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

Security-Review Report TiKV 02.2020

Cure53, Dr.-Ing. M. Heiderich, MSc. N. Krein, MSc. D. Weißer, H. Hippert, BSc. J. Hector

Index

Introduction

"TiKV is an open-source, distributed, and transactional key-value database. Unlike other traditional NoSQL systems, TiKV not only provides classical key-value APIs, but also transactional APIs with ACID compliance. Built in Rust and powered by Raft, TiKV was originally created to complement TiDB, a distributed HTAP database compatible with the MySQL protocol."

From https://github.com/tikv/tikv

This report documents the findings of a security assessment of the TiKV complex. The project was carried out by Cure53 in February 2020 and entailed a broad look at the maturity levels of security found on the TiKV software and surrounding scope, inclusive of a penetration test and a code audit.

It should be noted that the project was commissioned and funded by CNCF as a typical phase of the CNCF project graduation process. This assessment took place in the frames of long-term and well-established cooperation between Cure53 and CNCF. Five testers examined the scope in February 2020, namely in calendar weeks CW7 and CW8; the invested work amounted to a total of eighteen person-days.

After starting the project in a timely fashion, Cure53 effectively inspected the TiKV complex in terms of security processes, response and infrastructure. To best structure the work in relation to the objectives, the work was carried out in several phases. During Phase 1, Cure53 focused on general security posture checks, while Phase 2 was dedicated to manual code auditing. The latter was aimed at finding implementation-related issues that can lead to security bugs. The findings from each of the phases are recounted in respective chapters of this report.

Phase 1 notably yielded a rather high number of issues and impressions. On the contrary, Phase 2 was much less fruitful as regards discoveries, meaning that fewer findings stem from the manual code review parts of the audit. This is also because of the fact that the majority of time was invested into the posture review and a much shorter chunk of the budget was spent on code audits.

Over the duration of this engagement, the Cure53 team worked closely with the TiKV team, remaining connected with those in-house on a dedicated, private channel on the TiKV Slack workspace. The communications were smooth and the TiKV team was helpful in answering all of the Cure53's questions comprehensively.

In the following sections, the report will first shed light on the scope and key test parameters. Next, all findings will be discussed in dedicated chapters for each of the two phases, starting with Phase 1. After that, the report will discuss one finding from the code review phase, which is essentially a general weakness of lower severity. Finally, the report will close with broader conclusions about this 2020 project. Cure53 elaborates on the general impressions and issues a verdict on the TiKV project on the basis of the testing team's observations and collected evidence. Tailored hardening recommendations pertinent to the TiKV code, infrastructure and surroundings are also incorporated into the final section.

Scope

- **TiKV 4.0.0-alpha**
  - https://github.com/tikv/tikv/releases
    - Commit: bd94da3107ad4d515458068b124d8b107ebac1e6
  - A detailed scope document was shared with Cure53 by TiKV
  - A test server setup was made available for Cure53 by TiKV
  - Cure53 was given access to relevant documentation material by the TiKV Team

Test Methodology

The following paragraphs describe the metrics and methodologies used to evaluate the security posture of the TiKV project and codebase. In addition, it includes results for individual areas of the project's security properties that were either selected by Cure53 or singled out by other involved parties as needing a closer inspection.

As with previous tests for CNCF, this assignment was also divided into two phases. The general security posture and maturity of the audited code base, TiKV, has been examined in Phase 1. The usage of external frameworks is audited, security constraints for TiKV configurations were examined and the documentation had been deeply studied in order to get a general idea of security awareness at TiKV. This was followed with research on how security reports and vulnerabilities are handled and how much the entire standpoint towards a healthily secure infrastructure is taken as a serious matter. The latter phase covered actual tests and audits against the TiKV's codebase, with the actual code quality and its hardening being judged.

Phase 1: General security posture checks

As mentioned earlier, Phase 1 enumerates general qualities of the audited project. Here, a meta-level perspective on the general security posture is reached by providing details about the language specifics, configurational pitfalls and general documentation. An additional view on how TiKV handles vulnerability reports and how the disclosure process works is provided as well. A perception rooted in the maturity of TiKV is given, solely on a meta-level. Actual impressions linked to the code quality relate to Phase 2 of the audit process.

Phase 2: Manual code auditing

For this component, Cure53 performed a small-scale code review and attempted to identify security-relevant areas of the project's codebase and inspect them for flaws that are usually present in distributed database systems. This is an addition to the previous maturity analysis and supplies a more detailed perspective on the project's implementation when it comes to security-relevant portion of the code. Still, this Phase was limited by the budget and cannot be seen as complete without a large-scale code review with in-depth analysis of the multiple parts forming the project's scope. As such, the goal was not to reach an extensive coverage but to gain an impression about the overall quality of TiKV and determine which parts of the project's scope deserve thorough audits in the future.

Later chapters in this report will also elaborate on what was being inspected, why and with what implications for the TiKV software complex.

Phase 1: General security posture checks

This Phase is meant to provide a more detailed overview of the TiKV project's security properties that are seen as somewhat separate from both the code and the TiKV software. The first few subsections of the posture audit focus on more abstract components of a specific project instead of judging the code quality itself. Later subsections look at elements that are linked more strongly to the organizational and team aspect of TiKV. In addition to the items presented below, the Cure53 team also focused on the following tasks to be able to conduct a cross-comparative analysis of all observations.

- The documentation was examined to understand all provided functionality and acquire examples of how a real-world deployment of TiKV looks like.
- The network topology and connected parts of the overall architecture were examined. This also included consideration of relevant runtime- and environment-specifications that are necessary to run TiKV.
- The main control flow of the TiKV application was followed and the main structure of the codebase has been analyzed.
- High-level code audits have been conducted. This was necessary to get a quick impression of the overall style and to reach an understanding of which areas are interesting for a more deep-dive approach in *Phase 2* of the audit.

- Normally, past vulnerability reports in TiKV would have been checked out to spot interesting areas that suffered in the past. However, TiKV never received a vulnerability report.
- Concluding on the steps above, the project's maturity was evaluated; specific questions about the software were compiled from a general catalogue according to individual applicability.

Application/Service/Project Specifics

In this section, Cure53 will describe the areas that were inspected for having insight on the application-specific aspects that lead to a good security posture. These include choice of programming language, selection and oversight of external third-party libraries, as well as other technical aspects like logging, monitoring, test coverage and access control.

Language Specifics

Programming languages can provide functions that pose an inherent security risk and their use is either deprecated or discouraged. For example, *strcpy()* in C has led to many security issues in the past and should be avoided altogether. Another example would be the manual construction of SQL queries versus the usage of prepared statements. The choice of language and enforcing the usage of proper API functions are therefore crucial for the overall security of the project.

TiKV is written in *Rust*, which is a language with built-in memory management that can be both safe and unsafe depending on how it is used. It has proven to be a good choice for programmers that do not want to worry about dangling pointers or Use-After-Free vulnerabilities. The TiKV's development originally started with *Go* - another programming language with a good track record of keeping applications mostly free from memory safety issues. However, constraints with *Go*'s garbage collection and unsatisfactory bindings to the *C* language, the switch ultimately has been made to *Rust*.

Consequently, depending on how one chooses to write *Rust* code, as either *safe* or *unsafe,* it plays a big role on how defensively written the code has to be. It is also important to mention that TiKV solely makes use of *Rust Nightly* versions and thus uses features that are not yet enabled for the stable branch. Generally, TiKV's code makes a solid impression. Source code is sufficiently commented. Test-cases are separated from the rest of the runtime. Different components are independently packaged. Deep code nesting is avoided by early error handling. Since TiKV makes use of low-level unsafe code patterns, it is necessary to implement sufficient bounds and access checks. Here, TiKV uses assertions that are present in the release version as well and, thus, makes sure that program flow terminates early. At the time of testing, Cure53 did not manage to spot an issue with the unsafe parts of TiKV.

External Libraries & Frameworks

While external libraries and frameworks can also contain vulnerabilities, it is nonetheless beneficial to rely on sophisticated libraries instead of reinventing the wheel with every project. This is especially true for cryptographic implementations, since those are known to be prone to errors.

TiKV makes heavy use of external libraries and other server components, therefore avoiding reimplementation of already existing solutions. The framework uses *Rust*'s dependency manager called *Cargo*[1] to keep track of and manage all its dependencies.

The TiKV project is currently not using any kind of tracking (or security tracking) for external third-party dependencies. Running the *Cargo*-integrated tool *cargo-audit*[2] revealed multiple issues going back as far back as September 2018. This includes dependencies which are no longer actively maintained and, therefore, pose a substantiated concern in terms of security risk since issues will likely go unfixed or unnoticed and take a very long time to get addressed. Furthermore, multiple packages have been identified that contain active security issues, which have been patched and for which updates are available, leading to the conclusion that patch management is a key area which must be improved for further development of the project. This issue is described in more detail in TIK-01-001.

Further investigation revealed that the *cargo-audit* plugin was once integrated into the *CI* process but has since been disabled because a specific package could not be updated and generated constant notifications. After the package had finally been updated, the team forgot to enable the *audit* functionality again, leaving the project without checks or protection as regards security issues.

Configuration Concerns

Complex and adaptable software systems usually have many variable options which can be configured accordingly to the actually deployed application necessities. While this is a very flexible approach, it also leaves immense room for mistakes. As such, it often creates the need for additional and detailed documentation, in particular when it comes to security.

In terms of security, TiKV provides the means to configure TLS for the connections between the individual TiKV nodes. Due to the requirement of having valid certificates, it

[1] https://blog.rust-lang.org/2016/05/05/cargo-pillars.html
[2] https://blog.rust-lang.org/inside-rust/2019/10/03/Keeping-secure-with-cargo-audit-0.9.html

is hard to provide this feature by default. However, the documentation on the website on how TLS needs to be configured is fairly simple and straightforward and, as such, should be considered by everyone that uses TiKV across untrusted networks.

At the time of writing, on-disk encryption of data was not available and had a 'work in progress' status which can be tracked through the GitHub Issue 3680. In one of the ticket comments, it was mentioned that there may be problems regarding the log entries, which may contain some data. This should definitely be considered during the development of the feature. Once this feature is available on all releases, turning it on by default should be considered to add an additional layer of security to the stored data.

While auditing various code parts, it was discovered that the status server exposes two debug endpoints reachable through HTTP. These endpoints may leak sensitive information and it is recommended to extend the security configuration section, so as to mention this as a side-effect of enabling the status server. Overall, TiKV does not provide much room for misconfigurations that have a severe impact on the security.

Access Control

Whenever an application needs to perform a privileged action, it is crucial that an access control model is in place to ensure that appropriate permissions are present. Further, if the application provides an external interface for interaction purposes, some form of separation and access control may be required.

TiKV does not implement any sort of security model and has no *AAA* (*Authentication, Authorization, Accounting)* functionality and does not provide any method to limit access to the existing databases through user-accounts, roles or client certificates.

Instead of having to secure the custom interfaces or monitoring ports, TiKV relies on the features offered by Kubernetes and the permissions defined in the local Kubernetes environment. Thus, permissions can be managed by the cluster administration via the means provided by Kubernetes.

If Kubernetes is not in use, it uses Docke*r Swarms RBAC,* resource and network separation to achieve access control goals.

Logging/Monitoring

Having a good logging/monitoring system in place allows developers and users to identify potential issues more easily or get an idea of what is going wrong. It can also provide security-relevant information, for example when a verification of a signature fails. Consequently, having such a system in place has a positive influence on the project.

TiKV builds its logging mechanism on top of *Rust*'s *slog* crate. *Slog*'s extensibility allows for easy implementation of a standard logging interface that can be triggered with *Rust*'s default macros. Its functionality is centralized within a separate package called *tikv_util* and implements both formatting and file logging that is written to depend on the log level. A simple command-line switch allows you to specify where logs end up.

Monitoring itself is handled through *Prometheus* and *Grafana*, where *Prometheus* stores monitoring and performance data and while *Grafana* displays them. There are two interfaces one can use. First, there is an HTTP interface to return monitoring data about *PD* components such as information about load balancing or internal data such as cluster details and capacity levels. Generally, this acts as an interface for keep-alive type data. The metrics interface, on the other hand, exposes performance data ranging from garbage collection to number of failed commands. This data can be directly fed to *Prometheus* that by itself contains a useful feature set such as an *AlertManager* that additionally can forward notifications via Mail or SMS. Altogether, TiKV utilizes a modern software stack for logging and monitoring that leaves no real room for complaints.

Unit/Regression and Fuzz-Testing

While tests are essential for any project, their importance grows with the scale of the endeavor. Especially for large-scale compounds, testing ensures that functionality is not broken by code changes. Further, it generally facilitates the premise where features function the way they are supposed to. Regression tests also help guarantee that previously disclosed vulnerabilities do not get reintroduced into the codebase. Testing is therefore essential for the overall security of the project.

TiKV uses *Cargo* as a universal project tool as is the standard in Rust projects. Tests are split into test modules in the respective code files and a larger section for integration tests which reside in a separate directory. This follows the best practices for unit testing under rust, as can be found here[3]. Test runs are integrated into the projects Makefile and run in an automated fashion in TiKVs CI environment.

TiKV integrates multiple different fuzzing libraries to test their project extensively, namely *LLVMs libfuzzer*, *AFL* and Googles *Honggfuzz*. However, the tests do not run in an automated pipeline and are currently run sporadically in a manual fashion. To strengthen the projects security posture, it is recommended to reintegrate the tests into an automated CI task, running them at least in a monthly rhythm. The TiKV team plans to add the fuzz testing back to their regular, planned testing schedule.

[3] https://doc.rust-lang.org/book/ch11-01-writing-tests.html

Documentation

Good documentation contributes greatly to the overall state of the project. It can ease the workflow and ensure final quality of the code. For example, having a coding guideline which is strictly enforced during the patch review process ensures that the code is readable and can be easily understood by a spectrum of developers. Following good conventions can also reduce the risk of introducing bugs and vulnerabilities to the code.

Overall, the TiKV project leaves a good impression regarding the documentation aspect. Note that during the period of this review, the online documentation contained a small notification that there is currently a refactoring taking place. Thus, the state of the documentation, compared to the outline given here, may have changed. However, TiKV does a good job of providing documentation that helps users and developers get started. A general *docs* section provides information about features and the architecture of the project. Further, different aspects for general users are well-documented, for example the deployment, configuration, monitoring and scaling processes are all described in their dedicated sections. A very positive impression leaves the '*Deep Dive'* section, which provides a more in-depth explanation of various components which tremendously eases the process of new developers or contributors that are just getting started with the project.

In addition, the website provides a dedicated section which goes into little detail of how to become a contributor and provides references to various documentations contained in the repository. There, information about formatting code comments and the deployed style guide is provided which provides a good foundation for consistent and readable code throughout the project. The repository also contains a more detailed description about contributing with a rough flow of the contribution process outlined.

Although the overall impression is very good, there is a minor recommendation for improvement in regards to the documentation that is worth considering. Currently, the *"Secure Config"* section contains information on how to report security issues. This may be rather hard to find and should be more easily reachable from the main website. For example, the *"Community"* drop down could include a reference to the vulnerability disclosure documentation, to ensure security researchers can responsibly disclose potential security issues.

Organization/Team/Infrastructure Specifics

This section will describe the areas Cure53 looked at to find out about the security qualities of the TiKV project that cannot be linked to the code and software but rather encompass handling of incidents. As such, it tackles the level of preparedness for critical bug reports within the TiKV development team. In addition, Cure53 also investigated the degree of community involvement, i.e. through the use of bug bounty programs. While a good level of code quality is paramount for a good security posture, the processes and implementations around it can also make a difference in the final assessment of the security posture.

Security Contact

To ensure a secure and responsible disclosure of security vulnerabilities, it is important to have a dedicated point of contact. This person/team should be known, meaning that all necessary information such as an email address and preferably also encryption keys of that contact should be communicated appropriately.

The *MAINTAINERS.md*[4] file lists email addresses of project maintainers that can be contacted to report vulnerabilities. However, the document omits important details, such as the respective PGP keys and an outline of the disclosure process. The guideline on where to report security issues is quite hidden as it is part of the document that also explains how to set up certificates in TiKV[5]. This is clearly not the appropriate place to present this kind of information. Instead, it is advised to put all details related to reporting and disclosing security issues in a dedicated *SECURITY.md* in the project's repository.

Security Fix Handling

When fixing vulnerabilities in a public repository, it should not be obvious that a particular commit addresses a security issue. Moreover, the commit message should not give a detailed explanation of the issue. This would allow an attacker to construct an exploit based on the patch and the provided commit message prior to the public disclosure of the vulnerability. This means that there is a window of opportunity for attackers between public disclosure and wide-spread patching or updating of vulnerable systems. Additionally, as part of the public disclosure process, a system should be in place to notify users about fixed vulnerabilities.

At this point in time, it cannot be evaluated how security fixes are handled and how they are disclosed. This is because there are no public vulnerability reports, no CVEs and none of the commits mentions that a security issue was fixed.

[4] https://github.com/tikv/tikv/blob/master/MAINTAINERS.md
[5] https://tikv.org/docs/3.0/tasks/configure/security/#reporting-vulnerabilities

Bug Bounty

Having a bug bounty program acts as a great incentive in rewarding researchers and getting them interested in projects. Especially for large and complex projects that require a lot of time to get familiar with the codebase, bug bounties work on the basis of the potential reward for efforts.

The TiKV project does not have a bug bounty program at present, however this should not be strictly viewed in a negative way. This is because bug bounty programs require additional resources and management, which are not always a given for all projects. However, if resources become available, establishing a bug bounty program for TiKV should be considered. It is believed that such a program could provide a lot of value to the project.

Bug Tracking & Review Process

A system for tracking bug reports or issues is essential for prioritizing and delegating work. Additionally, having a review process ensures that no unintentional code, possibly malicious code, is introduced into the codebase. This makes good tracking and review into two core characteristics of a healthy codebase.

In TiKV, bugs which are not security related are handled via Github's issue tracker. There is a small guideline[6] on what to include in bug reports and an issue template exists as well[7]. However, there is room to improve in regards to visibility of those links as they are not easy to find. The guideline is not linked anywhere on the TiKV website and the template was only found in the security documentation.

Users can submit their own contributions to the TiKV project via pull requests on Github. The workflow for adding contributions is explained in detail in the project's *CONTRIBUTING.md* which is considered suitable for open source projects. Submissions are reviewed by two TiKV maintainers in order to prevent the submission of malicious or dysfunctional code.

[6] https://github.com/tikv/tikv/blob/master/.github/ISSUE_TEMPLATE/bug-report.md
[7] https://github.com/tikv/tikv/issues/new?template=bug-report.md

Evaluating the Overall Posture

Since TiKV is still a relatively young project, it is hard to judge its security posture in all aspects. A few parts of the posture audit were found inapplicable. For example, how TiKV handles vulnerability reports and disclosure processes will remain to be seen in the future. Also, things like implementation of access-control are outsourced to software like Kubernetes or Docker where the permissions have to be defined through *RBAC* and additional network segmentation.

Still, the code audits gave Cure53 the impression that TiKV is on a good path and that potential concerns about the project's maturity might be misplaced. The decision to use *Rust* as a base language helps a lot. Usage of *unsafe* code parts is rather limited, as such, the number of potential memory-safety issues is drastically reduced. Although the documentation is well-written and helps a lot with getting up to speed with several components of TiKV, concerns about correct and secure deployments arose. Cure53 hopes that the upcoming rewrite of the documentation will help in this regard and provide more insight into areas that can create pitfalls inside the configuration.

Phase 2: Manual code auditing & pentesting

This section comments on the code auditing coverage within areas of special interest and documents the steps undertaken during the second phase of the audit against the TiKV software complex.

- TiKV database encryption feature was evaluated and was found to use the standard *RocksDB* encryption feature. However, at the moment only a *bitwise XOR* is implemented as the feature is not yet production-ready and is to be replaced with an *AES i*mplementation in the future.
- The codebase features a large number of *TODO* blocks, which according to the development team have not been properly tracked or addressed so far. Those issues will now be evaluated and added to the Github issue tracker of the project.
- Handling of environment variables has been analyzed and produced no findings.
- TiKV's *SecurityManager* code has been analyzed and is responsible for the setup of the TLS configuration as well as the database encryption. No issues have been spotted.
- The code was analyzed for security critical debug code in production parts without any results.
- SST and metadata handling, as well as checksum verification, were analyzed. No immediate issues have been spotted. However, the code trail spans multiple projects and multiple programming languages and is rather complex, so it could not be audited in depth.

- The connection between the sample TiKV Go client and the server has been fuzzed on gRPC level to see how stable the protocol is handled, but no unintended behaviors or crashes were spotted.
- Use-cases of the *unsafe* statements in combination with buffer copy operations and length checks were audited. No issues were found in the given time. Two instances in the code seemed a bit risky at first but, upon closer inspection, they turned out to be safe and did not pose a risk.
- The status server component was checked in regard to the exposed HTTP endpoints. One of two debug endpoints (*/debug/pprof/hea*p) could potentially lead to leaking sensitive heap information. However, this information is collected by *Prometheus* and used for their profiling.

TLS Certificates/Handling

The *TiKV* project supports the use of TLS to establish secure sessions for communication. The overall implementation used for handling TLS and certificates throughout the project signifies standard components made available by the *OpenSSL* bindings into the *Rust* language. The *OpenSSL* package is used by TiKV to offer TLS functions throughout the implementation. A configuration option was found to disable proper *hostname* validation, which has been added to the configuration section of this document. However, no insecure standard values are in use.

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.

TIK-01-001 SCA: Security vulnerabilities in outdated library versions *(Info)*

Analyzing the libraries in use revealed that multiple ones do not leverage the most recent versions available. Some libraries are no longer actively maintained and pose a threat to the future security posture of the project and should either be exchanged for different libraries or have to be maintained by the TiKV project group. In addition, multiple libraries contain known security vulnerabilities for which patches and updates are available.

Steps to Reproduce:

1. Install the *cargo-audit* package
2. *cargo install cargo-audit --features=fix*
3. Change to the code directory of TiKV and run

4. *cargo-audit audit*
5. Results will be shown on the command line
6. Alternatively run *make pre-audit && make audit*

It is recommended to upgrade the necessary libraries and formulate a long-term plan on how to handle outdated and no longer maintained external dependencies. As the project *Makefile* contains the *audit* option, it should be resolved why it was not used or how it needs to be integrated into the CI build process.

Conclusions & Verdict

This assessment of the TiKV scope, curated by CNCF and executed by Cure53 in early 2020, concludes with generally positive results. The security posture of the TiKV has been positively evaluated by the involved five members of the Cure53 team. Similarly, high-quality premise was noted for the code base and documentation, therefore the state of the TiKV software stack can be summarized as mature.

To give some context, this investigation belongs to a series of high-level assessments conducted by Cure53 for projects selected by CNCF-selected. However, it stands in contrast to classic code audits and pentests due to its meta-level perspective and forward-looking foci. With such a framing of the project's premise, Cure53 comments mostly on the general security qualities (Phase 1) with minimal emphasis on individual findings (Phase 2). This was also reflected in the allocation of budget.

Starting with enumerating some of the positive aspects and findings, Cure53 would like to underline that the TiKV project makes a sound and strong appearance at the meta-level as regards code quality, coding patterns, style coherence and general structure. This is also reinforced by the fact that static code analysis in the later parts of the audit phase did not reveal significant problems. In Cure53's expert opinion, automated testing or vulnerability scanning will likely not yield more findings. However, deep-dives into specific code areas are definitely necessary.

Next up, fuzzing the gRPC API revealed a solid foundation. The software stack seemed stable and Cure53 did not run into any sort of unexpected behaviors or sudden crashes. The chosen language, *Rust*, does its job of providing a sound codebase which suffers from no obvious memory safety issues. The number of *unsafe* code blocks is kept at a minimum level. Those marked as *unsafe* are implemented in a defensive manner and include thorough error checks. Finally, logging and monitoring is well-handled by supplying the necessary endpoints for *Prometheus*. Pluggable *Grafana* instances can additionally be fed with data to visualize any abnormalities.

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

While the above conclusions point to stellar results, Cure53 also noticed some aspects that could be improved or reflect minor inconsistencies that should be addressed. This by no means overturns the above verdict but rather aims to present slight alterations that could be made to strengthen the perceived high-level of maturity even further. First, worth-highlighting is the fact that the TiKV's codebase contains a fairly large number of TODOs in the sources. This is generally a sign of incomplete functionalities and might mean that it is perhaps too early to judge the maturity of TiKV holistically and conclusively.

The Phase 1 of the project early on highlighted that "*Unit/Regression and Fuzz-Testing*" is somewhat incomplete. Specifically, the implemented fuzzing tests are not properly run or evaluated at the moment. This definitely requires attention. Additionally, the integrated dependency scanner was disabled for convenience reasons as one dependency could not be updated in the past. Sadly, this was later forgotten and never brought to the optimal state. Essentially, this also led to the finding documented as TIK-01-001.

It is nevertheless vital to highlight that the team behind TiKV was very quick to acknowledge the issues mentioned above (especially the disabled dependency scanner) and was very thankful to have them pointed them out, promising to have them addressed in the near future. Finally, while it is clear that resources are limited for the management and rewarding of the issues being found by external community members, the project would likely benefit from a bug bounty program. In other words, dedicating financial means to such a mechanism can be advised.

In conclusion, TiKV should be seen as properly mature and delivering on its security promises. This verdict mostly stems from the positive notes above and the overall good code quality and documentation. In light of the findings from this February 2020 assessment, Cure53 can recommend TiKV for public deployment, especially when integrated into a containerized solution via Kubernetes and Prometheus for additional monitoring.

Cure53 would like to thank Calvin Weng, Yongquan Ren, Jay Lee, Neil Shen, Nick Cameron, Wink Yao and Qiang Zhou from the TiKV team as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.