.

Tezos Response

This issue is remediated by changes to the Tezos RPC server. While the usual deployment model of tezos nodes is isolated and thus protected from this kind of attack, it is a concern with systems used in development or by some end users. We implemented host header/cors checks and are investigating options for local RPC authentication.

Status

This issue is not yet fixed. Tezos has informed us that a fix is planned by adding mutual authentication between node and clients. Currently, Tezos users should run their Tezos node on a different machine to their web browser; the Tezos development team informs us that this is the way Tezos is normally run. We advise making the expectation that tezos is run on a separate machine as explicit as possible.

Verification

Unresolved.

Suggestions

We list the issues we found in the code in the order we discovered them.

Suggestion A: Explicitly Disallow Cross Origin RPC Access

Synopsis

Accessing the Tezos local server from a website is not explicitly avoided in the Tezos codebase. It happens that Tezos is not vulnerable to access by a locally open website, but only as a coincidence of

several factors. It is recommended to have an explicit check. **Note**: Issue D, which was identified during the verification phase is far more impactful, but equates the same amount of risk posed.

Technical Details

All Tezos *http rpc* actions are by `POST`, which causes browsers to send an `OPTIONS` preflight if content type header is set to `application/json`. If `application/json` is not set, the Tezos server sends an 40x error. It is conceivable that someone updates the code to relax some of these (for example, in a pull request to add a local Web UI) and thus, opening this vulnerability.

Remediation

There should be an explicit check that host header is nothing other than what is expected, and along with this, tests should be written.

Tezos Response

While this issue is not a current problem (due to the other identified factors preventing exploitation), we will explicitly protect against it to be defensive in case of future changes in the codebase. We are preparing a local RPC pairing to explicitly allow RPC only with authorized clients and html5 local applications.

Status

While there are significant mitigating factors in place, this issue remains unfixed and we maintain our recommendation for remediation.

Verification

Unresolved.

Suggestion B: Use OS System Calls for Random Number Generation

Synopsis

The TweetNaCl code uses `/dev/urandom` as the source for random numbers. If this device file cannot be opened for some reason then an indefinite number of retries are performed. This may lead to the program busy-looping indefinitely in some situations.

Remediation

Tezos may want to use OS specific system calls (for example, getrandom(2) on Linux kernel).

Tezos Response

This issue has been addressed by switching to HACL. We will make a note of this issue as something to check when using random numbers generally.

Status

`7d6da7179b63b8b77dc30da5d66f918c3de3980f` removed the use of TweetNaCl and replaced it with HACL. HACL appears unaffected by this issue.

Verification

Resolved.

Areas for Further Discussion

These are significant areas for improvement, but do not have specific vulnerabilities identified, at this point.

Fuzzing

We attempted to discover issues with the p2p interface using American Fuzzy Lop (AFL) to automatically fuzz that interface. These efforts did not yield results within the time period of the initial audit for a number of reasons detailed below. Nevertheless, we think this is a valuable avenue for further investigation. The results of our AFL driver efforts are available in a private GitHub repo.

The first challenge we encountered was a result of OCaml's built-in AFL support. AFL operates, ideally, on an instrumented binary which guides data generation for fuzzing. AFL uses this guidance to seek inputs that result in novel codepaths. The OCaml toolchain natively supports AFL and can emit code including this instrumentation when requested. There appear to be at least two different versions of this support. We first attempted to build tezos with `OCAMLPARAM="_,afl-instrument=1"`. For reasons we did not determine, this did not produce the instrumentation expected by AFL. We resolved this issue by switching to `4.06.1+afl` and building with no special environment or arguments.

The next challenge we encountered arose from the need for the **driver** given to AFL to execute extremely fast, which is better and the upper bound for execution time is measured in milliseconds. The cost of starting an entire Tezos node and separate connection pool to interact with it vastly overran the practical execution time constraints. The driver we constructed which worked in this way took around 100ms to complete a single execution. This proved slow enough that no useful results could be obtained in a reasonable amount of time. We attempted to address this by using AFL's **persistent mode**. In this mode of execution, a process can live for longer than a single AFL fuzz attempt. This allows certain setup costs to be amortized. For OCaml programs, ocaml-afl-persistent provides this functionality in a somewhat automatic way.

Though persistent mode seems like the right path forward, it brings a new set of challenges. The driver must be modified to be compatible with the new execution environment. This means non-sharable resources (listening TCP ports, databases, lock files, etc) must be acquired differently. Unfortunately, when this resource management is not done correctly, the failure is such that AFL and `ocaml-afl-persistent` make it difficult to determine what has gone wrong. Our process for dealing with these issues came down to repeatedly running `afl-fuzz` under `strace` and reading the trace for errors. This is a tedious process but we haven't found a better one, at the point of finishing the report.

With resource acquisition issues resolved, the next challenge was connection management behaviors of Tezos itself. With a driver creating two long-lived nodes and establishing a new connection from one to the other as part of the persistent mode driver, we encountered errors from Tezos relating to duplicate peers. Our efforts to resolve this by ensuring connections closed before the next iteration began were unsuccessful. We did not overcome this challenge within the available time and so moved on to a different approach for the driver.

We eventually arrived at a persistent mode driver which, for each AFL iteration, created two new nodes, established a connection between them, and executed a series of instructions (sent a series of messages). While this driver did perform somewhat better than the initial driver, it encountered problems with AFL which we were unable to fully diagnose. AFL seemed unable to process extra path information from the information produced by this driver: `last new path : none yet (odd, check syntax!)`. It complained that the instrumentation varied from run to run (*Instrumentation output varies*

across runs). We suppose this may have something to do with multiple OS threads running in the driver process. We were unable to find a way to disable these OS threads. AFL itself only claims support for single-threaded drivers.

Orthogonal to the issues of AFL, persistent mode, OCaml, etc., we separately encountered challenges developing the drivers due to the interfaces published by Tezos itself. Functionality necessary to interact with the Tezos node using the p2p interface was often hidden within an implementation module where it is unavailable for use by the driver. Ultimately, we applied a number of changes to the Tezos codebase itself to publicize this functionality.

Tezos Response

We are discussing various fuzzing and testing strategies and potential implementation changes to make ongoing automated security analysis more feasible.

Status

The Tezos development team has indicated to us that discussions are underway and we look forward to further assisting them, if it is needed

The Tezos Self-Compiler

Tezos describes itself as a self-governing protocol, containing an explicit method for updating the protocol from within the protocol. Even the process for choosing when to apply an update is within the bounds that can be updated by this process. However, the Tezos whitepaper and other materials did not contain an argument about what made this secure, or what exactly are the bounds of what can happen or not.

Only the “economic” protocol can be updated, that is: what constitutes a valid block and a valid transaction, and how the protocol is updated. The network protocol (aka, the shell) is not updatable by this mechanism. The economic protocol deals in immutable data that will remain in the Tezos system forever, but the network protocol is negotiated between any two communicating peers, so it is easy for a peer to test if another peer supports some API and fallback to an older API, if not. Implementing a sandbox (and plugging *every* security hole) is generally considered a challenging task, so it is very important to understand why the method used by Tezos is claimed to be secure. Tezos does not use a separate VM for the economic protocol, but mainly relies on the OCaml type system. The economic protocol is simply implemented in OCaml, and then that is compiled with a specific version of the OCaml compiler, using flags that prevent arbitrary libraries from being used. The standard lib is disabled, preventing any access to network or filesystem IO, and the interface with the rest of the Tezos system is via a specific modules passed into the compiler directly.

This is a very flexible system, and allows changing the protocol to include very bad things, such as an infinite loop: OCaml has tail call optimization, so run away recursion can cause a infinite loop instead of a stack overflow. To use a political metaphor, Tezos has a “weak constitution”, in that it is legally possible to elect a dictatorship. Whether this is a feature or a bug is a matter of perspective, but it should be clearly stated. This means that proposed protocol changes must be carefully audited, more carefully than if there were additional safeguards.

It might be worth running an underhanded OCaml competition, in the spirit of the underhanded c contest. Although, the adversarial actions one can do with C are far more nefarious than the things that can be done with OCaml, it is possible to obfuscate intentions and write difficult to read code in any programming language.

Tezos Response

We agree that the system is biased toward flexibility in this area, and that security analysis of proposed protocol changes must be thorough. The suggestion of sponsoring an "underhanded OCaml competition" is very interesting and may be something we do in the near future, both for security reasons and to increase the visibility of OCaml in the broader community.

Status

Under consideration by the Tezos development team.

Secure Handshakes

There are quite a few details involved in designing a handshake protocol that safely archives privacy, mutual authentication, resists man-in-the-middle and replay attacks. Unfortunately, the most widely available protocols such as TLS and SSH were designed decades ago and are full of legacy cruft, which itself is often a source of vulnerabilities (such as downgrade attacks).

The following is a list of well designed protocols which utilize the same up to date crypto already used by Tezos, libsodium:

- CurveCP: http://curvecp.org/index.htm - CurveCP is designed by the author of the Sodium library, and specifically is designed to protect against DOS attacks (for example, by not allocating memory until the client is authorized). Also note, it's a UDP protocol.
- Noise protocol: http://noiseprotocol.org/ - Noise is a framework for constructing cryptographic handshakes. It is used in Whatsapp and has implementations in a number of languages.
- SecretHandshake: https://github.com/auditdrivencrypto/secret-handshake - An alternative to Noise, written prior to the current Noise version and language options. Although, it is available in fewer languages than Noise, the design paper contains a good overview of the various designs including CurveCP and an early version of Noise.

These are all more than sufficient for Tezos' needs, providing both strong authentication, forward secrecy, and for Noise and CurveCP also deniability. These may be stronger than Tezos requires, but it's preferred to over-engineer than under-engineer, in this area. On the other hand, if Tezos really does want to design something to especially fit its use case, we highly recommend studying these designs carefully, nonetheless.

Tezos Response

We are familiar with these handshake protocols and improving handshake procedures globally in the Tezos codebase is an area of interest.

Status

Under consideration by the Tezos development team.

Recommended Next Steps

We recommend that the unresolved *Issues* and *Suggestions* stated above be addressed as soon as possible in addition to continuing discussions on all of the *Other Observations* listed above.

In future security audits, we strongly suggest that audit results are responded to in a more timely manner and verification is performed within a reasonable time frame of the initial review and delivery of the initial audit report. Over time, it becomes increasingly difficult for auditing teams to contribute value if the focus

becomes on the code diffs of a significantly changed codebase (due to further development over time) and not the security issues themselves. Prompt responses can better facilitate meaningful auditor input throughout future development releases.