Fine penetration tests for fine websites

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

Pentest-Report Vitess 02.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, MSc. D. Weißer, J. Larsson

Index

Introduction

“Vitess is a database clustering system for horizontal scaling of MySQL”

From https://vitess.io/

This report documents the results of a security assessment targeting the Vitess software database scaler. Funded by the CNCF / The Linux Foundation, this project was carried out by Cure53 in February 2019 and revealed only three miscellaneous findings.

In terms of resources, the test was completed by six members of the Cure53 team who worked within a time budget of eighteen days. The testers are considered very experienced in their respective fields and have considerable expertise in regard to system complexity, cloud infrastructure, source code auditing, operating system interaction, low-level protocol analysis and multi-angled penetration testing.

Prior to the assessment, a CNCF-typical setup was requested by the testers and provided by the development team. Besides furnishing Cure53 with a Kubernetes-based cluster, locally installed systems were also used for testing. Access to all relevant code and documentation was granted. While the first project meeting provided the basis for the audit, a more ad-hoc kick-off meeting ensured that no major hurdles emerged. A

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

dedicated instant messaging channel was used for arising questions and further inspiration for the test.

An initial assessment of the interfaces and the system architecture, supported also by additional exchange with the development team, revealed a rather limited attack surface. This observation was later confirmed as the subsequent phases of the test ensued. While the results of this assessment are few and far between and may suggest some kind of test limitations, they in fact prove that the Vitess team delivers on the security promises they make. In Cure53's view, there is a clear intention and follow-through on providing a secure system for scaling *MySQL* databases. This was achieved by keeping the attack surface minimal and selecting the language suited for this implementation. The auditors managed to reach wide-spanning coverage of all aspects pertinent to the main repository of the Vitess software system. The most likely avenues for exploitation were chosen and verified for resilience.

In the following sections, the report first defines the scope of the test and then moves on to explaining the employed test methodology. Subsequent phases and details relevant for the test are covered next and clarify which aspects were investigated during this February 2019 assessment. Later in the document, each of the individual findings is discussed, with technical backdrop, illustrations of wider circumstances, and examples with code snippets. Finally, this document ends with some broader conclusions and a general impression that the Cure53 team gained about the Vitess scope system under scrutiny.

Scope

- **Vitess**
  - The publicly available main repository at https://github.com/vitessio/vitess was used as the codebase to be verified.
    - `branch master commit 092479406b27ae61a8fcd146a0e08af2d51a7245`
  - Furthermore, the minimal reference client https://github.com/vitessio/messages was used as additional illustration of use-cases.
    - `branch master commit 7d2ac2189573a7d26cf0f42e17df749673e3d16f`
  - The testers received unencumbered access to a Kubernetes test cluster hosted on Amazon Web Services, which was provided as a reference of an installation typical for a general Vitess deployment.

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

Test Methodology

This section describes the methodology that was used during this source code audit and penetration tests. The test was divided into two phases. Each phase had goals that were closely linked to the areas in scope.

The initial phase (Phase 1) mostly comprised manual source code reviews, in particular in terms of the API endpoints, input handlers and parsers. The review carried out during Phase 1 aimed at spotting insecure code constructs. These were marked whenever a potential capacity for leading to buffer corruption, information leakages and other similar flaws has been identified.

The secondary phase (Phase 2) of the test was dedicated to classic penetration testing. At this stage, it was verified whether the security promises made by Vitess in fact hold against real-life attack situations and malicious adversaries. This included watching out for disclosure of personally identifiable information (PII), particularly in rarely encountered error cases. Additionally, the deployment infrastructure was further investigated for generalizable problems in their instrumentation of the Kubernetes environment.

Phase 1. Manual Code Auditing

The following list of items presents the noteworthy steps undertaken during the first part of the test, which entailed the manual code audit of the sources of the Vitess software in scope. This is to underline that, in spite of the almost nonexistent findings, substantial thoroughness was achieved and considerable efforts have gone into this test. The completed tasks are listed next. Note that a given realm yielded no results unless otherwise indicated with a specific link to a finding.

- A comprehensive list of all accessible API endpoints was enumerated and checked for visible defects. This entailed the functionality exposed by *vtlctld* and the same functions that are also reachable via *vtctlclient*.
- Despite this being only an administrative functionality, a typical example for such functions interacting with the file system would be *ExecuteHook*. This item was analyzed in depth to see if it is by any means possible to inject API commands. The overarching goal was clearly to achieve injection of the OS-level commands. The filter implemented for this particular endpoint protects the function sufficiently and no path traversal instructions can be submitted via the hook's name.
- The monitor and debug web interfaces were analyzed for common vulnerabilities like SQL injection or XSS. However, in all encountered cases the user-input was found to be correctly sanitized, in particular due to the Angular framework's proper handling of parameter-supplied values.

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

- The cryptographic and authentication-related aspects were analyzed for potential general bypasses but no flaws allowing for such circumvention were found.
- A potential timing issue pointed out by the development team was investigated in-depth but revealed no readily available exploitation paths. The reason behind the secure premise is that the attacker would need to obtain the hashes contained in the user-table prior to the attack. A minor issue was filed as VIT-01-001 to describe the exact circumstances.
- As requested, plenty of additional effort was invested into discovering leaks of personally-identifiable information, for example during the extensive logging of executed queries. The *redactor* was checked for flaws allowing for the exfiltration of unredacted or incompletely redacted values. The minor issue was filed (see VIT-01-003) but the real-world impact, as with most information leak issues in general, would need to be considered as low.
- The configuration of the Kubernetes cluster deployment was investigated for common problems like *AllowPrivilegeEscalation*, the application of name-space rules in the network policies, the running of pods in privileged mode, and the characteristics of the *DefaultServiceAccounts, ContainerSecurityContext and RunAsNonRoot*'s usage were confirmed as secure, either because of being correct or by virtue of inapplicability.
- Furthermore, the used secret stores were analyzed for potentially being reused from across other contexts but no encryption prone to disclosure was found in any of the stores.

Phase 2. Code-Assisted Penetration Testing

A list of items below presents the noteworthy steps undertaken during the second phase of the test, which encompassed code-assisted penetration testing against the Vitess system in scope. Given that the manual source code audit did not yield an overly large number of findings, the second approach was added as means to maximize the test coverage. As for specific tasks taken on to enrich this Phase, these can be found listed and discussed in the ensuing list.

- Despite of the Kubernetes cluster provided by the development team, several other, local test installations were built and deployed; one type concerned simple 3-tablet versions that were crafted via the provided *docker* images, another type was built along the lines of the *minikube*-instructions. This was done to gain better understanding of the general deployment structure and the integration with the core components.
- The initially enumerated application endpoints were tested for potential input manipulation, i.e. path traversal and OS-level command injection were attempted for every function that interacted with the file system.

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

- Additional testing for filter circumvention did not uncover any methods to successfully achieve Remote Code Execution. All enumerated endpoints were investigated for bypasses of the described protections without success in a form of a compromise.
- Quite a few of the mentioned endpoints allow execution of the SQL statements either as the *Database administrato*r or *the Application user*. The testers sought to escalate the privilege level in a futile attempt to execute commands as *root.* The unachieved goal was to have file system-level capabilities and turn them into direct file manipulation.
- After checking all user-exposed endpoints, the application-level SQL parser was investigated for robustness. The parser enables features that are not directly present in the *MySQL* and therefore slightly extend the capacities of the database. In particular, the possibility of breaking out of strings by providing legitimately escaped data was attempted but no vulnerabilities could be spotted.
- Interesting behaviors, such as the comment directives, were investigated. In this realm, it is possible to supply additional Vitess runtime options during the execution of the SQL statements via special '/*vt+' directives; nothing particularly wrong with those extensions was uncovered, since it does not seem possible to inject such comment-style options in undesirable locations like strings and similar.
- The web interface was probed for XSS and other general web application flaws without any weaknesses being discovered. Additional path traversal was attempted in the topology browser and, while the application of the '../' path fragment does have an effect, it was found to be impossible to break out of the base directory.
- The network communication between the different Kubernetes application pods was analyzed in order to find potential logical flaws. Nothing prone to being leveraged could be identified.
- The runtime behavior of the different components was probed from a perspective of the services. In focus were Denial-of-Service and similar resource-depletion scenarios.
- The deployed TLS configurations were analyzed for common misconfigurations, again to no avail.

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

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.

VIT-01-001 MySQL: Comparison of Auth Token allows timing Attacks *(Info)*

One of the discovered issues allows an attacker to perform a timing attack against the authentication of the Vitess server. This attack requires an adversary who is in possession of the hashed *MySQL* password, e.g. by obtaining it beforehand from the *mysql.user* table. Thus, this issue has a rather low impact.

The *MySQL* authentication uses hashing and a salt in order to prevent authenticating with only a hash or replaying a previously recorded authentication request. The authentication protocol can be consulted next.

MySQL authentication protocol:

```
Server stores: plain pw OR sha1(sha1(pw))
Server -> Client: salt (randomly generated for each connection attempt)
Client -> Server: sha1(pw) ^ sha1(salt + sha1(sha1(pw)))
Server computes: sha1(client_response ^ sha1(salt + sha1(sha1(pw)))
Server compares: generated_hash == stored_hash
```

In case the password is stored as plain-text, Vitess spares itself the final *SHA1* operation on the server-side and compares the client's authentication token directly with its own version of the scrambled password, rendering the attack described below possible.

In this scenario, if the password hash is not known to an attacker, a timing attack is not possible. This is because the salt is never reused and causes unpredictable changes. If the attacker has retrieved the stored password hash (e.g. via SQL injection), a timing attack can be performed by xoring the tested bit with `sha1(salt + sha1(sha1(pw))`. By exploiting the timing-unsafe comparison, an attacker would be able to retrieve *sha1(pw),* which is sufficient for authenticating to the server. The relevant code is displayed in the following code snippet.

Affected file:
vitess/go/mysql/auth_server_static.go

Affected code:

```
computedAuthResponse := ScramblePassword(salt, []byte(entry.Password))
// Validate the password.
```

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

```
if matchSourceHost(remoteAddr, entry.SourceHost) && bytes.Compare(authResponse,
computedAuthResponse) == 0 {
       return &StaticUserData{entry.UserData, entry.Groups}, nil
}
```

As the attacker requires the double *SHA1* hash of the password and the server has to store the password as plain-text, attacks where this issue is of relevance are not likely. However, it is recommended to perform a timing-safe comparison instead. Go's *ConstantTimeCompare* can be used in this realm.

VIT-01-002 MySQL: Timing attacks due to plain-text password auth *(Low)*

Next to the authentication schemes mentioned above, Vitess also implements *MysqlDialog*, which makes use of plain-text password comparison from both the server's and the client's perspectives. The problem is similar to the one mentioned in VIT-01-001 because the method of comparing both passwords is incorrectly implemented. As such, the method allows timing attacks due to the '==' operator's behavior of aborting early if a match between the characters is not found. The relevant code is displayed below.

Affected File:
vitess/go/mysql/auth_server_static.go

Affected Code:

```
func (a *AuthServerStatic) Negotiate(c *Conn, user string, remoteAddr net.Addr)
(Getter, error) {
[...]
       for _, entry := range entries {
              // Validate the password.
              if matchSourceHost(remoteAddr, entry.SourceHost)
                     && entry.Password == password {
                     return &StaticUserData{entry.UserData, entry.Groups}, nil
              }
```

As in the previously mentioned issue, it is recommended to switch to a timing-safe variant of comparing strings. Using Go's *ConstantTimeCompare* in the *crypto/subtle*'s module is advised.

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

VIT-01-003 PII: Not all SQL values covered by SQL redaction *(Low)*

During an investigation of the Vitess logging mechanism, it was discovered that personally identifying information is not properly stripped from logged queries. Note that this PII potentially resides in *string* values of SQL statements This might lead to the leakage of sensitive user-data whenever the debug interfaces of Vitess are opened. In *MySQL*, strings are usually encapsulated by double- or single quotes but it is also possible to write them as hexadecimal strings prefixed by *0x*. As the variable deduplication ignores strings with the *0x* prefix, those values are not redacted. This behavior can be observed in the following example.

Code example:

```
sql := "select a,b,c from t where x = 1234 and y = 1234 and z = 'apple'
       and foo = 0x1337"
redactedSQL, err := RedactSQLQuery(sql)
if err != nil {
        t.Fatalf("redacting sql failed: %v", err)
}
fmt.Printf("redaction: %v", redactedSQL)
```

Output:

```
select a, b, c from t where x = :redacted1 and y = :redacted1 and z = :redacted2
and foo = 0x1337
```

Expected output:

```
select a, b, c from t where x = :redacted1 and y = :redacted1 and z = :redacted2
and foo = :redacted3
```

Furthermore, it was discovered that large numbers and boolean values are also not redacted. While this is not as crucial as having unredacted strings, certain scenarios where logging such values might be undesirable can be envisioned.

Example queries:

```
select * from t where x = 11111111111111111112;
select * from t where x = true;
```

It is recommended to include the affected data-types in the parameter deduplication. This will help prevent the leakage of data in the logged queries.

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

Conclusions

The results of this Cure53 assessment funded by CNCF / The Linux Foundation certify that the Vitess database scaler is secure and robust. This very good outcome is achieved by limiting the attack surface, taking appropriate care of user-supplied input with security-driven best practices, as well as - to a certain extent - the usage of the Go language ecosystem. A team of five Cure53 testers investigated the software system during a budgeted period of 18 days in February 2019. All tasks were completed in accordance with the specified testing methodology, namely pure code auditing in Phase 1 and source-code assisted penetration testing in Phase 2. The main source code repository examined during the audit pertained to the Vitess itself, while the rather minimal sample client called *messages* was employed as a use-case reference.

The scope of the test was well-defined but not particularly extensive. Conversely, the actual threat-model was left mostly undefined until the actual commencement of the assessment, with the progress of the test making it increasingly more precise. The communications between the auditors and the development team were fluent and incurred no delays. After introductory discussion via mail and video conferencing, the ensuing exchanges took place in a dedicated Slack channel. To give a more realistic assessment of the real-world deployment, the security of the provided Kubernetes cluster was scrutinized. It was considered that no wider-ranging problems impacted this scope item. Further, Cure53 can attest that the Vitess code is cleanly written and mostly well-documented, making it particularly easy for the auditors to review the software's structure. Except for the SQL parser, none of the components had overly complex logic or included typically vulnerable constructs. The above factors contributed to the impressive security posture found during this assessment. The number of issues found during this test is particularly low despite the testers' best efforts to locate additional problem-areas. Only three minor issues were identified and their respective implications should be evaluated as insignificant in the broad picture of assessing Vitess. The intermediate results of the test were shared with the development team during the course of the test, while the details of the discovered issues were only included in this final test report. In light of this February 2019 project, Cure53 concludes that the Vitess database scaler is mature and secure. Therefore, it is deemed fit-for-purpose as far as deployment in modern scalable environments is concerned.

Cure53 would like to thank Sugu Sougoumarane, Gary Edgar, Lori Clerkin and Deepthi Sigireddi from the Vitess 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.