ZSCALER

Google Safe Browsing v2 API: implementation notes

Technical information about using Google Safe Browsing v2

Julien Sobrier, Zscaler

1/17/2011

Version 1.1

http://research.zscaler.com/

Google provides a list of malicious URLs (malware and phishing). These are available through their custom API called Google Safe Browsing. Version 2 is the current version, which is quite different from version 1. This document is a collection of notes about the API, its usage, and its implementation. This document is intended as a complement to the official documentation.

1. Changes

Version 1.0, 12/10/2010

- First internal draft

Version 1.1, 01/17/2010

- First public draft

2. Introduction

Google provides two free lists of malicious URLs: a malware list and phishing list. These lists are not a plain-text listing of domains or URLs. Instead, they are comprised of hashes of parts of URLs. They can be accessed through the Google Safe Browsing API. Version 2 of the API was released to the public in early 2010. The new protocol is different from the first version, and few implementations of the API are available at this time.

Google Safe Browsing v2 has been integrated into several browsers including Firefox, Safari and Chrome.

This document contains a high-level description of the API, and notes about the use of the API and its implementation. Those notes come from questions, caveats or surprising elements or behavior of the API that I ran into while creating Net::Google::SafeBrowsing2, a Perl implementation of the Google Safe Browsing v2 API.

This is a complement to the official API description, and to the corresponding forum. There are still a lot of questions around how the API works and how it should be used. Some cases are not yet well described in the official documentation and have been interpreted differently by users.

I hope this document will be useful to people who want to use the Google Safe Browsing v2 API, or who want to create their own implementation.

3. The Google Safe Browsing v2 (GSB2) API

The full details of the API are available on Google's website.

The URL match is done in two main steps:

3.1. Pre-filtering

The pre-filtering is done on the URL hostname, and then full path. The pre-filtering data have to be stored locally, and synchronized with Google periodically (about every 30 minutes).

Hashes are generated from the URL. I'll use *http://www.example.com/path/file.html* as an example.

3.2. Host keys

The host keys are derived from the top-2 domain components, and the top-3 domain components (when possible) followed by a forward slash (/). In our example, host keys are calculated for *example.com/* and *www.example.com/*.

The host key is made of the first 32 bits of the SHA256 hash of the hostname:

SHA256(*www.example.com/*) =
0x**d59cc9d3**fecd8cf920eadd03012f0be497fb8c0e3c3e7ee8a5070fe145d87977
HOST_KEY(*www.example.com/*) = 0xd59cc9d3

SHA256(*example.com/*) =
0x**73d986e0**09065f182c10bcb6a45db3d6eda9498f8930654af2653f8a938cd801
HOST_KEY(*example.com*)/ = 0x73d986e0

3.2.1. URL key

If one of the host keys has been found in the local GSB2 database, the same type of pre-filtering must be done on the full URL. Up to 30 hashes can be required: 5 possible host names * 6 possible paths. For the example *http://www.example.com/path/file.html*, the different combinations are:

- www.example.com/path/file.html (0x02db21c6)
- www.example.com/path/ (0x4138f765)
- www.example.com/ (0xd59cc9d3)
- example.com/path/file.html (0x02db21c6)
- example.com/path/ (0x4138f765)
- example.com/ (0x73d986e0)

The key is created in the same way as the host key: the first 32 bits of the SHA256 hash. You will notice that the host key and the URL key could be the same (*example.com/* and *www.example.com/*)

3.3. Whitelist

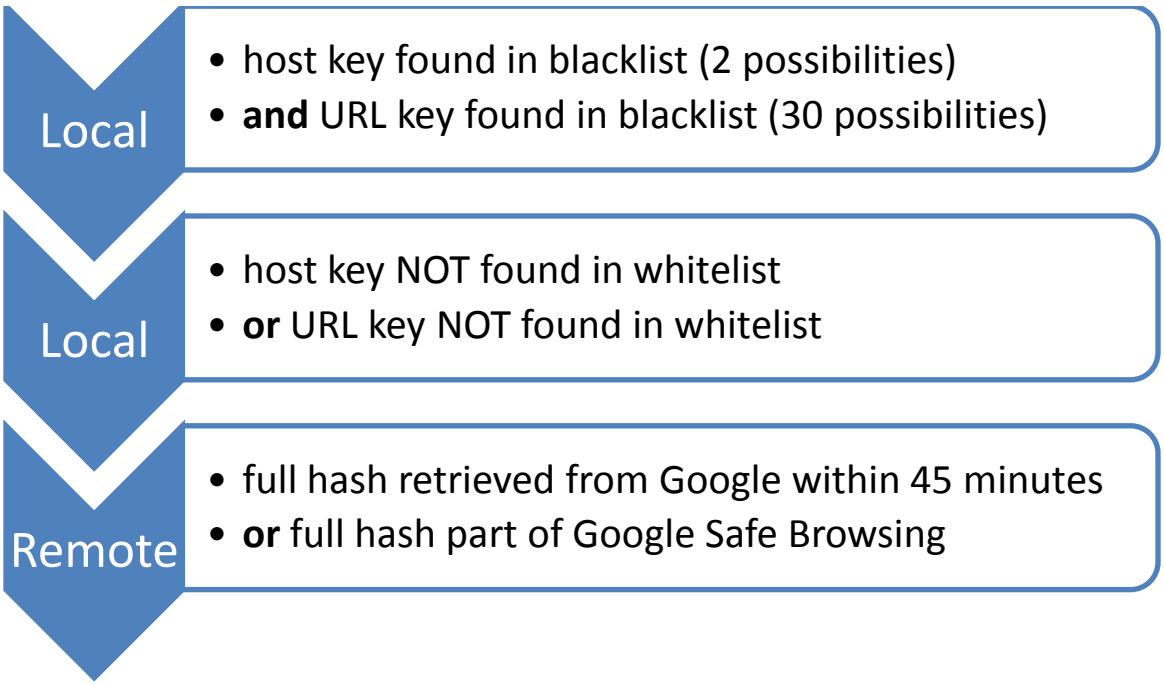
If the host key is found, and one of the URL keys is found, the same type of pre-filtering must be done against the local whitelist. If one host key and one of the URL keys is found in the whitelist, the URL is not malicious.

3.4. Full URL lookup

If the URL has been found in the local blacklist, and not in the local whitelist, the final step is to lookup the full URL hash.

The possible URLs are the same combinations created for the URL key (up to 30 URLs). But instead of checking the first 32 bits (4 bytes), the full hash (32 bytes) is used. This lookup is not done against a local database. It is done by sending a request to the Google Safe Browsing server. The full hashes found can be cached for up to 45 minutes.

Here is the entire flow:



4. Local database

Most of the work done by any Google Safe Browsing v2 library is to maintain a local database for pre-filtering, with the blacklist and whitelist. Full hash matches are also cached locally.

The local database should be updated about every 30 minutes. After each update, Google indicates when the next update should take place. Each list (malware and phishing) are handled separately: they have different content, and a different refresh time. But the library can perform one HTTP request to update both.

4.1. Add chunks

The blacklist is sent as “add chunks”. An add chuck is made of 3 parts:

- Number: an integer, currently between 1 and 28,772 (as of December 2010)
- Host key: 4-byte hostname hash

- Prefix key: 4-byte URL hash (could be empty)

None of these 3 values are unique. On average, one chunk number is associated with a 156 host key + prefix key pair.

An empty add chunk is a chunk number with no host key and prefix key.

4.2. AddDel chunks

The Google update response may include a list of sub chunks to be deleted. This list is referred as "ADDDEL chunks". Although it is called chunk, it is just a range of add chunk numbers. For example, "ad:500-520, 600" means the add chunk numbers 500 to 520, and the add chunk number 600 must be removed from the local database.

4.3. Sub chunks

The white list is sent as "sub chunks". Its structure is similar to add chunks:

- Number: an integer, currently between 1 and 41,546 (December 2010)
- Host key: 4-byte hostname hash
- Prefix key: 4-byte URL hash (could be empty)
- Add chunk number: integer, a reference to an existing add chunk number

On average, one sub chunk number is associated with a 105 host key + prefix key pair + add chunk number.

An empty sub chunk is a chunk number with no host key, prefix key and chunk number.

4.4. SubDel chunks

SubDel chunks are similar to AddDel chunks, but they are used to remove sub chunks:

"sd:500-520, 600" means the sub chunk numbers 500 to 520, and the sub chunk number 600 must be removed from the local database.

4.5. Update Process

The local database is updated incrementally. The list of current add chunk numbers and sub chunk numbers for each list is sent to Google. In return, Google sends a list of one or more URLs which contain the list of new sub and add chunks, and a list of add chunks and sub chunks that should be removed from the local database. It also returns the delay to wait for the next update.

Especially for the first updates, Google can return a list of 4-6 URLs. They have to be processed one after the other, in serial.

If an error happens during the update process, the client must wait between 1 minute and 8 hours to do the next request. The waiting period depends on how many errors in a row are received.

5. Differences between version 1 and 2

The official documentation points out a few differences between version 1 and version 2: the switch from MD5 to SHA256 hashes, smaller update sizes, etc. But there are other differences that have a bigger impact on the clients.

5.1. Update frequency

I have noticed that Google Safe Browsing is updated much more rapidly with version 2 than version 1. False positives are removed faster and malicious URLs are added more quickly. It is not rare to see version 1 lagging 10 to 24 hours behind version 2.

5.2. Offline lookup

In version 1, access to Google servers was required to update the local database only. The lookup process did not require access to the Internet. That means that it was fast and reliable.

Now, the lookup can require access to Google's server to get the full hash. This means up to one second of latency. This requires both Google servers to be available and your network access to Google to be adequate: no routing issues, no DNS downtime, etc.

5.3. Latency and speed

The API is designed to be used mostly primarily swith web browsers. In this case, very few domains are found in the pre-filtering blacklist. For the general use, version 1 is much faster: 2 hashes are created at most instead of up to 30 URLs hashes in version 1.

But when a URL is malicious, it takes longer with version 2:

- 3 steps (pre-filter blacklist, pre-filter white list, full hash lookup) instead of 1
- A server connection is required

Version 2 speeds up the best-case and most common scenarios (no match), but slows down considerably with the worst-case scenario (full look up required).

5.4. My concerns about version 2

A positive URL match requires a connection to Google servers to retrieve the full hash. It cannot be done offline anymore.

With version 1, the worst thing that could happen if you could not connect to Google was that you might be using a slightly older version of the database. With version 2, you may not be able to flag a URL as malicious if you cannot connect to Google. So, you have to hope that Google's server will always be up, and that the client's network will always be fine. In other words, there will be time when you cannot retrieve a full hash form Google's server.

My second concern is about the latency. If you are doing inline scanning of the URL, you have 2 choices:

1. Wait for the lookup to be complete, including the full hash request when needed, before you authorize the content

2. Authorize the content while you are doing the full lookup, and block it in case of a positive match

In case 1, the latency introduced when the URL is found in the pre-filtering is not negligible, and may not be acceptable.

In case 2, you might get a reply from Google too late and the malicious content may already have been accessed and run by the client.

5.5. Should you switch to version 2?

Version 2 has been designed to decrease the load on Google's servers and to work better with low-bandwidth connections. The new protocol may not fit your use case.

5.5.1. Switch to version 2 if…

- You need the most up-to-date blacklists
- You need to minimize the amount of data downloaded

5.5.2. Stay with version 1 if …

- You have an unreliable internet connection
- You need to scan a large collection of URLs at the same time (low latency)
- You need 100% availability
- You need to perform lookups offline
- You need the same latency in all cases (match and no match)

But as the official document states, "The v1 protocol will be phased out shortly, and you are encouraged to migrate to this version of the protocol as soon as possible". You may need to switch to v2 at some point anyway.

6. Usage notes

This is a collection of notes, comments and tips for people who are using the Google Safe Browsing v2 API, or plan on using it.

6.1. Current implementations

Although the version 2 has been available since early 2010, there remain very few implementations of it. As of November 2010, the API v2 is available in 3 languages:

- Python - This is the official implementation of the API from the Google team
- PHP - phpGSB was the first implementation done outside of Google. MAC is not yet supported.
- Perl - I released Net::Google::Safebrowsing2 in October 2010. This is currently the only implementation that allows the use of different back-ends for the local database.

There are implementations in C included in Google Chrome and Mozilla Firefox browsers, but they are not intended to be used as independent libraries.

6.2. First database update

It is very important to note that the first update, when the local database is empty, or any update done after a long period of time (several days) will not give you the complete list of chunks. You need to update the database until Google replies back with "no data", meaning your local database is up to date.

Make sure you continue doing URL lookups after your database is fully up-to-date, not just after the first update.

6.2.1. Regular updates

If you update the local database as the API documentation describes it, there is an average of 30 minutes between 2 possible updates. It takes **46 updates,** about **24 hours,** to get a full database of both lists. The first update gives only 10% of the data: 75,654 out of the final 773,404 add chunks, and 60,117 out of the final 577,568 sub chunks. To get 80% of the final data, 7 updates (4 hours) are required.

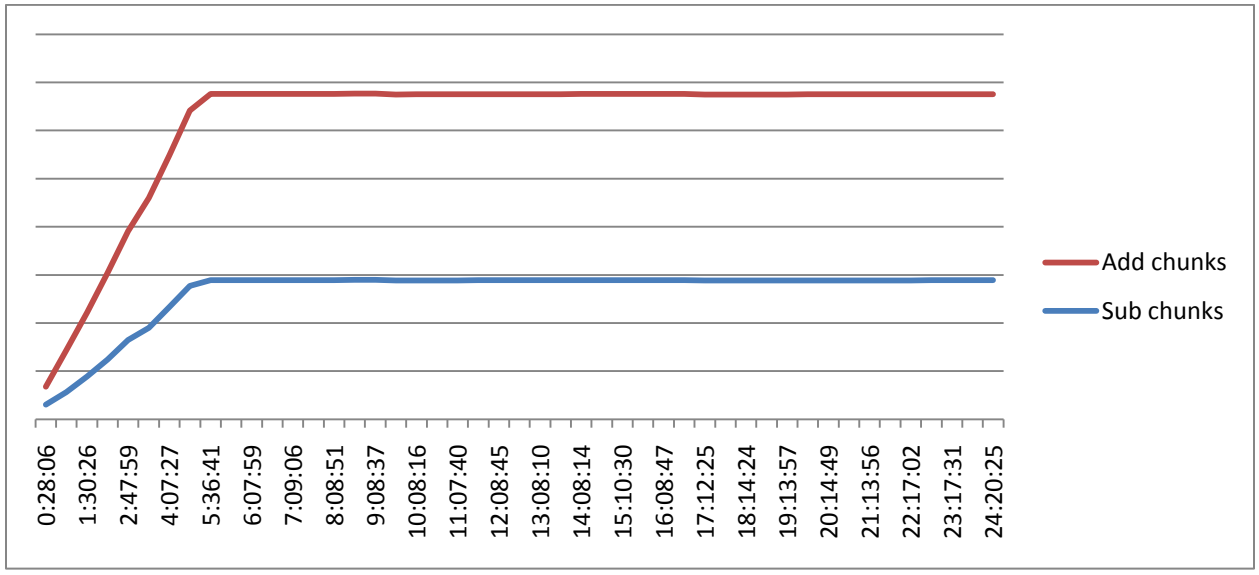


Figure 1 Number of chunks after each update

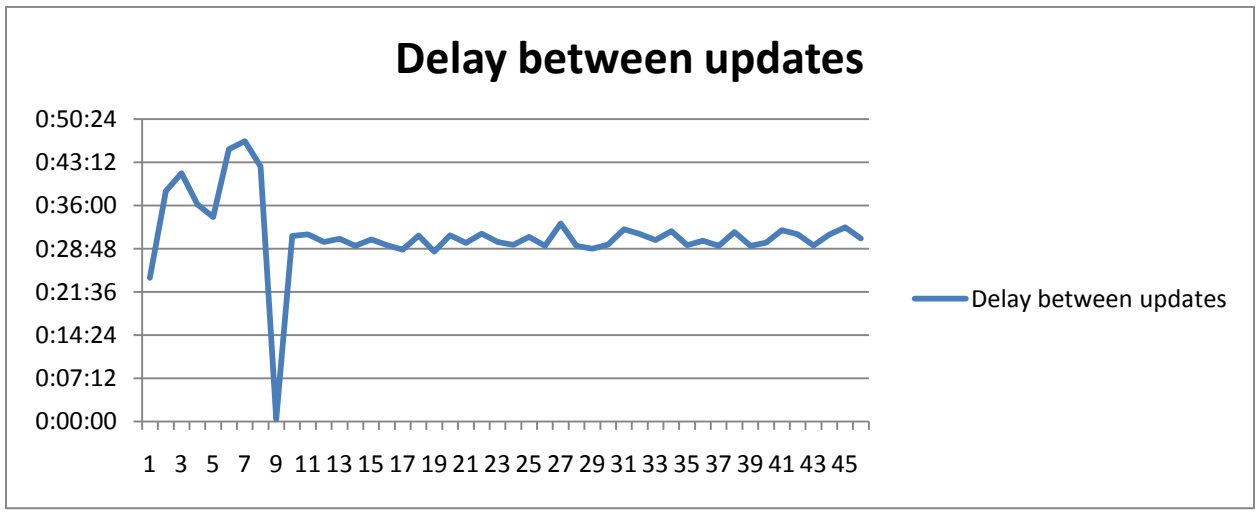


Figure 2 Delay between updates

After some fluctuations, Google indicates a delay of 30 minutes between each update. The delay requested between update 9 and 10 was only 12 seconds. This could be because the Malware and Phishing lists had different delays.

6.2.2. Fast updates

To speed up the first updates, you may skip the delay between each request to the Google API. This is a violation of the protocol, but it seems to be tolerated by Google.

If you do not wait between each update, the time required to get a full mirror of the Google Safe Browsing v2 database goes from 24 hours to 30 minutes (using Net::Google::SafeBrowsing2). The time required with the fast update depends on how efficient is the implementation you are using.

6.3. Following updates

Google provides new add chunks and/or sub chunks about every 30 minutes. That's about 42 updates a day.

Some updates contain no new sub chunks, or no new add chunks. In general, 90 add chunks and 75 sub chunks are added with each update. But once or twice a day, one update can remove 2,000+ add chunks and 1,000+ sub chunks. At the end of the day, the number of database records increases by less than 0.5% on average.

7. Implementation notes

Understanding the API documentation is not easy. If you browse the official Google Safe Browsing API forum, you will notice that a lot of questions get different answers because people interpret the official documentation differently.

It took me quite a few iterations to understand the API fully. I've compiled a list of items which were not straight forward for me, are subject to interpretation, or which I did not expect, while I implemented Net::Google::Safebrowsing2.

7.1. Database storage

Add chunks, sub chunks, full hashes, and information about the last updates must be stored locally. A database is probably the most suitable means of storing all the information.

You need to ensure that whatever backend storage you use supports binary data, not just ASCII, to save the hashes. If you need to convert the hundreds of thousands of binary hashes into ACII for storage, your library will be very, very slow. In my implementation, converting all hashes to ASCII doubles the time for the first update. In general, it is better to store the data in the same format Google sent them.

If you use a database, you need to pay attention to your indexes. The most common operations are:

- Get the list of unique add chunk IDs for each list to request an update

- Get add chunks and sub chunks for a given host key

You will need to store about 800,000 add chunks (id, host key, prefix), and 700,000 sub chunks (id, add chunk number, host key, prefix).

7.2. List of Google blacklists

There are currently two blacklists: *goog-malware-shavar* and *googpub-phish-shavar*. More could be added in the future. So it would be nice to get the list programmatically, rather than hardcode them, to pick up new lists automatically.

The API provides a command to get the list of blacklists. Unfortunately, the command returns 4 lists: the 2 blacklists, and other lists (*goog-regtest-shavar*, *goog-whitedomain-shavar*) used in the Google Toolbar, which should be ignored. In practice, it is not possible to get the list of blacklists programmatically since there is no way to tell which lists should be excluded.

7.3. Atomic updates

The official documentation mentions that all data should be retrieved, then saved: "Upon successful decoding of all the responses and all the binary data, the client MUST update it's lists in an atomic fashion." This means holding up to 120,000 add chunks and 65,000 sub chunks in memory, following up to 10 HTTP redirections before saving everything to disk.

In practice, to keep memory lower and to not loose all the information if one request fails during the entire update, many implementations, including Net::Google::SafeBrosing2, save the chunks after each redirection. Although this is against the official API, this should not cause trouble for Google. For example, if a failure happens during the 5th redirection out of 10, the next database update would simply reply with the 6 missing redirections. This should not be a problem for Google to handle clients which save data after each redirection, even in the case of error.

This also means all HTTP requests to retrieve the add chunks and sub chunks must be done one after the other. It is not authorized to do them in parallel to decrease the update time.

7.4. Error handling

When the client receives an error message from Google, it enters a backoff mode. Depending on the number of errors seen in a row, the client has to wait additional delays before making a new request. The backoff mode is different for the database updates and the full hash lookup.

The client must track the error message for each malware list, and for each of the database lookups independently. The only information needed is the time of the last error, and the number of continuous errors. The client exits the backoff mode after the first success message from Google, so there is no need to store a history of previous errors.

7.5. How to test your implementation?

Unfortunately, it is very tricky to test a new implementation of a library. Google provides very few resources for testing. Most errors result in a generic error message "Request malformed", without many details on the actual problem.

7.5.1. Unit tests

It's actually pretty rare that an API description includes unit tests. There are three types of unit tests provided by Google:

- Hashes: prefix, full hash, etc. These are trivial.
- Possible URLs for full hashes and suffixes
- URL canonicalization

These unit tests cover a very small portion of the entire protocol. But this is better than nothing

However, I think the unit tests for canonicalization miss the points. They include cases that will never be seen in real life, like a hostname that includes a # (pound) sign or other forbidden characters. If you want to pass all these unit tests, you would likely have to write your own URL canonicalization library. This is actually very difficult to do, and you are likely better off using the standard URL encoding library for your language or framework, and ignore the few cases that may not pass.

Also, the canonicalization does not include more important real-life use-cases such as directory traversing (*http://example.com/a/../b*) in the URL path, multiple anchors in the URL (*http://example.com/#a#b*), triple / (*http:///example.com/),* etc. Those are more important that some of the edge cases provided by Google.

7.5.2. API tests

Google provides only 1 URL for testing: *http://malware.testing.google.test/* is included in the *goog-malware-shavar* blacklist. This can be used to test the pre-filtering and full hash lookup.

But remember to check that your local database is up to date. This may require up to 24 hours as shown earlier. Before doing any test with real URLs, you should update your local database until Google replies back with a "No DATA" message.

7.5.3. Most common errors

The most common errors are the following:

- Missing trailing line in request for update

Each line in the request body must end with a \n (0x0A), even the last line. If you forget it, you will get a "Bad Request" error back from Google

- Missing empty chunks

Whenever a database update is requested, the client must send the list of existing add chunks numbers and sub chunks numbers present in the local database:

goog-malware-shavar;a:23550-28297:s:35826-35847,35849-35859,35861-35982,35984-36084,36086-36206,36208-36236,36238-36557,36559-37907,37909-38450,38452-38490,38492-41036

To keep this list compact, Google will send empty add and sub chunks: they have a number, but no host key and prefix. Those must still be saved in the local storage, and must be part of the list sent to Google. If you forget to do so, there will be a lot of holes in the list you send to Google. For example, it may look like this:

googpub-phish-shavar;a:70276-79303,79305-79365,79367-79473,79475-79498,79502,79505,79507-79512,79514-79517,79519-79540,79542-79646,79648-79658,79660-79684,79686-79719,79721-79728,79730-79732,79734-79735,79737-79751,79753-79759,79761-79765,79767-79770,79772-79875,79877-79898,79900-79901,79903-79908,79911-79936,79938-79940,79942-79943,79945-79957,79960-79966,79968-79971,79973-79996,79998-80019,80021-80039,80041-80089,80091-80464,80466-80471,80473-80624:s:1943-1950,1952,1954-1957,1963-1964,1966,1969-1972,1978-1980,1982,1985-1986,1989-1991,1993-1994,1996-1999,2001-2010,2012-2023,2025-2053,2055-2056,2058-2061,2063-2082,2084-2091,2093-2101,2103-2154,2156-2281

- Different canonicalization

The library handling your URL canonicalization may not behave exactly as described by Google, you may have to overwrite it to comply with the API. Look at the unit test cases for canonicalization.

7.5.4. Other resources

Since Google only provides one URL for testing, you will probably need to find other tools or list of URLs to help you check if your library behaves correctly. Unfortunately, it is not easy to find resources that you can trust completely.

7.5.4.1. Firefox, Safari, Chrome

Google Safe Browsing v2 is part of the Firefox, Safari and Google Chrome browsers. You could compare results from these browsers with yours.

But you can never be sure your browser's database is up to date. The browser may also use other lists to detect malware and phishing sites. For example, the 2 URLs provided by Firefox to verify the malware (*http://www.mozilla.com/firefox/its-an-attack.html*) and phishing (*http://www.mozilla.com/firefox/its-a-trap.html*) protections are not part of Google Safe Browsing.

7.5.4.2. Scroogle

Scroogle provides a list of URLs blocked by Google Safe Browsing at *http://www.scroogle.org/malware.txt*. There is not much information about how the list is gathered. It seems to match Google Safe Browsing v1, not v2.

7.5.4.3. Google Safe browsing diagnostic page

Google provides a diagnostic page to check whether a domain is listed as malicious or not. If you see the phrase "Site is listed as suspicious", this means the domain host key is part of the add chunks. To check the diagnostic page for *gumblar.cn*, go to *http://www.google.com/safebrowsing/diagnostic?site=**gumblar.cn***.

7.5.4.4. Other implementations

You can use the existing implementations listed earlier to compare your results. Again, make sure the database you are using is up to date.

7.5.4.5. Known malicious URLs

There are URLs that have been blacklisted for some time and should always be part of the Google Safe Browsing blacklists:

- http://www.gumblar.cn/
- http://flashupdate.co.cc/
- http://www1.rapidsoftclearon.net/

7.6. Message Authentication Code (MAC)

MAC is straightforward to implement. Google uses HMAC SHA256 to sign all requests.

The first HTTP request must include the HMAC key to enable MAC. But the client must also request MAC for each list (append *:mac* to each list), otherwise the redirections will not be signed.

The definition of "websafe base64 encoding" seems to differ with libraries. Google's definition is base64 encoding with:

- + (plus) is replaced by _ (underscore)
- _ (underscore) is replaced by / (forward slash)
- trailing = is kept

7.7. Optimizations

Each library implements the API slightly differently. Small changes can make big differences in the efficiency of the library when dealing with more than 1 million records (add and sub chunks).

7.7.1. Database storage

The first thing you should optimize is your local database storage, as mentioned earlier, by choosing your indexes wisely. This is particularly the case for getting the list of unique add and sub chunk number before each update.

Although using indexes does decrease the write performance, the local storage should mostly be optimized for writing. The initial updates require more than 1 million chunks to be added, then it will be in the thousands range for later updates. For example, disabling journalization in SQLite during the update speeds up the entire process 10 times.

If you use a database, you will probably use the same 3 to 5 SQL queries 90% of the time. Using prepared statements can optimize your SELECT and INSERT queries.

7.7.2. Lookup

The pre-filtering step can be optimized. Unless you are working with a list of suspicious URLs, the vast majority of domain hot keys will not be part of the local pre-filtering database.

One implementation possibility would be to calculate the host keys (maximum 2) and all the hashes for up to 30 possible URL combinations up front, and then look for the host keys and URL keys in the local database. But a faster option would be to calculate the host keys only, do a local lookup for these hashes, and then create the URL keys to filter the matches. In most cases, no host key will be found, and you avoid creating many more SHA256 hashes.

8. Conclusion

Google Safe Browsing v2 is very different from the previous version. Its implementation is also more complex. But the blacklists are updated much faster. It is unclear if and when version will be unavailable.

I believe there's a need for more implementations of the version 2 (C#, Java, Ruby, etc.). I hope this document will answer some of the questions developers will face while reading the official API documentation. Google does not, yet, throttle clients. It is possible to make a lot of tests without receiving error messages back from Google about hitting the limit of numbers of queries. After many trials and errors, the API started to make sense to me. I realized it is not as complicate as I first thought.

You can get in contact with some of the API developers at Google through the Google Safe Browsing v2 API forum if you have any further questions

9. Resources

- Google Safe Browsing v2 API documentation: http://code.google.com/apis/safebrowsing/developers_guide_v2.html
- Google Safe Browsing v2 API forum: http://groups.google.com/group/google-safe-browsing-api
- Google Safe Browsing v2 API official implementation (Python): http://code.google.com/p/google-safe-browsing/downloads/list
- Google Safe Browsing v2 API PHP implementation: http://code.google.com/p/phpgsb/
- Google Safe Browsing v2 API Perl implementation: http://search.cpan.org/perldoc?Net::Google::SafeBrowsing2
- Web Security blog: http://research.zscaler.com/