On the (In)security of Google Safe Browsing

Thomas Gerbet[1], Amrit Kumar[2], and Cédric Lauradoux[2]

[1] Université Joseph Fourier, France
thomas.gerbet@e.ujf-grenoble.fr
[2] INRIA, France
{amrit.kumar,cedric.lauradoux}@inria.fr

Abstract. Phishing and malware websites are still duping unwary targets by infecting private computers or by committing cybercrimes. In order to safeguard users against these threats, Google provides the *Safe Browsing* service, which identifies unsafe websites and notifies users in real-time of any potential harm of visiting a URL. In this work, we study the Google Safe Browsing architecture through a security point of view. We propose several *denial-of-service* attacks that increase the traffic between Safe Browsing servers and their clients. Our attacks leverage the *false positive probability* of the data structure deployed to store the blacklists on the client's side and the possibility for an adversary to generate (second) pre-images for a 32-bit digest. As a consequence of the attacks, an adversary is able to considerably increase the frequency with which the distant server is contacted and the amount of data sent by the servers to its clients.

Keywords: Safe Browsing, Phishing, Malware, Hashing, Security

1 Introduction

Internet users often become victims of *phishing* and *malware* attacks, where dangerously convincing websites either attempt to steal personal information or install harmful piece of software. Phishing and web-based malwares have become popular among attackers to a large degree since it has become easier to setup and deploy websites, and inject malicious code through JavaScript and iFrames.

In order to safeguard users from these malicious URLs, Google has put in place a service-oriented architecture called *Safe Browsing* [11] at the HTTP protocol level. Google Safe Browsing enables browsers to check a URL against a phishing or malware pages' database. Essentially, a browser computes a cryptographic digest of the provided URL and checks if the 32-bit prefix of this digest matches a local database. If there is a match, a server is queried to eliminate any ambiguity. Browsers can then warn users about the potential danger of visiting the page.

Browsers including Chrome, Chromium, Firefox, Opera and Safari (representing 65% of all the browsers in use[3]) have included Google Safe Browsing

[3] Source: statcounter.com

as a feature, while Yandex.Browser, Maxthon and Orbitum rely on an identical service provided by YANDEX. Several other services such as Twitter, Facebook, Bitly and Mail.ru have also integrated these mechanisms to filter malicious links. GOOGLE further claims more than a billion Safe Browsing clients until date [12].

Considering the high impact of GOOGLE Safe Browsing and its siblings, we provide an in-depth assessment of the underlying security issues. To the best of our knowledge, this is the first work that analyzes Safe Browsing services in an adversarial context. In this paper, we describe several attacks against GOOGLE Safe Browsing. Conceptually, our goal is to increase the traffic towards GOOGLE Safe Browsing servers and its clients in the form of a Denial-of-Service (DoS).

Our attacks leverage the data representation employed on the client's side. The current representation stores 32-bit prefixes of malicious URLs in a compact data structure: it entails a false positive probability. In order to mount our attacks, the adversary needs to forge URLs with malicious content corresponding to certain 32-bit prefixes. This is highly feasible, because the computation of a pre-image or second pre-image for a 32-bit digest is very fast. We propose two classes of attacks.

Our *false positive flooding attacks* aim to increase the load of GOOGLE Safe Browsing servers. The adversary forces the service to contact the servers even for a benign URL. Consequently, every time a user visits this URL, he is forced to send a request to the server to resolve any ambiguity. If several popular web pages are targeted, then the Safe Browsing service can be brought to its knees.

Our *boomerang attacks* target the traffic from GOOGLE Safe Browsing servers to its clients. The adversary creates many malicious webpages sharing the same 32-bit prefix digest. After being discovered by GOOGLE, they will be included in GOOGLE Safe Browsing servers and their prefix included in the local database of all the clients. Each time, a client makes a request for this particular prefix, he receives the full digests of all the URLs created by the adversary.

The paper is organized as follows. Section 2 presents the related work. An overview of all prevalent Safe Browsing like services is given in Section 3. In Section 4, we focus on the GOOGLE Safe Browsing architecture. We present the data structures used and a simplified finite state machine for the behavior of the client. Section 5 is the main contribution of the paper and elucidates our attacks. Section 6 discusses the feasibility of the attacks. Finally, Section 7 presents the different countermeasures to prevent our attacks, namely, randomization and prefix lengthening.

2 Related Work

To the best of our knowledge, no prior work has studied Google Safe Browsing service. However, several other web-malware detection systems, such as *Virtual Machine client honeypots*, *Browser Emulator client honeypots*, *Classification based on domain reputation* and *Anti-Virus engines* have been extensively studied in the past. The VM-based systems [18, 19, 26] typically detect exploitation of web browsers by monitoring changes to the OS, such as the creation of new

processes, or modification of the file system or registries. A browser emulator such as [8, 21] creates a browser-type environment and uses dynamic analysis to extract features from web pages that indicate malicious behavior. In the absence of malicious payloads, it is also possible to take a content-agnostic approach to classify web pages based on the reputation of the hosting infrastructure. Some of these techniques [1, 10] leverage DNS properties to predict new malicious domains based on an initial seed. Finally, Anti-Virus systems operate by scanning payloads for known indicators of maliciousness. These indicators are identified by signatures, which must be continuously updated to identify new threats.

Our DoS attacks retain some flavor of *algorithmic complexity attacks*, which were first introduced in [23] and formally described by Crosby and Wallach in [9]. The goal of such attacks is to force an algorithm to run in the worst-case execution time instead of running in the average time. It has been employed against hash tables [2, 3, 9], quick-sort [14], packet analyzers [22] and file-systems [6]. For instance, in [9], hash tables are attacked to make the target run in linear time instead in the average constant time. This entails significant consumption of computational resources, which eventually leads to DoS.

Our work on Google Safe Browsing distinguishes itself from previous works on algorithmic complexity attacks in two aspects. First, the verification algorithm that we attack is distributed over the client and the server. While in the previous works, the target algorithm was run solely on the server side. Second, the data representations used by these Safe Browsing services do not have different worst-case and average case complexities. Instead, these data representations entail false positives. Increasing the false positive probability implies increasing the number of requests towards the Safe Browsing servers made by honest users: Safe Browsing will be unavailable.

3 Safe Browsing: An Overview

The essential goal of any Safe Browsing (SB) mechanism is to warn and dissuade an end user from accessing malicious URLs. All the existing SB services filter malicious links by relying either on a dynamic blacklist of malicious URLs, domains and IP addresses or on a whitelist of *safe-to-navigate* websites. The blacklist is dynamic in nature and hence incorporates the fluctuating behavior of malicious domains. Indeed, many of these malicious traits are short-lived in the sense that a safe-to-navigate domain often transforms into a malicious one and vice versa. A safe-to-navigate website may turn into a malicious one when hackers inject malicious codes into it. Conversely, a domain owner may clean his malicious web page which eventually can be reconsidered as non-malicious. SB features are often included in browsers and hence the service has been carefully implemented to remain robust and keep browsers' usability intact. The robustness of the service relies on fast lookup data structures which may generate *false positives*: a non-malicious URL getting detected as a malicious one.

In addition to providing the basic warnings to users, the SB services often provide webmaster tools. These tools allow a user to report malicious links un-

known to the service. The submitted link upon analysis may get included in the blacklists. Symmetrically, an administrator may explicitly ask the service to remove the malicious flag from an administered domain.

Google started its SB service in 2008, and has since proposed three different APIs. Since the inception of the Google Safe Browsing architecture, other prominent web service providers have proposed similar solutions to detect malicious websites and inform users about them. This includes Yandex Safe Browsing [28], Microsoft *SmartScreen URL Filter* [16], *Web of Trust* (WOT) [27], Norton *Safe Web* [25] and McAfee *SiteAdvisor* [13]. Apart from Google and Yandex Safe Browsing, the remaining services employ a simple and direct *look-up* in a database of potentially malicious links. This requires a client to send each URL in clear to the server and obtain a response. In this paper, we focus on Google and Yandex Safe Browsing since these entail an intrinsic false positive probability which can be modified and exploited by an adversary to mount DoS attacks. We further note that the remaining services are not affected by our attacks since they employ a direct lookup in the database.

4 Google Safe Browsing

Google Safe Browsing (GSB) [11] aims to provide a comprehensive and timely detection of new threats on the Internet. We highlight that the GSB service is not restricted to search, but it also extends to adds. It has further opened paths for new services such as instantaneous phishing and download protection, i.e., protection against malicious *drive-by-downloads*, chrome extension for malware scanning and Android application protection. According to a 2012 report[4], Google detects over 9500 new malicious websites everyday and provides warnings for about 300 thousand downloads per day.

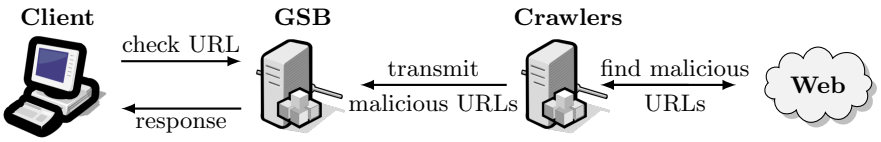

Fig. 1: High level overview of Google Safe Browsing.

Fig. 1 shows a simplified architecture of the service. Google crawlers continuously harvest malicious URLs available on the web and then transmit them to the SB server. Clients can then consult the server to check if a link is malicious.

GSB essentially provides two blacklists: malware and phishing. The number of entries per list is given in Table 1. The lists contain SHA-256 digests of

[4] `http://googleonlinesecurity.blogspot.fr/2012/06/safe-browsing-protecting-web-users-for.html`

malicious URLs and can either be downloaded partially to only update the local copy by adding new hashes and removing old ones or can be downloaded in its entirety. The lists can be accessed by clients using two different APIs, software developers choose the one they prefer according to the constraints they have. In the following, we describe these APIs.

Table 1: Lists provided by the Google Safe Browsing API as on 10/04/2015.

List name	Description	#prefixes
goog-malware-shavar	malware	317,807
googpub-phish-shavar	phishing	312,621
goog-regtest-shavar	test file	29,667
goog-whitedomain-shavar	unused	1

4.1 Lookup API

Google Lookup API is a quite simple interface to query the state of a URL. Clients send URLs they need to check using HTTP GET or POST requests and the server's response contains a direct answer for each URL. The response is generated by looking up in the malicious lists stored on the distant server. This is straightforward and easy to implement for developers, but has drawbacks in terms of privacy and performance. Indeed, URLs are sent in clear to Google servers and each request implies latency due to a network round-trip. To solve the privacy and bandwidth issues, Google offers another API: Google Safe Browsing API described below.

4.2 Safe Browsing API

The Google Safe Browsing API (version 3) is now the reference API to use GSB. The previous version 2.2 is now deprecated and has been replaced by version 3, while keeping the same architecture. In this work, we focus on version 3, but our attacks remain backward compatible.

Google Safe Browsing API has been positively received by the community as a major improvement for privacy. The API is however more complex than the Lookup API and implements a distributed verification algorithm. Moreover, the client now does not handle a URL directly. Instead, the URL is canonicalized following the specifications [4] and hashed with SHA-256 [20]. The digest is then checked against a locally stored database which contains 32-bit prefixes of the malicious URL digests. If the prefix is not found to be present in the local database, then the URL can be considered safe. However, if there is a match, the queried URL is suspicious, but may not necessarily be malicious (it may be a false positive). The client then must query the remote Google server to get all the full digests of malicious URLs corresponding to the given prefix. Finally, if

the full digest of the client's URL is not present in the list returned by the server, the URL could be considered safe. We note that most of the malicious links are short-lived and hence the lists and the local database are regularly updated to incorporate this dynamic behavior. Fig. 2 summarizes a request to the Google Safe Browsing API from a client's point of view.

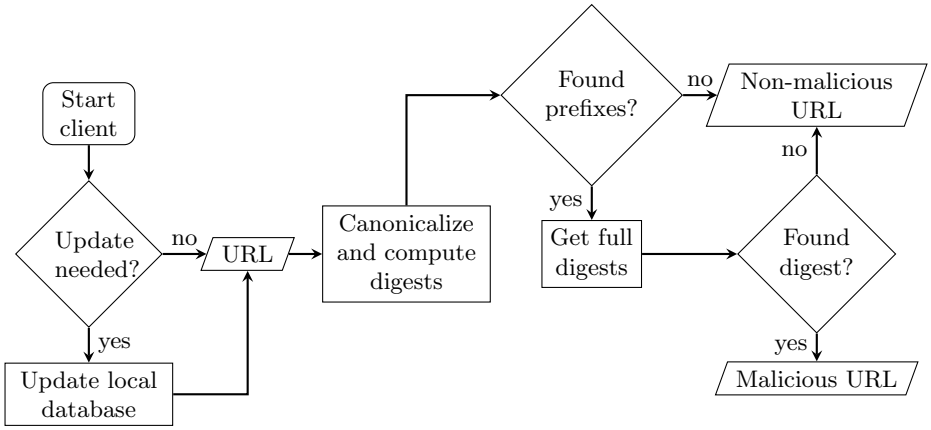

Fig. 2: Google Safe Browsing API: Client's behavior flow chart.

More precisely, the verification requires the client to test several URLs corresponding to a given URL. This is necessary since the complete URL might not have been included in the blacklists. For instance, in order to check whether the URL: `http://a.b.c/1/2.html?param=1` is malicious, the client will lookup for the following 8 strings:

```
a.b.c/1/2.html?param=1          b.c/1/2.html?param=1
a.b.c/1/2.html                  b.c/1/2.html
a.b.c/                          b.c/
a.b.c/1/                        b.c/1/
```

If any of the above URLs creates a hit in the local database, then the initial link is considered as suspicious and the prefix can be forwarded to the Google servers for a confirmation. If there are more than 1 hits, then all the corresponding prefixes are sent. In response, all the full digests corresponding to each prefix is sent by the server to the client. Table 2 presents the statistics on the number of prefixes and the number of full digests matching them. 36 prefixes in the malware list do not have any corresponding full hash, this also holds for 123 prefixes in the phishing list. This may be due to the delay in the file downloads. This comparison reveals that in the worst case, the response contains two SHA-256 digests.

After receiving the full digests corresponding to the suspected prefixes, they are locally stored until an update discards them. Storing the full digests prevents

Table 2: Distribution of the reply size in the number of full hashes per prefix for malware and phishing blacklists.

# full hashes/prefix	Malwares	Phishing
0	36	123
1	317759	312494
2	12	4

the network from slowing down due to frequent requests. In oder to maintain the quality of service and limiting the amount of resources needed to run the API, Google has defined for each type of requests (malware or phishing) the frequency of queries that the clients must restrain to. Behavior of clients when errors are encountered is also fully described (for details refer to [11]).

4.3 Local Data Structures

The choice of the data structure to store the prefixes is constrained by two factors: fast query time and low memory footprint. Google has deployed two different data structures until now: *Bloom filters* [5] and *Delta-coded tables* [17].

In early versions of Chromium (discontinued since September 2012), a Bloom filter was used to store the prefixes' database on the client's side. Bloom filters provide an elegant and succinct representation allowing membership queries. This solution was however abandoned, since the Bloom filters come with false positives. The Chromium development team has now switched to another data structure in the more recent releases. Unlike the classical Bloom filter, this data structure called delta-coded table is dynamic and yet incurs a low memory footprint. Indeed, memory usage is a critical point for web browsers, especially when they are used in memory constrained mobile environments. Furthermore, unlike the Bloom filters, the current data structure, does not have any "intrinsic" false positive probability. However, its query time is slower than that of the Bloom filters. The balance between the query time and the memory usage seems suitable with the operational constraints. At this point, it is worth mentioning that even though the delta-coded table does not have any "intrinsic" false positives, its use to store 32-bit prefixes indeed leads to a false positive probability. False positives arise since several URLs may share the same 32-bit prefix.

We have implemented all the data structures to understand why Bloom filters were abandoned and why Google has chosen 32-bit prefixes. The results are shown in Table 3. If 32-bit prefixes are stored, the raw data requires 2.5 MB of space. Storing these prefixes using a delta-coded table would only require 1.3 MB of memory, hence Google achieves a compression ratio of 1.9. For the same raw data, a Bloom filter would require 3 MB of space. However, starting from 64-bit prefixes, Bloom filter outperforms delta-coded table. This justifies Google's choice of delta-coded tables over Bloom filters and the choice of 32-bit prefixes.

Table 3: Client cache for different prefix sizes.

		Data structure (MB)			
		Delta-coded table		**Bloom filter**	
Prefix size (bits)	**Raw data** (MB)	**size**	**comp. ratio**	**size**	**comp. ratio**
32	2.5	1.3	1.9		0.8
64	5.1	3.9	1.3		1.7
80	6.4	5.1	1.2	3	2.1
128	10.2	8.9	1.1		3.4
256	20.3	19.1	1.06		6.7

5 Attacks on GSB

In this section, we present our attacks on Google Safe Browsing. Since, Yandex Safe Browsing employs an identical architecture, we later conclude that our attacks trivially extend to the service proposed by Yandex. In the following, we first describe our threat model and then in the sequel, we develop our attacks. For the attacks, we first discuss the attack routine, i.e., the steps followed by the adversary and then we develop them.

5.1 Threat Model

There are three possible adversarial goals: increase the traffic towards SB servers, or increase the traffic towards the clients or both. A considerable increase in the traffic towards the server leads to DoS: the Safe Browsing service will be unavailable for honest users. Symmetrically, an increase in the traffic towards a client might quickly consume its bandwidth quota.

An adversary against GSB has the following capabilities. She can create and publish malware and phishing websites with URLs of her choice and can submit them to Google. This is equivalent of being capable of inserting prefixes in the local database of each client.

5.2 Attack Routine

Our attacks follow a three-phase procedure (see Fig. 3): forging malicious URLs, including these URLs in GSB, and updating the local database. In the following, we discuss these phases in detail.

(i) **Forging malicious URLs:** The first step of the attack aims to generate malicious URLs corresponding to prefixes that are not currently included in GSB. This requires an adversary to find unregistered domain names and URL paths on which malicious content could be later uploaded. These canonical URLs are so chosen such that their digests yield the desired prefixes. Hence, in general our attacks are either pre-image or second pre-image attacks on the underlying hash function but restricted to truncated

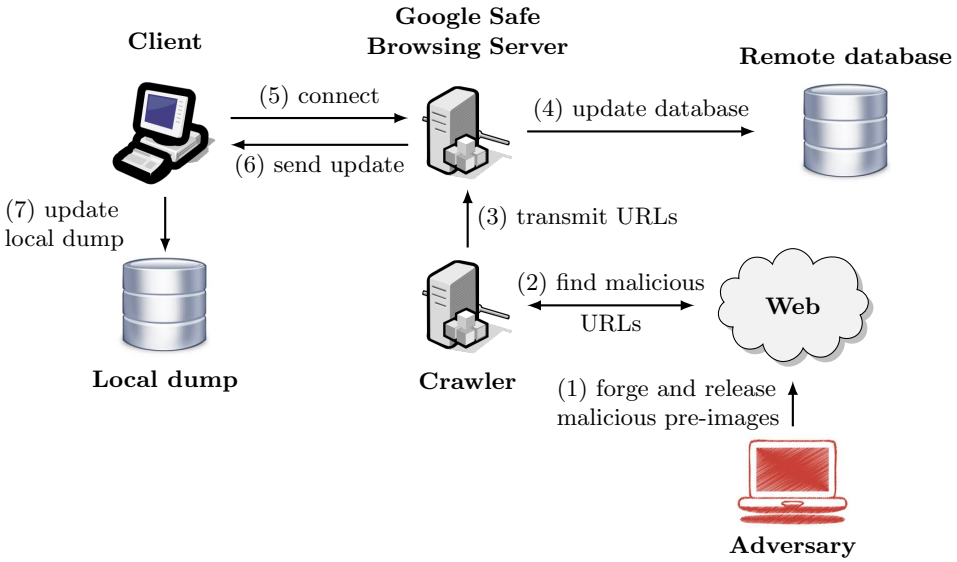

Fig. 3: The attack routine: An adversary forges malicious URLs and diffuses them. These URLs then get detected and indexed by the GOOGLE crawlers and eventually reach the clients through the database update.

32-bit digests. We defer the discussion on the feasibility of generating these URLs till Section 6. We note that rendering these URLs malicious simply requires to upload malicious content by including malicious links to other websites.

(ii) **Including URLs in GSB:** After forging malicious URLs and adding malicious content, the second step of our attack consists in including the newly forged URLs in GSB. There are three ways to do so. Either these pages get detected by the GOOGLE crawlers themselves, or we explicitly ask GOOGLE to crawl our URL[5] which eventually would detect it as malicious, or we use the Safe Browsing API to report these URLs as malicious using the following interfaces:

Malware: `google.com/safebrowsing/report_badware/`
Phishing: `google.com/safebrowsing/report_phish/`

Finally, in order to increase the visibility of these URLs, an attacker may also submit these URLs to GOOGLE's sources such as `phishtank.com` and `stopbadware.org`. We note this is the only step that the adversary cannot control and hence she has to wait until GOOGLE flags the newly found URL as malicious.

(iii) **Local database update:** Once these malicious URLs reach the GSB server, it diffuses them by sending an updated local database to clients

[5] `https://support.google.com/webmasters/answer/1352276?hl=en`

in order to incorporate them. In this way, the adversary has established a data flow with all the end users through GOOGLE.

At the end of the afore-described phases, there are several new malicious links on the web and the local database copy of each user has been accordingly updated. We highlight that the vulnerability in GSB comes from the fact that the local data structure has a false positive probability. With the current prefix size of 32-bits, an adversary can increase the false positive probability by inserting certain prefixes corresponding to malicious URLs. In the remaining part of this section, we explain how to judiciously create URLs in order to pollute GSB.

5.3 False Positive Flooding Attacks

An adversary using pre-images or second pre-images may force the GSB client to send queries to the distant server more frequently than required. The actual attack depends on whether pre-images or second pre-images are found in the first phase of the attack routine. We describe below these two attacks.

Pre-image Attacks. A pre-image for a chosen prefix not currently in the local database is a malicious URL that yields this prefix when SHA-256 function is applied on it. Pre-image attacks on GSB consists in generating pre-images in the first phase of the attack routine. In case of a delta-coded table, this results in an increased false positive probability. This is because at the end of the third phase, the table represents a much larger set of prefixes. In fact, the false positive probability of a delta-coded table is $\frac{\#\text{prefixes}}{2^{32}}$. Hence, the false positive probability linearly increases with the number of new prefixes added to the local database at the end of the third phase of the attack routine.

As a result of the increased false positive probability, many URLs will be declared to be potentially malicious by the local verification. Eventually, the traffic towards the SB server would be increased, since the server would be contacted more frequently than necessary for a confirmation of a true positive. In other words, the attack challenges the fundamental design rationale of reducing the number of requests to the remote server.

Second pre-image Attacks. This attack consists in generating second pre-images in the first phase of the attack routine. While this attack is in principle the same as the afore-described pre-image attack, the impact of second pre-image attacks can however be much more severe. In order to exemplify this, let us consider a non-malicious pre-existing URL, say the RAID 2015 conference web page: `http://www.raid2015.org/` and its 32-bit prefix `0x07fe319d`, an adversary would exhaustively search for a second pre-image of the 32-bit prefix in the first phase of the attack routine. An illustrative example is `http://www.malicious-raid2015.org/115124774`. Such a URL (with malicious content) is then released on the Internet in the second phase. GOOGLE crawlers eventually detect the URL as malicious and add it to its database. The prefix dump on the client side is accordingly updated to incorporate the newly found URL. Now, whenever a user

visits the actual conference web page, the corresponding prefix creates a hit in the local database. Consequently, the browser is forced to request the server to get full digests for the concerned prefix. The threat towards the servers is further exacerbated when an adversary can insert prefixes corresponding to popular web pages. Since these web pages are frequently visited, the probability that a client creates a hit in the local database is high. Consequently, the number of queries can grow quickly and would consume the network bandwidth.

5.4 "Boomerang" Attacks

An adversary may further magnify the amount of bandwidth she can target at SB clients by mounting what we refer to as "*boomerang*" attacks. The term "boomerang" attack is borrowed from [24].

Boomerang attacks come into effect when a client sends a small packet that elicits a full hash for a certain prefix. In reply, a large response packet is returned to the client by the SB server. We note that a few such requests can significantly contribute in the consumption of the allowed bandwidth to a client.

In order to mount boomerang attacks, the adversary generates t (second) pre-images (in the form of URLs) of a target prefix in the first phase of the attack routine. At the end of the third phase, the SB server includes the common prefix in the clients' local database. Now, whenever a client accesses one of these URLs, the corresponding prefix creates a hit in the local database, and hence the client sends the prefix to the server eliciting the corresponding full hashes. In reply, the client receives at least t full hashes, symmetrically forcing the GSB servers to send this data. Consequently, network bandwidth on both sides is increased. We highlight that boomerang attacks are particularly interesting in the case where prefixes corresponding to popular web pages are targeted. Since these pages are frequently visited, a request for the full hashes would quickly consume the allowed bandwidth to a client. Furthermore, since the client has to store these full hashes until an update discards them, these full hashes also consume the browser's cache.

5.5 Impact

Measuring the Increase in Traffic. We have seen that in case of boomerang attacks, the increase in traffic towards clients can be measured by the number of pre-images found. After a week of computation (see Section 6), we obtained an average of 100 second pre-images per prefix. These prefixes correspond to popular web pages. Considering that Google servers currently send at most 2 full digests per prefix (see Table 2), it is possible to achieve an amplification factor of 50 in the average case. The maximum number of second pre-images that we found at the end of one week was 165. This leads to an amplification factor of 82 in the best case for an adversary.

In order to precisely determine the size of the request and response, we have used Mozilla Firefox as the client together with OWASP ZAP proxy[6] and Plug-

[6] `https://github.com/zaproxy/zap-core-help/wiki`

n-Hack, a Firefox plugin. Firefox with the plugin enabled allows the proxy to intercept each communication between the client and the SB server. As measured, the actual request size is 21 Bytes, while a response with at least 1 full digest has a size of 315 Bytes. Hence, if an adversary is able to generate 100 (second) pre-images for a prefix, then the server would send a response of size 30KB. The size of the response received by the client linearly increases with the number of pre-images found. With only 10^4 pre-images the adversary can force the server to send 3MB of data to the client. For the sake of comparison, the local database updates sent to a client during a period of 24 hours was measured to be around 400KB.

Impact on Other Services. It is worth noticing that our attacks on GSB have a wide reach (see Fig. 4). In the first place, our proposed attacks directly affect the usability of the browsers. All popular browsers including Chrome, Chromium, Firefox, Safari and Opera include GSB as a feature. Even worse, several big companies such as Facebook and Twitter import GSB in their social networking services. These services have billions of users and the user activity on these websites is extremely high.

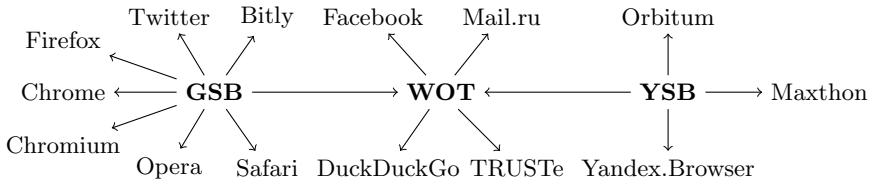

Fig. 4: Impact of our attacks: All popular browsers and social networking services are affected.

The impact of our attacks is even more severe because the client code is available as open-source, hence any external service can use it together with the Safe Browsing API to build its own SB solution. For instance, Yandex provides the Yandex Safe Browsing (YSB) API [28]. This API is a verbatim copy of the Google Safe Browsing API with the only difference being that in addition to the phishing and malware lists proposed by Google, the API also includes 17 other blacklists. Each of these blacklists contains malicious or unsafe links of a given category. Since, the client side implementation of GSB and YSB is identical, all our attacks on GSB trivially extend to YSB.

6 Feasibility of Our Attacks

In this section we empirically study the feasibility of our attacks. More precisely, we consider the feasibility of generating polluting URLs. We highlight that the URL generation is the only attack step with a considerable overhead.

It is believed that for a cryptographic hash function h producing ℓ-bit digests, the basic complexities for finding pre-images and second pre-images is 2^{ℓ}. Hence, if ℓ is large, (second) pre-image attacks are not feasible. However, in case of GSB, the adversary exploits the fact that the SHA-256 digests of malicious URLs are truncated to 32-bits. The truncation allows the adversary to obtain pre-images and second pre-images in reasonable time, i.e., only 2^{32} brute-force operations. We hence employ a brute-force search in our attacks. Since, finding a pre-image or a second pre-image through brute-force entails the same cost, we do not distinguish them in the sequel unless it is required.

6.1 Generating Domain Names and URLs

We note that a pre-image for a 32-bit prefix can be computed using brute-force search in a few hours on any desktop machine. Relying on this fact, we have written an attack-specific pre-image search engine in Python implementing a brute-force search. It was built with a specific goal of searching multiple second pre-images for 1 million popular web pages in the Alexa list[7]. The ensuing results therefore determine the feasibility of mounting a second pre-image based attack on a target benign URL in the Alexa list. Moreover, the engine also allows to generate pre-images required for a pre-image based false positive flooding.

Since a brute-force search is highly parallelizable, we exploit the module Parallel Python[8]. In fact, two levels of parallelization can be achieved. At a higher level, the search for multiple pre-images can be parallelized. Consequently, two pre-images can be obtained in roughly the same time as the search for one pre-image. At a lower level, the generation of a single pre-image can also be parallelized by dividing the interval of search, $[0, 2^{32})$ into sub-intervals.

We also check the availability of the domain name corresponding to each URL. This is necessary since the adversary should own the domain name to be able to upload malicious content on it. This verification is performed using the python module `pywhois`: a wrapper for the Linux `whois` command. Finally, to ensure that the URLs are human readable, we have employed the Fake-factory[9] (version 0.2), a python package to generate fake but human readable URLs.

All our experiments were performed on a cluster with CPython 2.6.6 interpreter. The cluster runs a 64-bit processor powered by an Intel QEMU Virtual CPU (cpu64-rhel6), with 32 cores running at 2199 MHz. The machine has 4 MB cache and is running Centos Linux 2.6.32.

Fig. 5 presents the results obtained by our search engine at the end of 1 week. The total number of second pre-images found was $111,027,928$, i.e., over 111 million. Since the Alexa list contains 1 million entries, as expected the number of second pre-images found per URL is highly concentrated around 100. For around $38,000$ URLs in the Alexa list, we found 110 second pre-images per URL. The tail corresponds to the URLs for which we found the largest number of second pre-images. A summary of the results for the tail is presented in Table 4.

[7] `http://www.alexa.com/`
[8] `http://www.parallelpython.com/`
[9] `https://pypi.python.org/pypi/fake-factory/0.2`

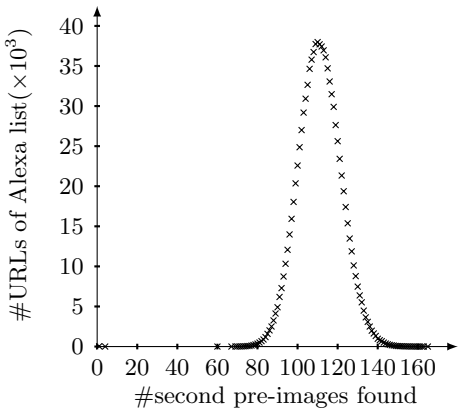

Fig. 5: Number of second pre-images found per URL in the Alexa list of 1M URLs.

Table 4: Prefixes and the domains in the Alexa list for which most second pre-images were found. A sample second pre-image is also provided. The search gave 160 second pre-images for 2 websites.

Prefix	**# Second pre-images**	**Alexa Site**	**Sample second pre-image**
0xd8b4483f	165	http://getontheweb.com/	http://9064606pearliefeil.com/
0xbbb9a6be	163	http://exqifm.be/	http://ransomlemke.com/id15926896
0x0f0eb30e	162	http://rustysoffroad.com/	http://62574314ginalittle.org/
0x13041709	161	http://meetingsfocus.com/	http://chloekub.biz/id9352871
0xff42c50e	160	http://js118114.com/	http://easteremmerich.com/id12229774
0xd932f4c1	160	http://cavenergie.nl/	http://41551460janaewolff.com/

6.2 Cost-Efficient Strategies

The afore-presented results show that the generation of multiple pre-images of a 32-bit digest is time-efficient. However, for an attack to be successful, it is also extremely important that the attack be cost efficient. As described earlier, an adversary upon finding an available pre-image has to purchase the domain name. The cost of each is typically $6-10 for a `.com` top-level domain. As boomerang attacks require an adversary to generate multiple (second) pre-images, the final cost of an attack might become prohibitive. A more cost-efficient solution is to purchase one single domain to cover several prefixes. Let us say that the chosen domain is `deadly-domain.com`. The adversary then simply requires to create several malicious and non-malicious links on the domain. To this end, two strategies can be employed. We describe below these strategies. In order to exemplify these, let us consider for instance that the adversary wishes to use `deadly-domain.com` to cover three prefixes: `prefix1`, `prefix2` and `prefix3`.

1. **Same Depth Strategy**: Search for malicious tags at the same depth, such as `maltag1`, `maltag2` and `maltag3` such that:

prefix(SHA-256(deadly-domain.com/maltag1)) = prefix1,
prefix(SHA-256(deadly-domain.com/maltag2)) = prefix2,
prefix(SHA-256(deadly-domain.com/maltag3)) = prefix3.

These tags may correspond to name of files, such as .html, .php, etc. Once these tags are found, these pages or files are then linked to malicious content. Table 5 presents sample second pre-images for popular web pages generated using the same depth strategy.

Table 5: Same depth strategy: Sample second pre-images for popular web pages.

malicious URL	popular domain	prefix
deadly-domain.com/4294269150	google.com	0xd4c9d902
deadly-domain.com/21398036320	facebook.com	0x31193328
deadly-domain.com/5211346150	youtube.com	0x4dc3a769

2. **Increasing Depth Strategy**: Search for malicious tags at increasing depth, such as maltag1, maltag2 and maltag3 such that:

   prefix(SHA-256(deadly-domain.com/maltag1)) = prefix1,
   prefix(SHA-256(deadly-domain.com/maltag1/maltag2)) = prefix2,
   prefix(SHA-256(deadly-domain.com/maltag1/maltag2/maltag3)) = prefix3.

   These tags correspond to the name of directories. Once these tags are found, malicious files are uploaded in these directories. Table 6 presents sample second pre-images for popular web pages generated using the increasing depth strategy.

Table 6: Increasing depth strategy: Sample second pre-images for popular web pages.

malicious URL	popular domain	prefix
deadly-domain.com/4294269150/	google.com	0xd4c9d902
deadly-domain.com/4294269150/3263653134/	facebook.com	0x31193328
deadly-domain.com/4294269150/3263653134/2329141652/	youtube.com	0x4dc3a769

The malicious tags are randomly generated using the previous search engine. Once all the tags are found, the malicious URLs found can be released on the web. If GSB considers all these URLs as malicious, then it will include all the three prefixes in their blacklists. Hence, only one domain suffices to cover three prefixes. The same strategy can be used for second pre-images based attacks, where popular websites are targeted. In this case, the prefixes correspond to the

domains in the Alexa list. Last but not least, this also allows to generate multiple pre-images: it suffices to fix a prefix and search for several malicious tags.

However, there is a small caveat in the above strategies. Considering that several URLs on the same domain host malicious content, the GSB servers may decide to blacklist the entire domain by adding only the digest prefix of the domain name in their lists. This results in the degenerate case, where only one prefix gets included in the blacklist. In order to circumvent this issue, the adversary would need to create intermediary malicious pages on the domain which correspond to useful or safe to browse content. Without these pages, there is a chance that only the initial domain name is flagged as malicious.

6.3 Comparing of Domain Generation Strategies

As described in the previous sections, three strategies are possible for generating a malicious domain and the corresponding URLs: the naive strategy of generating one domain name per prefix, the same depth strategy and the increasing depth strategy. The final topologies of the domains obtained through these are schematically presented in Fig. 6.

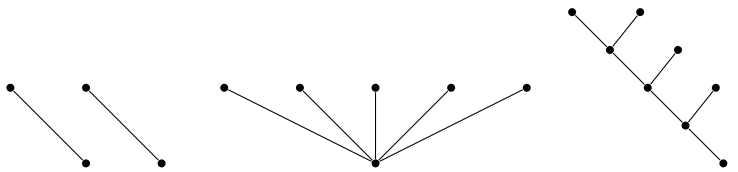

Fig. 6: Three possible topologies of the domains generated by an adversary are depicted. The lowest node in the graphs represent the root domain, while the other nodes correspond to the files or the directories created on the root domains. The topology (a) represents the case where a new domain is created for each prefix, (b) represents the domains generated by the same depth strategy, and (c) corresponds to those corresponding to the increasing depth strategy.

We reiterate that while the naive strategy is highly parallelizable, it may be cost prohibitive. The same depth strategy however assures the same level of parallelization while remaining cost efficient. Compared to these strategies, the increasing depth strategy is relatively less parallelizable since the malicious tag generation is sequential. Indeed, the adversary has to wait for the first malicious tag to be able to generate the next one. While search for multiple pre-images cannot be parallelized, yet the search for a single pre-image is parallelizable.

Despite its disadvantages, the increasing depth strategy greatly impacts the size of the full hash request and that of the corresponding response. In order to understand this, let us suppose that all the corresponding malicious URLs get included in the blacklists. If a client visits the final URL, he first decomposes the

URL into smaller URLs and checks if their corresponding prefixes are in the local database. Since, these decompositions correspond exactly to the URLs generated by the attacker, all these prefixes create hit in the local database. Consequently all the prefixes are sent to the server. The exact number of prefixes sent to the server is equal to the depth of the tree (as in Fig. 6). In case only the URLs up to a depth d are blacklisted by the SB service, then the number of prefixes sent is exactly d. Symmetrically, the server in response has to send back all the full hashes corresponding to all the received prefixes. In this sense, the final URL is the deadliest URL. We conclude this section by summarizing the comparison between these strategies in Table 7.

Table 7: Comparative summary of the different strategies for generating malicious domains. The comparison is relative to a scenario without any attack on the service. ϕ denotes that no change could be obtained by the strategy. Adversary's complexity inversely relates to the level of parallelization that she can achieve.

	Client's request		Server's response		
Strategy	size	frequency	size	frequency	Adversary's complexity
Naive	ϕ	+	+	+	+
Same depth	ϕ	+	+	+	+
Increasing depth	++	+	++	+	++

7 Countermeasures

The design of a good countermeasure to our DoS attacks must be easy to deploy, ensure compatibility with the current API and entail acceptable cost in terms of time and memory at the client and the server side. We investigate two countermeasures to our attacks. The first and the obvious one is to increase the size of the prefix. The second solution consists in randomizing the system with keys.

7.1 Lengthening the Prefixes

The core of our attacks is the computation of pre-images and second pre-images. As empirically shown in the previous section, 32-bit prefixes are not enough to prevent those attacks and thus increasing their length is the obvious choice. Unfortunately, it has some effect on the size of the data structure at the client side. Furthermore, the designers of Safe Browsing want to keep this size below 2MB. From Table 3, we observe that delta-coded tables do not scale well if we increase the prefix length. For 64-bit prefixes, the size of the data structure gets tripled. The size of the Bloom filter remains immune to the change in the prefix size, but unlike the delta-coded table, the data structure would no longer be dynamic which is critical in the context of Safe Browsing. It can be argued that storing

the 20.3MB of all the full digests at the browser side is not a big issue. Unfortunately, publishing the full digests has very dangerous side effects. Indeed, the URLs included in GSB may correspond to legitimate websites that have been hacked. With the full digests, hackers can use GSB to identify weak websites and increase the problem for the corresponding administrators. Recovering URLs from full cryptographic digests was demonstrated to be possible when the MD5 digests of the URLs censored in Germany were published and inverted (see `bpjmleak.neocities.org`).

7.2 Probabilistic Solutions

Since our attacks retain some flavor of algorithmic complexity attacks, we explore existing solutions to prevent such attacks. The first countermeasure proposed to defeat algorithmic complexity attacks was to use universal hash functions [7] or message authentication codes (MAC) [15]. Such methods work well for problems in which the targeted data structure is on the server side. The server chooses a universal hash function or a key for the MAC and uses it to insert/check elements submitted by clients. The function or the key is chosen from a large set so that it is computationally infeasible for an adversary to either guess the function/key or pre-compute the URLs for all the possible values. We further highlight that all the operations made on the data structure can only be computed by the trusted server.

For GSB, the situation is different. The data are provided by Google and inserted in the data structure by the client. It implies that all the prefixes and any keys can not be kept secret to the client and are therefore known by an adversary. With the knowledge of 32-bit prefixes, the adversary can mount false-positive flooding attacks or boomerang attacks.

The first solution is to use a MAC directly on the URL prefixes. A key is chosen by Google for each client and then shared. The prefixes received are unique to each client. An adversary can mount a second pre-image attack on a given user if she knows his key. But it can not be extended to other users without the knowledge of their key. This solution defeats both pre-image and second pre-image based attacks. However, it requires that Google recomputes the prefixes for each client. It does not scale well but it can be used by Google to protect important clients such as Facebook or Twitter (see Fig. 4).

Another solution could be that all the clients share the same function/key, but this key is renewed by Google every week or month. The key is used to compute the prefixes as in the previous solution. An adversary can pollute Google Safe Browsing only for a short period of time. In this strategy, all the prefixes must be re-computed on a regular basis by Google servers with a different function or key and diffused to all the clients. However, this may represent a high communication cost once per week/month.

8 Conclusion

We have described several vulnerabilities in Google Safe Browsing and its siblings which allow an adversary to mount several DoS attacks. The attacks allow an adversary to either increase the false positive probability or force the service to believe that a target benign URL is possibly malicious, hence leading to a DoS. The situation is further exacerbated through our described boomerang attacks. The current Safe Browsing architecture further permits the adversary to simultaneously affect all the users of the service. In Google Safe Browsing, the back-end service attempts to implement a cache at the client side to reduce their servers' load. The design of this cache leads to the use of potentially insecure data structures. In fact, the digests stored are too short to prevent brute-force attacks. Increasing the digest length is a secure countermeasure, but the current data structure does not scale well with this modification and it can amplify the security issues by increasing the number of attacks on vulnerable websites.

References

1. Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., Feamster, N.: Building a Dynamic Reputation System for DNS. In: 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings. pp. 273–290 (2010)
2. Bar-Yosef, N., Wool, A.: Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In: Filipe, J., Obaidat, M. (eds.) E-business and Telecommunications, Communications in Computer and Information Science, vol. 23, pp. 162–174. Springer Berlin Heidelberg (2009)
3. Ben-Porat, U., Bremler-Barr, A., Levy, H., Plattner, B.: On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks. In: Networking (1). Lecture Notes in Computer Science, vol. 7289, pp. 135–148. Springer (2012)
4. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD) (Jan 2005), `http://www.ietf.org/rfc/rfc3986.txt`, updated by RFCs 6874, 7320
5. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM 13(7), 422–426 (1970)
6. Cai, X., Gui, Y., Johnson, R.: Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In: IEEE Symposium on Security and Privacy (S&P 2009. pp. 27–41. IEEE Computer Society, Oakland, California, USA (2009)
7. Carter, L., Wegman, M.N.: Universal Classes of Hash Functions (Extended Abstract). In: ACM Symposium on Theory of Computing - STOC. pp. 106–112. ACM, Boulder, CO, USA (May 1977)
8. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In: Proceedings of the 19th International Conference on World Wide Web. pp. 281–290. WWW '10, ACM, New York, NY, USA (2010)
9. Crosby, S.A., Wallach, D.S.: Denial of Service via Algorithmic Complexity Attacks. In: USENIX Security Symposium. pp. 3–3. Lecture Notes in Computer Science 7668, USENIX Association, Washington, USA (December 2003)
10. Félegyházi, M., Kreibich, C., Paxson, V.: On the Potential of Proactive Domain Blacklisting. In: LEET (2010)

11. Inc., G.: Safe Browsing API. https://developers.google.com/safe-browsing/
12. Inc., G.: Google Transparency Report. Tech. rep., Google (June 2014), http://bit.ly/11ar4Sw
13. McAfee: McAfee Site Advisor. http://www.siteadvisor.com/
14. McIlroy, M.D.: A Killer Adversary for Quicksort. Softw. Pract. Exper. 29(4), 341–344 (Apr 1999)
15. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., 1st edn. (1996)
16. Microsoft: Windows SmartScreen Filter. http://windows.microsoft.com/en-us/windows/smartscreen-filter-faq#1TC=windows-7
17. Mogul, J., Krishnamurthy, B., Douglis, F., Feldmann, A., Goland, Y., van Hoff, A., Hellerstein, D.: Delta encoding in HTTP. RFC 3229, RFC Editor (January 2002), http://tools.ietf.org/html/rfc3229
18. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: SpyProxy: Execution-based Detection of Malicious Web Content. In: Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007 (2007)
19. Moshchuk, A., Bragin, T., Gribble, S.D., Levy, H.M.: A Crawler-based Study of Spyware in the Web. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA (2006)
20. National institute of standards and technology: Secure Hash Standard (SHS). Tech. Rep. FIPS PUB 180-4, National Institute of Standards & Technology (march 2012)
21. Nazario, J.: PhoneyC: A Virtual Client Honeypot. In: Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More. LEET'09, USENIX Association, Berkeley, CA, USA (2009)
22. Papadogiannakis, A., Polychronakis, M., Markatos, E.P.: Tolerating Overload Attacks Against Packet Capturing Systems. In: USENIX Annual Technical Conference. pp. 197–202. USENIX Association, Boston, MA, USA (June 2012)
23. Peslyak, A.: Designing and Attacking Port Scan Detection Tools. Phrack Magazine 8(453), 13 (July 1998), http://phrack.org/issues/53/13.html#article
24. Schultz, E.E.: Denial-of-Service Attack. In: Bigdoli, H. (ed.) Handbook of Information Security, vol. 3. Wiley (December 2005)
25. Symantec: Norton Safe Web. https://safeweb.norton.com/
26. Wang, Y.M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.T.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In: NDSS (2006)
27. WOT Services, L.: Web of Trust. https://www.mywot.com
28. Yandex: Yandex Safe Browsing. http://api.yandex.com/safebrowsing/