PRESENTS

Fuzzing integration for Vitess

In collaboration with the Vitess maintainers and The Linux Foundation

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 14th May, 2021

Executive summary

This report details an engagement by Ada Logics focused on improving the fuzzing infrastructure of Vitess. In this engagement we built upon earlier fuzzing of Vitess efforts done by Ada Logics by developing more fuzzers that target critical parts of Vitess. The project was stretched between March and April 2021, and the reason an Ada Logics fuzzing audits stretches over a longer period of time in comparison to regular security audits is the need for having the fuzzers run for an extended time as well as iteratively developing the fuzzers based on their initial results.

Vitess has a large user base of Vitess which includes Slack and Youtube. This makes security highly relevant for Vitess and thus improving on the Vitess continuous fuzzing efforts must be considered a priority.

Ada Logics developed a total of 10 fuzzers that together target a significant part of the Vitess codebase. The engagement resulted in finding 5 bugs and none of these bugs have direct security implications. This is a testament to the security and robustness of Vitess. The fuzzers all continuously run following this engagement by way of OSS-Fuzz, which is a free service by Google that supports running fuzzers for critical open source projects.

Results summarised

10 fuzzers developed for both critical endpoints and critical internals.
All fuzzers and fixes pushed to upstream Vitess repository.
All fuzzers running continuously through OSS-fuzz.
5 bugs found: 3 Fixes deployed, 2 still pending.

Engagement process and methodology

The approach to this engagement was to work openly by way of the Vitess Github repository and contribute to the open source project as a regular open source contributor. In addition to communicating by way of pull requests and reviews, communication was also done by way of a private messaging application and we also had several video conference calls throughout the process. All of the work was done against the latest master branch on the Vitess public repository (https://github.com/vitessio/vitess) and the scope was specified to be the source code in https://github.com/vitessio/vitess/tree/master/go. The focus of the engagement was to look at endpoints of the MySQL and the grpc protocols. All fuzzers were implemented by way of go-fuzz.

Overview of fuzzers

In this section we will go through the fuzzers developed for Vitess and in total 10 fuzzers were contributed to Vitess. The following table gives an overview of the fuzzers:

Fuzzer function name	Path	Package
FuzzIsDML	go/vt/test/fuzzing	sqlparser
FuzzNormalizer	go/vt/test/fuzzing	sqlparser
FuzzTLSServer	go/mysql	mysql
FuzzHandleNextCommand	go/mysql	mysql
FuzzReadQueryResults	go/mysql	mysql
Fuzz	go/vt/vtgate/grpcvtgateconn	grpcvtgateconn
FuzzEngine	go/vt/vtgate/engine	engine
Fuzz	go/vt/vttablet/tabletserver/vstreamer	vstreamer
Fuzz	go/vt/test/fuzzing	vtctl
FuzzAnalyse	go/vt/vtgate/planbuilder	planbuilder

We now give a short description of what each of these fuzzers do and, in particular, how they use the fuzz payload to target Vitess.

FuzzIsDML
Targets `sqlparser.IsDML` with SQL queries created from the fuzzing payload.

FuzzNormalizer
Targets `sqlparser.Normalize` with SQL statements created from the fuzzing payload.

FuzzTLSServer
Sets up an `AuthServerStatic` server and establishes a connection to it. It then attempts to find crashes by sending packets to the connection and the contents of these packets are created by way of the fuzzing payload.

FuzzHandleNextCommand
Tests `handleNextCommand` with a single packet created from the fuzzing payload.

FuzzReadQueryResults
Tests the `ComQuery` packet wrapper with an SQL query created by the fuzzing payload.

grpcvtgateconn.Fuzz
Creates a fake vtgate service, connects to it and executes SQL queries against the connection. The contents of the SQL queries are created from the fuzzing payload.

FuzzEngine
Targets `vtgate/engine` by executing queries created by way of the fuzzing payload.

vstreamer.Fuzz
Tests the VStream by building a plan with a filter that has a rule created with the fuzzing payload.

vtctl.Fuzz
A high-level fuzzer that tests the robustness of the `vtctl`.

FuzzAnalyse
Creates a query graph with a `SELECT` statement created from the fuzzing payload.

FuzzTLSServer in-depth

Due to space limitations we will not go in details with all of the different fuzzers. However, to get insight into how they are implemented, we will go into details with one of them, namely the mysql.FuzzTLSServer. From a high level this fuzzer sets up a server, establishes a connection to it and sends packets that are created from the fuzzing payload to the server.

The fuzzer first sets up an AuthServerStatic and create a listener on it:

```
func FuzzTLSServer(data []byte) int {
	th := &fuzzTestHandler{}

	authServer := NewAuthServerStatic("", "", 0)
	authServer.entries["user1"] = []*AuthServerStaticEntry{{
		Password: "password1",
	}}
	defer authServer.close()
	l, err := NewListener("tcp", ":0", authServer, th, 0, 0, false)
	if err != nil {
		return -1
	}
	defer l.Close()
```

The fuzzer then proceeds with getting the host, port and root. The host and the port are needed to establish a connection to the server, and the root is needed to set up the SSL certificate.

```
	host, err := os.Hostname()
	if err != nil {
		return -1
```

```
        }
        port := l.Addr().(*net.TCPAddr).Port
        root, err := ioutil.TempDir("", "TestTLSServer")
        if err != nil {
                return -1
        }
```

Next, the creates a signed certificate:

```
        tlstest.CreateCA(root)
        tlstest.CreateSignedCert(root, tlstest.CA, "01", "server", host)
        tlstest.CreateSignedCert(root, tlstest.CA, "02", "client", "Client Cert")

        serverConfig, err := vttls.ServerConfig(
                path.Join(root, "server-cert.pem"),
                path.Join(root, "server-key.pem"),
                path.Join(root, "ca-cert.pem"),
                "")
        if err != nil {
                return -1
        }
        l.TLSConfig.Store(serverConfig)
```

And the fuzzer then allows the listener to accept incoming connections:

```
         go l.Accept()
```

Finally, the fuzzer connects to the server:

```
        connCountByTLSVer.ResetAll()
        // Setup the right parameters.
        params := &ConnParams{
                Host:  host,
                Port:  port,
                Uname: "user1",
                Pass:  "password1",
                // SSL flags.
                Flags:   CapabilityClientSSL,
                SslCa:   path.Join(root, "ca-cert.pem"),
                SslCert: path.Join(root, "client-cert.pem"),
                SslKey:  path.Join(root, "client-key.pem"),
        }
        conn, err := Connect(context.Background(), params)
        if err != nil {
                return -1
        }
```

The fuzer has now completed initialization and has a working connection to a server. It is now time to send the packets and to do this the fuzzer uses a small helper function that converts the fuzzing payload to a packet and sends it. This helper function is defined as follows:

```
func (c *Conn) writeFuzzedPacket(packet []byte) {
        c.sequence = 0
        data, pos := c.startEphemeralPacketWithHeader(len(packet) + 1)
        copy(data[pos:], packet)
        _ = c.writeEphemeralPacket()
}
```

and it is invoked by the fuzzer as follows, where `query` is the payload from the fuzzer:

```
conn.writeFuzzedPacket(query)
```

At this point all the necessary components for the fuzzer have been completed: A server is created and a connection is established to it and, finally, packets defined by way of the fuzzing payload are being sent to the connection.
However, because the server initialization is expensive in terms of performance, the fuzzer is at this point executing approximately 1 iteration per second. To improve the execution speed we came up with a solution to send 20 packets per fuzz iteration instead of just 1. To achieve this, we create an array of byte arrays in the beginning of the fuzzer:

```
        totalQueries := 20
        var queries [][]byte
        c := gofuzzheaders.NewConsumer(data)
        for i := 0; i < totalQueries; i++ {
                query, err := c.GetBytes()
                if err != nil {
                        return -1
                }
                if len(query) < 40 {
                        continue
                }
                queries = append(queries, query)
        }
```

If `err` in `query, err := c.GetBytes()` should not be nil, we abandon the fuzz iteration and start over. This ensures that we only proceed with the expensive server setup if `queries` have a length of 20. At the end of the fuzzer, instead of sending a single packet as we did above, we send all 20 packets from `queries`:

```
        for i := 0; i < len(queries); i++ {
                conn.writeFuzzedPacket(queries[i])
        }
```

The finished mysql.FuzzTLSServer can be found here.

Issues found

The table below gives an overview of the issues found by the fuzzers. Bug 0, 1 and 2 were triggered by untrusted input, however they are all recoverable. When encountering these in a production environment, they will not hurt a particular process but rather show up as an

error. Bug 4 and 5 were found by fuzzing endpoints with trusted input. As such, none of the found bugs represent a security risk for Vitess.

ID	Public	OSS-fuzz report	Fix	Type of Issue
0	Yes	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=32369	https://github.com/vitessio/vitess/pull/7925	slice bounds out of range in `sqlparser`
1	No	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=32445	https://github.com/vitessio/vitess/pull/8033	index out of range in `sqltypes`
2	No	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=31771	Pending	index out of range in `vtctl`
3	No	https://oss-fuzz.com/testcase-detail/4877295513894912	Pending	runtime error in `vt/topo/shard`
4	Yes	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=32376#c4	https://github.com/vitessio/vitess/pull/8106	Index out of range in `mysql.(*Conn).handleNextCommand`

At the time of project termination, 3 of the 5 bugs have been fixed. We will now go into details with the issue and fix for issue 0,1 and 4.

Issue 0: Slice bounds of range in sqlparser

This bug happens when a “@” in a query would be superseded by EOF. The fix was to check if a particular token is equal to EOF:

go/vt/sqlparser/token.go:

```
if tkn.cur() == '`' {
tkn.skip(1)
tID, tBytes = tkn.scanLiteralIdentifier()
} else if tkn.cur() == eofChar {
        return LEX_ERROR, ""
} else {
        tID, tBytes = tkn.scanIdentifier(true)
}
```

Issue 1. index out of range in sqltypes

This bug is triggered due to improper encoding of the character “§” in SQL queries. The issue was debugged in depth here. It was fixed with:

go/sqltypes/value.go

```
func BufEncodeStringSQL(buf *strings.Builder, val string) {
        buf.WriteByte('\'')
        for _, ch := range val {
                if ch > 255 {
                        buf.WriteRune(ch)
                        continue
                }
                if encodedChar := SQLEncodeMap[ch]; encodedChar == DontEscape {
                        buf.WriteByte(byte(ch))
                        buf.WriteRune(ch)
                } else {
```

4. Index out of range in `mysql.(*Conn).handleNextCommand`

The panic occurs when a packet of length 0 is successfully passed to `mysql.(c *Conn) handleNextCommand`. A fix was introduced to check for 0-length cases:

go/mysql/conn.go

```
                }
                return false
        }
        if len(data) == 0 {
                return false
        }
        switch data[0] {
        case ComQuit:
```

Advice following engagement

The engagement did produce a number of great results but this is not the end. In this section we go through ideas for future work that can improve the fuzzing infrastructure of Vitess even further. In short, we present three areas of future work:

1. Fuzz against a complete cluster.
2. Integrating in structure-aware fuzzing.
3. Connect the VTGate handler with the endpoints.
4. Integrate Vitess fuzzing into CIFuzz.

Fuzz against a complete cluster

We briefly investigated the possibilities of running a full cluster during each fuzz iteration, but this was deemed unachievable within the scope of the engagement. However, we suspect this to be an interesting avenue for further research as it will stress test a complete Vitess

setup in comparison to more targeted fuzzers. A good place to start with this is *TestMain* from the VSchema endtoend test.

Integrate structure-aware fuzzing

Vitess accepts untrusted sql queries as input and an interesting avenue for further work is creating a grammar-aware SQL fuzzer. This could benefit several of the existing fuzzers and allow them to generate valid SQL queries for each fuzz iteration. This type of fuzzer is observed in other database projects as well as in Chromium.

Connect the VTGate handler with the endpoints

The preference of the Vitess maintainers in this engagement was to focus on endpoints. However, when investigating the reach of the endpoints in the currently Vitess build on OSS-Fuzz, we observe that the endpoints do not reach all internal parts of Vitess. The reason for this is that the vtgateHandler is not properly utilized and a testhandler is used instead. Improving on this by allowing the endpoint fuzzers to reach VTGate is a potential area of significant improvement to the fuzzing suite. It is noteworthy that to mitigate this Ada Logics wrote fuzzers targeted VTGate, although it would be preferred to have the fuzzers reach VTGate without through the endpoints.

Integrate Vitess fuzzing into CIFuzz

CIFuzz is a service that enables execution of fuzzer as part of a continuous integration system. This can assist in catching bugs early in the development process and it can easily be integrated into projects already part of OSS-Fuzz. For further information on this we refer to CIFuzz.

Conclusions

In this engagement researchers from Ada Logics carried out an extensive fuzzing audit of Vitess stretched over March and April 2021. A total of ten fuzzers were developed and these fuzzers were developed iteratively based on results during the engagement. During the engagement the fuzzers found five bugs although none of these are security critical. The fuzzer infrastructure has been integrated into OSS-Fuzz which allows the fuzzers to run continuously following the ending of the engagement.

Our efforts show that Vitess benefits from fuzzing and that the fuzzers are capable of finding bugs. However, none of the bugs found in the engagement are critical, and we, therefore, conclude that the security of Vitess is of very high standard.

We thank the Linux Foundation for sponsoring this work as well as the Vitess maintainers for the collaboration.