**Final Report**

**1) Description:**

The malicious URL checker service is used to check whether the input URL is safe or has malicious resources. Its direct consumer is an HTTP proxy that is scanning traffic looking for malicious URLs. Before allowing HTTP connections to be made, this proxy queries our service that handles a database of malicious URLs and checks if the resource being requested is known to contain malicious resource. The user is returned with a JSON response if a URL is safe for consumption.

**2) Planning:**

**Caching:**

1) We analyzed that the system will be a read-heavy application, since there would be a lot of requests to query whether a URL resource is malicious or not as opposed to addition of new malicious URL definition in the database. This warrants the use of caching in our System.

2) Upon looking at the different caching technologies, I narrowed it down to Redis and Memcached because of their low latency retrievals and developer ease-of-use features. Out of these two I chose Redis as our caching system as a prudent measure.

3) Redis offers replication, which would allow me to have scalable reads and high availability clusters. This addresses the future scope of our problem, when we would have millions of users using our URL checker service which would in turn query our cache and database. In a scenario, like this we would want to scale our caching system as well so that it is not overloaded.

4) Apart from this, Redis also offers data partitioning which would allow our data to be partitioned among multiple nodes and would help to handle more requests as and when the demand grows.

**Database:**

1) We analyzed that the malicious URL is an indefinitely growing list. Since this comes under the purview of Big Data, we prefer using a NoSQL database as opposed to a relational Database. This is because when the data inside the database increases, we would need to scale our application.

2) A relational database could be scaled vertically by increasing the storage, but this would be a costly endeavour. Horizontal Scaling of our relational database could add increased complexity to our code. To address this problem a NoSQL database would be a more suitable choice. NoSQL databases are easier to horizontally scale by the addition of more nodes to the cluster.

3) We use mongoDB as they store data as documents. Since each document is in JSON and has the support to embed fields. We can embed additional fields related to a URL should we modify the schema as it allows us to store data of any structure. A key-value NoSQL database wouldn’t be suitable as we would need to store a composite object as a value and would need encoding/decoding logic to extract all the fields of that object to be used in our application layer code.

4) Additionally, mongoDB provides high availability, replication, data partitioning, and scalability which addresses all the needs for the future scope of our problem.

**API:** We use Golang’s gorilla/mux package which implements a request router and dispatcher for matching incoming requests to our handlers. We use the HTTP server from Golang’s net/http package. The server uses the mux routers and is hosted on port 8080.

**3) High-Level Design:**

A picture containing text, indoor, tiled, bath

Description automatically generated  
  
1) Based on our description, this is the system design of our current system. Clients will make REST calls on our malware check service endpoint. The service will query the cache/db to formulate a response for the user. In this system, we would have one instance of the malware check service, cache, and db.

2) The URL data will be seeded to the NoSQL (mongoDB) database during its instantiation. Our malicious link dataset would consist of multiple CSV files each belonging to a different category, namely – benign, defaced, malware, spam, and phishing and with a specific risk associated with it like – Nil (No Risk), High, Medium, and Low. A database start-up script would be used to populate the initial data in the database.

3) This design for current system can be easily scaled by the addition of load balancers to handle multiple client requests incoming on the malware check service, by having a cluster for our caching and database systems (as both Redis and mongoDB support scalability mentioned before). The proposed design diagram for future scope is as follows:

Diagram

Description automatically generated

**4) Low-Level Design:**

**Structure:** We structure our code into three layers, which are as follows:

1. **Application Layer** – The application layer is responsible for handling the incoming request using its handlers, validation of the request, handling business logic, making calls to the store layer and formulating a response to be sent back to the client.
2. **Datamodel Layer** – The datamodel layer consists of the definition of the object (struct) that is inserted/read from the database. The datamodel layer objects are passed to store layer from the application layer (and vice-versa).
3. **Datastore Layer** – The datastore layer is responsible for maintaining database and cache connection objects to make calls to them to retrieve datamodel objects and to cascade them to the application layer.

Apart from this we have a service file which is responsible for making connections to the database, cache, and for running our application server.

**Database Design:** We use a NoSQL database whose fields match the fields of datamodel layer object. Every record in the database will consist of three fields, namely – “url”, “risk”, and “category”. The url field will contain the link of the resource, the risk field will capture the risk associated with that resource (Nil, High, Medium, Low) and category talks about which category it belongs to (Benign, Malware, Spam, Phishing, Defaced).

**Workflow:** The input GET request is intercepted by our application server and routed to the business logic. The code then parses the GET URL, stores the URL parameters like – hostname and port, original path, and request id. The GET URL can also have query parameters which can be used to mention the scheme of the malicious URL link.  
  
The general structure of the GET URL is as follows: **/urlinfo/{request\_id}/{hostname\_and\_port}/{original\_path}?scheme=<scheme>**  
  
Now let’s take an example to understand this better. For example, consider the following GET url incoming on our service: https://hybrid-analysis.com/sample/resource  
In this URL the scheme is https, the hostname and port is hybrid-analysis.com and the original path is sample%2Fresource. So, the incoming GET request would look like the following:

/urlinfo/1/hybrid-analysis.com/sample%2Fresource?scheme=https

* In the above, the keyword “urlinfo” helps the mux router to route it to the correct handler.
* The integer 1 corresponds to the request id.
* The parameter “hybrid-analysis.com” corresponds to hostname\_and\_port and is parsed accordingly.
* The original path “sample/resource” has to be URL encoded before the GET request is issued. This is because the original path might contain characters like “/” that need to be encoded or they could be construed as different parameters by the mux router.
* Lastly, we need to mention the scheme of the URL. If the scheme is not mentioned then “http” is used as default. In this case since it was “https” we added it to the query parameter as shown above.

After we parse the GET URL we can successfully form the original URL that we want to check for malicious behaviour. We then use this URL to query our database if it consists of any such record with that particular URL. If the database contains such a URL then we capture the category it belongs to and the risk associated with it to formulate our response, otherwise we declare the category and risk to be unknown.

**System APIs:**

1. **GetInfo (w http.ResponseWriter, r http.Request)** – This function is defined in the application layer which has the response writer and http request objects as input parameters. This function is responsible for parsing the incoming GET request, storing the URL parameters, forming a malicious URL string, and making calls to the store layer to check if the malicious URL exists in the database (as explained in the workflow above). The return type of this function is a Response object with the following fields: input request id, malicious URL, the risk associated with the URL and the category which it belongs to.
2. **GetInfo(string)** – This function is defined in the datastore layer and is the counterpart of the one in the Application layer. This function takes in the input URL string and checks whether it exists in the cache and then the database. If it exists in the cache, then the meta data object corresponding to that URL is fetched from the cache to form the necessary datamodel layer object and returned back to the application layer, thereby saving successive database calls. If the URL does not exist in the cache then we query the database to look for the URL. If we find it in the database we return the relevant datamodel layer object, and if we don’t then we populate the fields – risk and category, of the datamodel layer object with “Unknown”. Also, before returning we cache the results in our cache so that we can use it later.

**Testing:** The code consists of Unit Testing at the application layer, and integration testing which tests the entire system to check if all the components work in tandem.

The unit tests mock the database by storing datamodel layer objects in a map. The map consists of URL as the key and datamodel object as value to mimic an actual database. Since the unit test is in the application layer, from its perspective we are mainly testing the functionality of GetInfo function. Since, we don’t need to concern ourselves with how the corresponding GetInfo of store layer retrieves the datamodel object, we mock the getInfo function of the store layer. In the mocked GetInfo function of the store layer we perform a map search in constant time to retrieve the necessary datamodel object.

The integration tests verify the behaviour of the application server with respect to routing and the behaviour of database to retrieve the queried data. For performing integration tests we needed to have live database and cache containers. So, we created a test-init-db.js start-up script for the database container to seed data from upon instantiation. We also created a docker-compose file where we specify the containers to instantiate (mongoDB and Redis cache containers in our case). We also specify the ports to expose them on and set the environment as well. We also have an additional Makefile which calls docker-compose build, up, and down, fetches go dependencies, and runs the unit-tests and integration tests.

**Build:**

1. We use Dockerfile to build our Golang application. We also use a docker-compose file that uses this Dockerfile to create a docker image of our application. After that we leverage docker-compose to create a docker container to run our application server. Docker exposes the port 8080 on the container for incoming http requests, and we connect the corresponding port on our local machine to connect to the docker container.
2. We also instantiate the mongoDB and cache service by leveraging docker-compose which creates a docker container for both these services. This allows our application to query the cache and the database.
3. Apart from this we also use Makefile to run commands like docker-compose build and docker-compose up. In our Makefile, we have targets like “build” and “up” that perform the above operations. The “up” target has “build” target as prerequisite, so if one simply types “make up”, they should be able to run our system. There is an additional target called “clear” which basically gracefully shuts down all the containers (basically runs docker-compose down).
4. We have an init-db.js script that is used to seed data in the mongoDB container. This script contains more than 150,000 malicious URL definitions. To create the init-db.js we used a python script called load.py which parses the different csv files present in url\_dataset folder to get the malicious links and populates the init-db.js start-up script.

**5) Instructions to Run**

1. Prerequisites – Make sure to install – Git, Docker (which comes with docker-compose), Golang, python3, Make, Curl or Postman (for API testing)
2. The init-db.js has already been populated with the relevant insert commands, so there is no need to run the python load script.
3. Clone the project from <https://github.com/rchitta2205/url-checker>
4. cd into the project folder and run “make up”. This will instantiate the application, database, and cache containers. Wait for a few seconds until you see “MongoDB init process complete; ready for startup”. Since mongoDB is inserting about 150,000 records it take some time for the init process to complete.
5. Do the API testing by running curl commands – See “Examples” section below.
6. After the API testing is done, press cmd + c to exit, and then issue “make clear” to remove all containers.
7. If you don’t have make installed then you need to run “docker-compose build” and “docker-compose up” to run the server, and then “docker-compose down” to remove the containers.

**For Running Unit Test and Integration Test do the following**:  
1) cd into “server” directory and issue “make unit-test” to run the unit tests. The “unit-test” make target has a dependency on another target “dependency” that fetches all the go dependencies before running the unit tests.  
2) In the server directory, run “make integration-test” to run the integration tests. The “integration-test” make target has a dependency on the “up” target which runs docker-compose up in detached state to instantiate database and cache containers. It also has a dependency on the target “dependency” (mentioned above) for fetching all go dependencies.  
3) If make is not installed then to run unit-test you need to run “go get -v ./…” for fetching the dependencies in the server folder, and then run “go test -v -short ./…”.   
Also, to run the integration-tests you need to run “go get -v ./…” for fetching dependencies and then run “docker-compose up -d” to run the containers in detached state. Then run “go test -v ./…” to run all tests including integration tests. After that, run “docker-compose down” to stop and remove all containers.

**6) Examples:**

1. Let’s say our input URL that we want to check is this - <http://www.atthetopproperties.com/wp-includes/limit/dwgdoc>

We form the GET URL as follows:

<http://localhost:8080/urlinfo/1/www.atthetopproperties.com/wp%2Dincludes%2Flimit%2Fdwgdoc>  
  
In the above GET URL localhost is the machine on which our application is running and 8080 is the port of that machine that has been exposed to the application server. As explained before (in the workflow section) we have the urlinfo keyword for routing purposes, request id in this case is 1, the hostname\_and\_port corresponds to [www.atthetopproperties.com](http://www.atthetopproperties.com) and wp%2Dincludes%2Flimit%2Fdwgdoc is the url encoded original path for “<http://www.atthetopproperties.com/wp-includes/limit/dwgdoc>”. The scheme here for the link is http so we didn’t set it. The output is as follows:

  
  
The URL is a malware and has a high risk associated with it.

1. Let’s say our input URL consists of “https” as the scheme - <https://www.gov.uk/government/policies/water-and-sanitation-in-developing-countries>

We form the GET URL as follows:

<http://localhost:8080/urlinfo/1/www.gov.uk/government%2Fpolicies%2Fwater%2Dand%2Dsanitation%2Din%2Ddeveloping%2Dcountries?scheme=https>

Here, Request ID – 1

Hostname\_and\_port - [www.gov.uk](http://www.gov.uk)

Original Path (URL encoded)- government%2Fpolicies%2Fwater%2Dand%2Dsanitation%2Din%2Ddeveloping%2Dcountries

Query parameter (scheme) – https

The output is as follows:

Text

Description automatically generated

The URL is benign and has “Nil” (No) Risk associated with it.

**Part – 2**

**Q1) The size of the URL list could grow infinitely. How might you scale this beyond the memory capacity of the system?**Ans – As mentioned during the planning phase when the URL list grows indefinitely, we would need to scale our database. With that intention we have used mongoDB in our current application, as it is a NoSQL database with features like high availability, replication, data partitioning, and scalability. MongoDB can be horizontally scaled, easily, by adding more nodes to the database cluster.

**Q2) Assume that the number of requests will exceed the capacity of a single system, describe how might you solve this, and how might this change if you have to distribute this workload to an additional region, such as Europe.**Ans – As mentioned before, when the number of requests exceed the capacity of a single system we can scale our application by adding more application servers and then using load balancers to route them to the appropriate server. Load Balancers use consistent hashing to route traffic efficiently, and we can leverage their inbuilt health check mechanism to only send requests to healthy servers. Apart from this, we also proposed caching which would help us in not making database calls if the required information is present in the cache thereby improving the latency of our application.  
If the workload has to be distributed to additional regions, then we can enable cross-zone load balancing feature that load balancers like elastic load balancer have. This would allow us to distribute traffic across the registered targets in all enabled availability zones.

**Q3) What are some strategies you might use to update the service with new URLs? Updates may be as much as 5 thousand URLs a day with updates arriving every 10 minutes.**Ans – When updating the service with new URLs we can’t use regular POST calls, as every call would have a network overhead, and to update 5 thousand URLs we would waste a lot of time and network resources. To tackle this issue, we can do updates in BATCH. We receive 5K URLs a day, which means on an average we are receiving about 35 URLs every 10 minutes. To address this, we can expose batch or bulk endpoints which would receive these new URLs and store it in our database.

**Q4) You’re woken up at 3am, what are some of the things you’ll look for?**Ans – If I’m woken up at 3 am, then it’s possibly an application issue which requires urgent attention. There could be issues with DNS and clients are not routed to our application. There could also be a network failure because of which our service doesn’t receive client requests. Lastly, there could be a traffic spike and this could have caused network congestion which has resulted in slow service and loading times.   
Apart from this we could also have issues with our infrastructure – like it’s possible that the load balancers could be malfunctioning because of bugs, high incoming requests. There could also be issues with the database due to which we might have slow read/write speeds.  
On our application we could have some bug that we might have failed to capture in our tests as we mostly test in local environments and real-time environment is radically different.

**Q5) Does that change anything you’ve done in the app?**Ans – The above issues are mostly pertaining to the infrastructure. For DNS related issues we could add DNS monitoring safeguards to identify those issues and checking our network infrastructure. For network failure issues we can consider adding redundant routers so that our network is reachable during router failures. Also, we could increase our network bandwidth for bandwidth related issues. With respect to our application, we could implement performance/stress testing that would capture bugs when there is heavy load on our application service.

**Q6) What are some considerations for the lifecycle of the app?**Ans – After the initial development and unit testing/integration testing, as part of the lifecycle of the application we could add performance/stress testing as part of quality assurance. We could follow agile and DevOps development approaches like – Continuous Integration and Continuous Delivery for our application. We also need to look at investment on operations and maintenance as regular maintenance and updates need to be considered. Retirement of an application or service should also be considered as part of maintenance. We could define at what point an application will no longer be supported (End of Life) or a newer version will become available.

**Q7) You need to deploy a new version of this application. What would you do?**Ans – As suggested before, we could employ Continuous Integration, Continuous Delivery and Continuous/Manual deployment of our application. Once the new code changes of the application are requested to be merged to our repository, we could have a build that runs unit tests and integration tests before it can be accepted to be merged. This ensures that the new version integrates with the existing code.   
Next, we use continuous delivery which automatically deploys all code changes to a testing and production environment after the build stage. Through this we have an automated release process and can choose to release/deploy our application to the customers. The decision to have continuous deployment or manual deployment would need further deliberation/consensus of all the stakeholders involved. For this application, since I am the only stakeholder, I would opt for continuous deployment which would imply that my code is deployed without my intervention after the “continuous delivery” phase.